Master Thesis

Integrating Pareto Optimization into the Dynamic Programming Framework Bellman's GAP

Thomas Gatter

Informatics in the Natural Sciences

(Matr. No.: 2172346)

November 10, 2015

Research Group Practical Computer Science
Technical Faculty
Bielefeld University

Supervisors: **Dr. Cédric Saule**
Prof. Dr. Robert Giegerich

Abstract

Pareto optimization allows to combine independent objectives by computing the Pareto front of the search space, yielding a set of optima where none scores better on all objectives than any other. Recently, it could be shown that Pareto optimization seamlessly integrates with algebraic dynamic programming, suggesting the operation of a “Pareto product algebra“.

In this thesis, such a product was implemented in the dynamic programming framework Bellman’s GAP, allowing the programmer to obtain Pareto optimization with respect to separately given objectives by a single keystroke. Code generation for Pareto operations in the Bellman’s GAP compiler faces many alternatives. This work explores and implements several new strategies to compute Pareto fronts. In sorted implementations, the search space is kept ordered based on the objective functions. In Pareto eager implementations, the search space is kept as Pareto fronts at all times. Such implementations are not trivial to achieve and face many problems in practice. Combinations with other “product algebras” are not always easily possible.

For sorted implementations, six different algorithms were tested against a standard Quicksort. While evaluation of sorting operations on random data gives mixed results, their use in the context of dynamic programming applications favours a sorting algorithm we call *Merge In-Place*. For Pareto eager implementations, three new algorithms are introduced to combine Pareto fronts, the most efficient of them joining two-dimensional fronts in linear time and space. A new algorithm was added and successfully tested to improve runtimes on Pareto definitions with more than two dimensions.

All implementations were benchmarked against each other on a set of biologically motivated tasks. Our work shows that no strategy outperforms all others in general. The nature of the search space decomposition has a major impact on the relative performance of algorithm variants. A “naive”, unsorted implementation can be shown to perform best in most cases, however.

Several code generation techniques for Pareto optimization are now provided with the Bellman’s GAP compiler. Bellman’s GAP is available at:
`http://bibiserv.cebitec.uni-bielefeld.de/gapc`

Contents

List of Algorithms

List of Figures

List of Tables

1 Introduction

Dynamic programming is an optimization method for solving complex problems by combining (stored) sub-solutions of the same type to a solution of the bigger problem. In many cases, dynamic programming allows to solve combinatorial optimization problems over a search space of exponential size in polynomial space and time [1]. It was developed by Richard Bellman in the 1950s, who formulated a set of properties an objective function has to fulfil so that dynamic programming can be used to solve an optimization problem [2] (see Definition 2.5). Traditionally, dynamic programming algorithms are formulated as matrix recurrences with case distinctions over multiple tables storing sub-solutions. This approach complicates both the construction and the analysis of algorithms.
Search space definitions and choices of objective function are co-dependent and directly connected to subjects of code efficiency. Implementations are error prone, hard to optimize and often almost impossible to debug. For many applications, one is not only interested in the result of the optimization problem, but also in the structure of the candidate, creating the need for rigid backtracing algorithms that compute the sequence of optimization steps back from a global solution. Even small changes can lead to a full reimplementation of an algorithm.

1.1 Bellman's GAP

In 2004, Giegerich et al. set the foundation for the discipline of Algebraic Dynamic Programming (ADP) [3], a framework for developing dynamic programming algorithms over sequential data. It provides a clear separation of the issues of search space construction, tabulation and scoring, removing error-prone indices completely out of the perspective of the user. The first generation of ADP was realized as an embedded, domain-specific language in Haskell [4]. In 2006, during the course of his doctorate, Peter Steffen created the first compiler to compile ADP programs to imperative programming languages, utilizing a Haskell dialect for problem definitions [5].
Bellman's GAP is a second generation implementation of ADP, aiming to remove the restrictions imposed by Haskell and to extend the existing framework [6, 7, 8]. Together with ADPfusion [9], implemented in Haskell, it marks the latest development of ADP. In its current version, Bellman's GAP is able to generate C++ code that is competitive to handwritten programs out of a user created definition file. As such, it is a straightforward process to extend, integrate, and optimize new algorithms for any application, regardless if high level definitions or low level memory manipulation is needed. Generated code can be trivially modified for code profiling or logging. Within the code generation process, functions and structures can be defined with arbitrary arities as needed. With these positive properties, Bellman's GAP lends itself for the investigation of dynamic programming problems. Due to its iterative character, however, redefinitions of more general concepts, such as the (functionally described) standard implementation of ADP, become very complex. The need for such modifications has not yet been considered in the framework.
Bellman's GAP consists of three essential components:

- *GAP-L* – a declarative Language similar to C and Java. All components of ADP are declared in this language, although C functions can be included in the definitions.
- *GAP-C* – an optimized compiler that translates *GAP-L* programs to efficient C++ code.

- *GAP-M* – a C++ runtime library that contains datastructures and functions for compiled GAP-L code, as well as a module providing functions for accessing energy parameters and computing energy contributions as used in RNA secondary structure prediction.

In the course of this work, all three parts were modified, although the biggest changes are limited to GAP-C.

1.2 Pareto Optimization

One of the remaining problems for many ADP applications is the efficient handling of multi-objective optimization problems that arise when more than one criterion is used to evaluate the search space. Both from a theoretical and a practical perspective, it is useful to express the objective function of an optimization problem as the combination of a *choice function* φ and a *scoring function* σ, $\psi = \varphi \circ \sigma$. Overall, solutions are computed as their composite $(\varphi \circ \sigma)(X) = \varphi(\{\sigma(x) | x \in X\})$ over a search space X. Most applications employ a scoring or cost function as σ that evaluates candidates to a numerical value, φ choosing optimal candidates by minimization, maximization, or computing candidates within a threshold of optimality. Many other useful types of choice functions exist within the context of ADP, including, but not limited to, stochastic sampling, search space enumeration, candidate counting, or the computation of score sums.

When confronted with two or more objective functions, the question on how to combine them arises. For now, examples will be limited to two-dimensional cases to simplify definitions, although redefinitions for higher dimensions are within reach in many cases. Let us consider two objective functions $\psi_1 = \varphi_1 \circ \sigma_1$ and $\psi_2 = \varphi_2 \circ \sigma_2$ defined over the same search space that are combined with a product $*$.

Additive combination, Parametrized additive combination A classical approach is the additive combination of all scoring functions into a new combined score, possibly adding additional parameters to relate between them. As a precondition for this, however, the corresponding choice functions need to be identical $\varphi = \varphi_1 = \varphi_2$, restricting the areas of potential applications. For two dimension, we define

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

$$\psi_1 *_{+\lambda} \psi_2 = \varphi \circ (\lambda\sigma_1 + (1 - \lambda)\sigma_2), 0 \le \lambda \le 1. \tag{2}$$

For all such instances, the relationship between the scoring functions should be regarded and closely examined on its implications for the combined score. Eq. 1 should be used for instances where both scores imply real costs of the same type that ultimately sum up, disregarding their origin. The λ factor of Eq. 2 is used to relate between different kinds of scores. In many cases, it is not clear how to choose λ optimally, and the parameter has to be trained from a dataset or the choice is left entirely to the user. Different choices will often result in different results of the optimization problem, obscuring the "actual truth" of the problem. While this method works well in practice, this often artificial choice of λ leaves room for improvement.

Lexicographic combination In 2005, Peter Steffen and Robert Giegerich suggested the use of Lexicographic combinations in ADP [10], defining a primary and a secondary objective for optimization:

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

This method is useful if σ_1 returns multiple co-optimal candidates that σ_2 can choose from. Refining the problem this way is often preferable to returning a large number of solutions or choosing among them arbitrarily. Lexicographic combinations have a wide array of applications outside of direct optimization, most noticeably for combining candidate string representations to (numeric) optimization results and for use in classified dynamic programming [11].

Pareto combination If no meaningful way exists to combine or prioritize objective functions, another mean of optimization is needed. This case arises, for example, when the choice functions are different $\varphi_1 \neq \varphi_2$, but both objective functions are equally important. As an additional factor to consider, the reduction to just one result is not always advantageous. Returning a list of co-optimals among all dimensions might be more comprehensive in many cases. Pareto optimization is a common method for multi-objective optimization [12]. Recently, Cédric Saule and Robert Giegerich proposed the use of Pareto combinations in ADP [13]. Pareto optimality is defined by the concept of domination, implying an ordering on all dimensions that is indirectly defined by the applied choice functions. A candidate dominates another one if it is strictly better in one dimension, and equal or better in all other dimensions. A Pareto front is defined as a set of non-dominated elements. The result of a Pareto combination is the biggest possible Pareto front of all tuples of values over all dimensions.

$$\mathbf{pf}(S) = \{(a,b) \in S \mid \nexists (a',b') \in S \setminus \{(a,b)\} \text{ such that } (a',b') \text{ dominates } (a,b)\} \quad (4)$$

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

In Sections 2.2 and 4, a more comprehensive definition of Pareto optimization in the context of ADP will be given and a series of algorithms to compute Pareto fronts will be highlighted respectively.

1.3 Dynamic Programming in Bioinformatics

In the context of bioinformatics, dynamic programming is one of the most prevalent paradigms [14]. Many classical and modern problems of sequence analysis can be expressed as decomposable optimization problems and as such, can be solved using methods of dynamic programming or ADP.

One of the cornerstones of biosequence analysis is the detection of regions of homology in pairs of sequences. For this task, different edit operations are defined by assigned scores. An optimal alignment is then defined as the sequence of operations either minimizing or maximizing the score, depending on the problem. Examples include the famous Needleman–Wunsch algorithm [15] and Goto's algorithm [16]. Both are good examples for additive combinations of objective functions, combining different respectively distance or similarity values into one score. In order to create meaningful results, individual scores have to be balanced based on known datasets, however.

A second common application is the field of RNA structure analysis. Using dynamic programming, the search space of all possible foldings of an RNA molecule can be analysed and optimized to some specific measurement of accuracy. Examples for structure prediction analysis are manifold, including both the folding of single sequences as well as that of sequence alignments [17]. A variant poses the classic Sankoff problem, optimizing sequence alignment scores (ψ_1) and basepairing as a measurement for structure stability (ψ_2) at the same time as a parametrized additive combination. As a more recent application, RNAalifold combines free energy and covariance scores [18, 19]. Other areas of

RNA structure analysis include the restriction of the search space to specific motives for detection in larger sequences [20].

Pareto optimization has been used in bioformatics mainly in a heuristic fashion for evaluating candidate subsets created by genetic algorithms. Only few examples exist where Pareto optimization was used in the context of dynamic programming. Examples include the shortest path problem [21] and the Allocation Problem [22]. Even fewer examples exist in the context of bioinformatics. Schnattinger et al. [23, 24] promoted the use for solving the Sankoff problem. Libeskind-Hadas et al. [25] used Pareto optimization to compute reconciliation trees in phylogeny.

1.4 Problem Statement

The stated goals of this work are manifold, combining all aspects of the introduction:

- Next to identifying different algorithms for computing Pareto fronts, Saule and Giegerich reported two alternative definitions of ADP in combination with Pareto optimization, respectively called lexicographically sorted and Pareto eager implementation [13]. Instead of restricting changes to these implementations, the aim of this work is to create a generalized interface for changes to the ADP definition.
- Subsequently, the described implementations, lexicographically sorted ADP and Pareto eager ADP, will be realized along the new interface. Pareto optimization of two or higher dimensions will be implemented. If applicable and the best optimization is not directly discernible, multiple versions of algorithms will be implemented.
- Additionally, algorithms to efficiently compute higher-dimensional fronts in the standard formulation of ADP are discussed and implemented.
- Lastly, all algorithms are benchmarked to choose the best implementations that, finally, will be compared against each other on four applications of bioinformatics.

Prior to this work, two-dimensional, as described by Saule and Giegerich [13], and selected implementations for higher-dimensional Pareto operators have already been introduced into the Bellman's GAP framework together with the replacement of several code objects. The essence of some of this work will be highlighted here again as it closely relates to the new content of this thesis.

The rest of this thesis is organized as follows: In Section 2, basic definitions of ADP and the integration of products are described, followed by an analysis of the specialised ADP implementations. Afterwards, in Section 3, the integration of Pareto products and the generalized implementation interface is presented, followed by a description of viable algorithms for computing Pareto fronts in Section 4. In Section 5, the influence of algebra products on the definitions of lexicographically sorted ADP and Pareto eager ADP will be analysed. The realizations of lexicographically sorted ADP and Pareto eager ADP are given in Sections 6 and 7 respectively. Finally, all benchmarks and a final conclusion will be presented in Sections 8 and 9.

The main results of this thesis have been summarized in an article currently in submision to the Algorithms journal.[1] The Appendix of this version of the thesis has been modified to also contain the notations used in this paper.

[1]T. Gatter, R. Giegerich and C. Saule, "Integrating Pareto Optimization into Dynamic Programming"

2 Definitions

Before progressing to the details of the practical implementation, it is important to recall the basic definitions of ADP and its domains. This section will closely follow the definitions as presented in [6] and [13].

2.1 Algebraic Dynamic Programming

Each ADP problem is defined by three main components.

Definition 2.1 *(Signature, Evaluation Algebra, Regular Tree Grammar)* A *signature* Σ is a set of function symbols and a data type place holder S. Each $f \in \Sigma$ has a return type S, and arguments of each either S or a fixed alphabet $\mathcal{A}$. T_Σ denotes the term language described by the signature Σ, and $T_\Sigma(V)$ is the term language with additional variables from a set V. An *evaluation algebra* defines a function for each $f \in \Sigma$ as well as an objective function $\varphi_\text{A} : [S] \to [S]$, square brackets denoting multi-sets. A carrier set S_A is set for S. $A(t)$ gives a result in S_A for each $t \in T_\Sigma$. Finally, a *regular tree grammar* $\mathcal{G}$ is defined over a signature Σ as a Tuple $(V, \mathcal{A}, Z, P)$ where V is the set of non-terminal symbols, $\mathcal{A}$ is an alphabet, Z is the axiom, and P a set of production rules in the form of

$$v \to t \text{ with } v \in V, t \in T_\Sigma(V). \tag{6}$$

The language of a tree function is then defined as

$$L(\mathcal{G}) = \{t \in T_\Sigma \mid Z \to^* t\}. \tag{7}$$

Definition 2.2 *(Yield Function)* The *yield function* y is of type $y : T_\Sigma \to \mathcal{A}^*$, with $y(a) = a$ and $y(f(x_1, \dots, x_n)) = y(x_1) \dots y(x_n)$, for all $f \in \Sigma$ and $a \in \mathcal{A}$.

These definitions now allow us to define the full ADP problem.

Definition 2.3 *(Search Space)* Given an *input sequence* $z \in \mathcal{A}^*$, the search space is defined over the combination of a yield function y and a tree grammar $\mathcal{G}$.

$$X_z = \{x \in L(\mathcal{G}) \mid y(x) = z\} \tag{8}$$

Definition 2.4 *(ADP Problem Solution)* Given a specific tree grammar $\mathcal{G}$ and evaluation grammar A, an ADP problem is solved by computing

$$\mathcal{G}(A, x) \coloneqq \varphi_A([A(x) \mid x \in X_z]). \tag{9}$$

In the context of Bellman's GAP, the definitions of $\mathcal{G}$ and A are given in the GAP-L language. GAP-C then creates a C++ program solving the ADP problem. For this, search space construction of X, evaluation of $A(X)$, and the choice from it with φ_A are combined into one algorithm. In order for a dynamic programming program to yield correct results, the evaluation algebra A must satisfy Bellman's Principle of Optimality.

Definition 2.5 *(Bellman's Principle of Optimality)*

$$\begin{aligned} \varphi_A[f_A(x_1, \dots, x_k) | x_1 \leftarrow X_1, \dots, x_k \leftarrow X_k] = \\ \varphi_A[f_A(x_1, \dots, x_k) | x_1 \leftarrow \varphi_A(X_1), \dots, x_k \leftarrow \varphi_A(X_k)] \end{aligned} \tag{10}$$

$$\varphi_A[X_1 \cup X_2] = \varphi_\text{A}(\varphi_\text{A}(X_1) \cup \varphi_A(X_2)) \tag{11}$$

$$\varphi_A[] = [] \tag{12}$$

2.2 Products

While not strictly needed in the concept of ADP, products of algebras are of a great practical value, bridging the gap to multi objective optimization without manually defining a combined algebra.

Let us define $C = A \times B$ as the Cartesian product of two algebras. The solution of any algebra product is a subset $X \subseteq C$. Analogous definitions can be found for combinations of more than two products, accordingly resulting in an k-ary Cartesian product over k algebras. In Bellman's GAP, various products are defined, but only two of them are of interest for this work. From the combinations introduced in Section 1.2, only lexicographic and Pareto products have been implemented so far, the second of which as a part of this work.

Definition 2.6 *(Product of Algebra Functions)* For all variants of products, the same basic definition holds for two algebras A and B.

$$f_{\mathrm{A}\,*\,\mathrm{B}}((a_1,b_1),\dots,(a_m,b_m)) = (f_{\mathrm{A}}(a_1,\dots,a_m), f_{\mathrm{B}}(b_1,\dots,b_m)) \tag{13}$$

Definition 2.7 *(Lexicographic Product)* Given a function *set*(X) that reduces a multiset X to a set and two evaluation algebras A and B over the same signature Σ, the *lexicographic product* $A *_{\mathrm{lex}} B$ is defined as the functions $f_{A *_{\mathrm{lex}} B}$ as in Definition 2.6 for all $f \in \Sigma$ and the choice function

$$\begin{aligned} \varphi_{A *_{\mathrm{lex}} B}[(a_1,b_1),\dots,(a_m,b_m)] \quad & (14)\\ = [(l,r) \mid \quad & \\ l \in \mathit{set}(\varphi_{\mathrm{A}}[a_1,\dots,a_{\mathrm{m}}]), \quad & (14a)\\ r \leftarrow \varphi_B[r' | (l',r') \leftarrow [(a_1,b_1),\dots,(a_{\mathrm{m}},b_{\mathrm{m}})], l' = l]] \quad & (14b) \end{aligned}$$

The definition of Pareto products is more complicated, as the original choice functions can no longer be directly used. In Section 1.2, the notion of domination was already introduced, but it was left open how exactly it was defined within ADP. Setting C as above, A and B each must define a total order on its candidates in order to be suitable for a Pareto product. For simplicity, in the context of this work, it will be assumed that choice functions that are part of a Pareto product either maximize or minimize over a total order. For this case, Saule and Giegerich could prove that the Pareto product preserves Bellman's principle [13]. Indirectly, defined over the choice functions φ_A and φ_B, A and B each are totally ordered by relations $>_A$ and $>_B$ respectively. We say $(a,b) \in C$ dominates $(a',b') \in C$ if $a \geq_A a'$ and $b >_B b'$ or $a >_A a'$ and $b \geq_B b'$. This sparks a redefinition of the Pareto front operator as

Definition 2.8 *(Pareto front operator)*

$$\mathbf{pf}_{>_A,>_B}(X) = \{(a,b) \in X \mid \nexists\, (a',b') \in X \setminus \{a,b\} \text{ with } a \leq_A a', b \leq_B b'\}. \tag{15}$$

However, this notation is somewhat cumbersome, and becomes even longer for higher-dimensional products. Therefore, these details will be suppressed, and **pf** will be used instead regardless of arity or underlying orders.

Definition 2.9 *(Two-Dimensional Pareto Product)* Given two evaluation algebras A and B over the same signature Σ, the *Pareto product* $A *_{\mathrm{Par}} B$ is defined as the functions $f_{A *_{\mathrm{Par}} B}$ as in Definition 2.6 for all $f \in \Sigma$ and the choice function

$$\varphi_{A *_{\mathrm{Par}} B}[(a_1,b_1),\dots,(a_m,b_m)] = \mathbf{pf}([(a_1,b_1),\dots,(a_m,b_m)]) \tag{16}$$

The Pareto front size for a random set X of size N is expected as $H(N)$, where H is the harmonic number and closely related to $\log(N)$ [26, 27]. This interacts in a fortunate way with dynamic programming, where for input size n, the search space X grows with $O(2^n)$, and we can expect Pareto fronts of size $O(n)$. This has been confirmed in practice in [13].

Given all previous definitions, generalizing the Pareto product for higher dimensions is a straightforward process.

Definition 2.10 *(Pareto Product)* Given $k \geq 2$ evaluation algebras $A_1, \dots A_k$ over the same signature Σ, the *Pareto product* $*_{\text{Par}}\{A_1, \dots A_k\}$ is defined as the functions

$$\begin{aligned} &f_{*\{A_1,\dots,A_k\}}((a_{1,1},\dots,a_{k,1}),\dots,(a_{1,m},\dots,a_{k,m})) \\ &\quad = (f_{A_1}(a_{1,1},\dots,a_{1,m}),\dots,f_{A_k}(a_{k,1},\dots,a_{k,m})) \end{aligned} \tag{17}$$

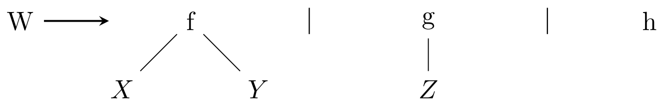
for all $f \in \Sigma$ and the choice function

$$\begin{aligned} &\varphi_{*_{\text{Par}}\{A_1,\dots A_k\}}[(a_{1,1},\dots a_{k,1}),\dots,(a_{1,m},\dots a_{k,m})] \\ &\quad = \mathbf{pf}([(a_{1,1},\dots a_{k,1}),\dots,(a_{1,m},\dots a_{k,m})]) \end{aligned} \tag{18}$$

It is important to note that $*_{\text{Par}}\{A_1, \dots A_k\} \neq A_1 *_{\text{Par}} \dots *_{\text{Par}} A_k$ for all cases except $k = 2$ as Pareto products are not associative. The intuition for this is fairly simple, as the result of a Pareto product no longer defines any order on the result set and therefore can not be combined with another Pareto product. Hence, each k describes a unique product. In Section 3.1, we will see how this is handled in practice.
For $d > 2$ dimensions, the Pareto front size for a random set X of size N is expected as $H^{(d)}(N)$, where $H^{(d)}$ is the generalized harmonic number and closely related to $log^{d-1}(N)$ [27]. Hence, increasing the dimension increases the expected front size exponentially.

2.3 Implementation

As can be seen by the definition, the Pareto product can be implemented simply by providing a Pareto front operator as the choice function. We will see in Section 4 on how these operators can be implemented in practice. For now, these details are not important. Nevertheless, it should be noted that Pareto fronts can be computed more efficiently for lexicographically sorted lists (according to the order imposed by the choice functions). For now, we will have a look at different implementation strategies in the broader context of ADP algorithms.
Following the example and definitions of [13], we will describe the implementation options on an example production that covers all relevant cases.
Let f, g, and h be a binary, a unary, and a nullary function in Σ. The tree grammar rule



specifies the production of subresults of type W from subresults of types X, Y, and Z, as well as an empty subproblem h, each computing a (for h constant) list of candidates. This combination process can be defined by the introduction of three operators $\otimes, \oplus$, # , respectively called “extend”, “combine” and “select”. We append lists of solutions with

$\oplus$ and $\otimes$ extends solutions from smaller subproblems to bigger ones. Generally, W is constructed as

$$W = (\otimes(f, X, Y) \;\oplus\; \otimes(g, Z) \;\oplus\; h) \# \mathbf{pf}, \tag{19}$$

although h and **pf** must be more closely defined for sorted and Pareto eager implementations.
Beyond this example, algebra functions can have arbitrary arity and tree grammar rules can have arbitrary height, but these cases can be handled analogously. Including a binary operator, however, is important as it – representative for all operators of arity two and above – does not conserve the Pareto property and is therefore more difficult to handle.

If there are no requirements on the search space to be sorted, the *standard implementation* of ADP can be used:

$$\begin{aligned} l \# \mathbf{pf} &= \mathbf{pf}(l) && (20)\\ l_1 \oplus l_2 &= l_1 {+}{+} l_2 && (21)\\ \otimes(f, X, Y) &= [f(x, y) \mid x \in X, y \in Y] && (22)\\ \otimes(g, X) &= [g(x) \mid x \in X] && (23) \end{aligned}$$

In this context, ++ denotes a simple list concatenation for $\oplus$. The operator $\otimes$ extends the set of solutions by applying the respective function of the evaluation algebra to combined subsolutions.

If the Pareto operator requires a sorted input list, this version can by easily adapted by changing Eq. 20 to apply a sorting function *sort* before applying the Pareto product. The rule then becomes

$$l \# \mathbf{pf} \;=\; \mathbf{pf}(sort(l)). \tag{24}$$

While this is simple to implement, a more specialized version exists for sorted lists.

2.3.1 Lexicographically Sorted Implementation

In the lexicographically sorted implementation, operators are chosen in a way that all intermediate results are kept as lexicographically sorted lists, from now on just referred to as sorted. Two sorted lists can be merged in linear time by a function denoted as *merge*. The function *foldr* executes a function given as first argument iteratively on the elements of a list, using the provided second element as initial value.

$$\begin{aligned} l \# \mathbf{pf} &= \mathbf{pf}(l) && (25)\\ l_1 \oplus l_2 &= merge(l_1, l_2) && (26)\\ \otimes(f, X, Y) &= foldr \quad merge \quad [] \; [[f(x, y) \mid x \leftarrow X] \mid y \in Y] && (27)\\ \otimes(g, X) &= [g(x) \mid x \leftarrow X] && (28)\\ h &= sort(h) && (29) \end{aligned}$$

Eq. 27 iteratively merges the elements of a list of sorted lists. Please note that applying any function of the evaluation algebra, here $f, g \in \Sigma$, to a sorted list will again result in a sorted lists, as any such function is required to be strictly monotone by Bellman's principle

[13]. y is to be assumed fixed for every computation of a sorted sublist $[f(x,y) \mid x \leftarrow X]$, indicated by the formulation as $y \in Y$. Eqs. 27 and 28, therefore, result in sorted lists again. Constant functions are required to create sorted results as initial values. In reality, constant functions only create single elements that are naturally already sorted.

2.3.2 Pareto eager Implementation

In a more complicated approach, instead of constructing a list of all intermediate results first, Pareto operators can be applied at each intermediate step. The rationale behind this idea is to reduce intermediate list sizes as early as possible, potentially cutting down on the overall runtime. For this, Pareto operators are included in the $\oplus$ and $\otimes$ phases. We will later see that two Pareto fronts can be merged into a combined Pareto front in $O(l)$ time for two-dimensions, and in $O(l \log^{d-1} l)$ for d-dimensional products, where l is the combined size of both fronts to join. We call this function $\overset{p}{\vee}$.

$$\begin{aligned}
l \;\#\; \mathbf{pf} &= l && (30)\\
l_1 \oplus l_2 &= l_1 \overset{p}{\vee} l_2 && (31)\\
\otimes(f,X,Y) &= foldr \overset{p}{\vee} [] \; [[f(x,y) \mid x \leftarrow X] \mid y \leftarrow Y] && (32)\\
\otimes(g,X) &= [g(x) \mid x \leftarrow X] && (33)\\
h &= \mathbf{pf}(h) && (34)
\end{aligned}$$

Much like for the sorted implementation, h must create a Pareto front as an initial result. In practice, constant functions usually create only single elements that are by definition also Pareto fronts. By the same arguments as before, applying a function of the evaluation algebra to an intermediate list that constitutes a Pareto front, the result will again be a Pareto front. Since the choice function is already applied at individual steps in Eqs. 31, 32 and 34, $\#$ can simply return the identity.

2.3.3 Generalized Implementation

Following the previous implementations, the generalization is defined by three functions we denote as *nullary*, *append* and *choice*, respectively applied to constant functions, to combine results, or as a choice function. The essence of the $\otimes$ steps remains unchanged, as they represent the core functionality of the ADP problem, combining solutions of smaller subproblems into solutions of bigger subproblems. While specific choices for *nullary*, *append* and *choice* can certainly violate the properties of ADP, changing the list generators definitely will.

$$\begin{aligned}
l \;\#\; \mathbf{pf} &= choice(l) && (35)\\
l_1 \oplus l_2 &= append(l_1, l_2) && (36)\\
\otimes(f,X,Y) &= foldr \quad append \quad [] \; [[f(x,y) \mid x \leftarrow X] \mid y \in Y] && (37)\\
\otimes(g,X) &= [g(x) \mid x \leftarrow X] && (38)\\
h &= nullary(h) && (39)
\end{aligned}$$

For the sorted and Pareto eager implementation, it is easy to see how each function needs to be defined. To define again the standard implementation, we need to set *nullary* to the identity function, $choice = \mathbf{pf}$ and $append = $ **++**, as Eq. 22 indirectly contains $foldr + +$ by removing the inner brackets of the otherwise nested list definition.

While mathematically already well defined, this definition leaves open many practical questions. With the presence of potential other algebra products and arbitrarily $k\!-\!ary$ Pareto products, the functions *nullary* and *choice* are potentially complex to construct. A similar problem arise for *append*, as it can depend on the choice functions of the underlying algebras. Most interesting however, is the implementation of $foldr$, that can be interpreted differently outside of the context of lazy evaluation functional programming. As an additional concern, these problems should not be in the hands of the user, but rather should be solved by the ADP implementation generically. In the context of Bellman's GAP, this means that GAP-L should not be modified to accommodate the generalized implementation and its realizations. Instead, the user should be able to switch between implementations with simple compiler flags of GAP-C. This puts additional strain on the compiler. In the next section, we will describe in detail on how this problem is handled.

In the course of this work, the term *generalized implementation* will be used to reference this definition in general, and the sorted and Pareto eager implementation in particular.

3 Integration in Bellman's GAP

The integration of new concepts into the Bellman's GAP system can be a daunting task, as new complex constructs need to be embedded into a system of tight dependencies. While the definitions of the last section imply a three phase computation (search space construction of X, evaluating the algebra $A(X)$ and application of the choice function), Bellman's GAP does not make a clear distinction of these phases. Identifying the correct places to insert changes is often not a trivial task. In theory, GAP-C was designed to generate code for arbitrary (imperative) programming languages [6]. As we will see in Section 3.6, the GAP-C compiler does not directly generate code, but rather creates an object tree representation of the program first, adding another layer of abstraction to the problem. As a direct result of this, for example, no C specific concepts can be directly integrated.
Lingering over everything is the demand for optimization. In many cases, it is not simply enough to add functionality, but rather to bridge to the gap between the usually contrasting objectives of maintainability, generality, and optimality, while keeping concepts broad enough to also apply to programming languages outside of the C context.
We will approach this task in multiple steps, first explaining the concepts of implementations before addressing the actual changes within the Bellman's GAP system.

3.1 Arbitrary Dimensional Products

Arbitrarily dimensional products pose a unique challenge for any ADP system, as variadic functions can cause various problems for implementations. Within the Bellman's GAP framework, functions of the compiled code can be generated as needed during the code generation process. Varying arities, therefore, do not pose any directly visible problems other than their representation within the compiler. The GAP-C compiler itself is written in C++. As an extension of C, C++ supports basic variadic functions in all versions. With the C++ standard, also template support for variadic functions has been introduced. Following this rationale, at least in theory, products of any dimension could be expressed as single objects of the same class within the program code. The practicality, however, is questionable.
A similar question arises when considering the definitions of the GAP-L language. In its original definition, products can solely be defined via infix notation, allowing only two algebras to be combined by one product at the same time, setting a hard limit for dimensionality. In the previous section, it was already established that a Pareto product cannot be combined to the results of another Pareto product, as Pareto products are non-associative and this will no longer pose a well defined problem. Effectively, this results in the need for a special operator for each arity, a problem that could be easily remedied by the introduction of an (alternative) prefix notation for products.
While mathematically elegant, parsing a prefix operator to a single representing object is not a simple task and should only be attempted if it is of value to the problem. When analysing the structure of GAP-C, one will notice that the overall architecture was built to create code for only two-dimensional products, thus handling them as binary trees throughout the program. In fact, this tree-like structure remains visible in the generated program code. All signature and choice functions are created individually for each algebra product and each algebra (at the leaves of the tree), each respectively calling their children functions in the tree to compute their own results. Ultimately, this effect can also be seen

in the storage of candidates in the solution list, each product representing a tuple in a nested type structure. For now, letting aside the actual notation in GAP-L and the representation of tuples in C++ code, let's say we have algebras $A_1, \dots, A_5$ with return types of $S_{A_1}, \dots, S_{A_5}$ respectively. The product combination

$$(A_1 *_{\text{Par}} A_2) *_{\text{lex}} (A_3 *_{\text{lex}} A_4 *_{\text{lex}} A_5)$$

will then result in an overall candidate type of

$$((S_{A_1}, S_{A_2}), ((S_{A_3}, S_{A_4}), S_{A_5})).$$

The build $\otimes$ and combine $\oplus$ phases are constructed so that they will create lists of candidates over all algebras at the same time, accordingly calling the root functions of the product tree. During execution, for each product along the tree, the outer tuple is dropped, and the left and right elements are passed down to the respective children along the tree. After all children return their results, the result of each product is computed according to its definition. For our example, the leftmost lexicographic product would split up the problem into tuples

$$(S_{A_1}, S_{A_2}) \text{ and } ((S_{A_3}, S_{A_4}), S_{A_5})$$

and join the results of the left-hand Pareto combination and the right-hand other lexicographic products. It should be noted that no elements are actually copied during this process, but rather pointers on substructures of the same candidate list are passed down. While very efficient, this behaviour can be critical, as we will see later on.
Choice functions of Pareto products play a special role in this process, as they in fact can not simply call the choice functions represented by their children on the passed down substructure lists. Instead, the choice functions are called to compare individual pairs of candidates (see Sections 3.2 and 3.6.1).
Introducing k-ary nodes with $k > 2$ would result in a massive change of GAP-C, most noticeably in the structural layout and the generation of candidate lists. By introducing a new k-tupel datastructure, the code generation of practically all signature and choice functions would need to be changed to accept this datatype. Additionally, new generators would have to be implemented to create functions for the new products that do not rely on a binary call hierarchy. We introduce the following theorem, showing us that this effort is not needed for most functions.

Theorem 3.1 *(Relation of Algebra Products)* Let us define $k \geq 2$ evaluation algebras $A_1, \dots, A_k$ over the same signature Σ and $C_{i,j} = A_i * \dots * A_j$ with $1 \leq i < j \leq k$ and arbitrary parentheses. Then for any algebra function $f \in \Sigma$ the following holds: for any combination of parentheses in the product $C_{1,k} := A_1 * \dots * A_k$, $f_{C_{1,k}}$ can be transformed to $f_{*\{A_1,\dots,A_k\}}$ by a series of applications of functions $rbleft\{((x_1), x_2)\} = (x_1, x_2)$ and $rbright\{(x_1, (x_2))\} = (x_1, x_2)$.

Proof. By induction:
$k = 2$:

$$\begin{aligned} & f_{A_1 * A_2}((a_{1,1}, a_{2,1}), \dots, (a_{1,m}, a_{2,m})) \\ \overset{\text{Eq. 13}}{=} \; & (f_{A_1}(a_{1,1}, \dots, a_{1,m}), f_{A_2}(a_{2,1}, \dots, a_{2,m})) \\ \overset{\text{Eq. 17}}{=} \; & f_{*\{A_1,A_2\}}((a_{1,1}, a_{2,1}), \dots, (a_{1,m}, a_{2,m})) \end{aligned}$$

$k > 2$:

$$\begin{aligned}
& rbleft\left\{f_{C_{1,k-1}*A_k}((c_1, a_{k,1}), \ldots, (c_m, a_{k,m}))\right\} \\
\overset{\text{Eq. 13}}{=}\; & rbleft\left\{\left(f_{C_{1,k-1}}(c_1, \ldots, c_m), f_{A_k}(a_{k,1}, \ldots, a_{k,m})\right)\right\} \\
\overset{\text{I.H.}}{=}\; & rbleft\left\{\left(f_{*\{A_1,\ldots,A_{k-1}\}}((a_{1,1}, \ldots, a_{k-1,1}), \ldots, (a_{1,m}, \ldots, a_{k-1,m})), f_{A_k}(a_{k,1}, \ldots, a_{k,m})\right)\right\} \\
\overset{\text{Eq. 17}}{=}\; & rbleft\left\{\left((f_{A_1}(a_{1,1}, \ldots, a_{1,m}), \ldots, f_{A_{k-1}}(a_{k-1,1}, \ldots, a_{k-1,m})), f_{A_k}(a_{k,1}, \ldots, a_{k,m})\right)\right\} \\
=\; & (f_{A_1}(a_{1,1}, \ldots, a_{1,m}), \ldots, f_{A_k}(a_{k,1}, \ldots, a_{k,m})) \\
\overset{\text{Eq. 17}}{=}\; & f_{*\{A_1,\ldots,A_k\}}((a_{1,1}, \ldots, a_{k,1}), \ldots, (a_{1,m}, \ldots, a_{k,m}))
\end{aligned}$$

or

$$\begin{aligned}
& rbright\left\{f_{A_1*C_{2,k}}((a_{1,1}, c_1), \ldots, (a_{1,m}, c_m))\right\} \\
\overset{\text{Eq. 13}}{=}\; & rbright\left\{\left(f_{A_1}(a_{1,1}, \ldots, a_{1,m}), f_{C_{2,k}}(c_1, \ldots, c_m)\right)\right\} \\
\overset{\text{I.H.}}{=}\; & rbright\left\{\left(f_{A_1}(a_{1,1}, \ldots, a_{1,m}), f_{*\{A_2,\ldots,A_k\}}((a_{2,1}, \ldots, a_{k,1}), \ldots, (a_{2,m}, \ldots, a_{k,m}))\right)\right\} \\
\overset{\text{Eq. 17}}{=}\; & rbright\left\{\left(f_{A_1}(a_{1,1}, \ldots, a_{1,m}), (f_{A_2}(a_{2,1}, \ldots, a_{2,m}), \ldots, f_{A_k}(a_{k,1}, \ldots, a_{k,m}))\right)\right\} \\
=\; & (f_{A_1}(a_{1,1}, \ldots, a_{1,m}), \ldots, f_{A_k}(a_{k,1}, \ldots, a_{k,m})) \\
\overset{\text{Eq. 17}}{=}\; & f_{*\{A_1,\ldots,A_k\}}((a_{1,1}, \ldots, a_{k,1}), \ldots, (a_{1,m}, \ldots, a_{k,m}))
\end{aligned}$$

□

Effectively, this theorem tells us two important things. For all functions $f \in \Sigma$, k-ary products produce de facto the same candidates as a series of applications of $k-1$ binary operators, as long as the overall order of algebras stays the same. Changing the implementations of algebra functions, therefore, is unnecessary. Additionally, the proof gives a constructive algorithm on how to transform between the two cases for the application in k-ary choice products. Since k is independent of the input length of the problem, the transformation can be done in $O(1)$ time.

Following this, it is sensible not to change the definitions of either GAP-L or GAP-C to specifically handle k-ary products. Instead, all products are expressed as a combination of binary operators as if such an expression was valid. GAP-C will then detect this case for choice functions (and only there), and transform them to the k-ary case where needed without changing the overall candidate structure. More specifically, multi-dimensional products will show up as a cluster of the same product type in the product tree. When generating the choice functions for each product, GAP-C will perform a depth-first search for connected products of the same type, starting at the current product as the root. This already solves the problem of Theorem 3.1 and yields the series of operations needed for the transformation between binary and k-ary products. After this, the k-ary choice function can be directly generated with minimal effort.

3.2 Comparison Objects

Another complex problem arises for the interfaces of the generalized implementation, as the definitions are potentially dependent on the choice functions of underlying algebras. For the sorted and Pareto eager implementations, the implicit order defined by the choice functions is needed for object comparisons. In practice, the question arises on how these definitions can be generated and passed to functions of the generalized implementations. In Bellman's GAP, this will be handled by comparison functions (from now on called comparators).

C++ has full support for passing function pointers as parameters. However, function pointers are problematic for performance, as they often lead to false branch predictions [28]. As the comparators are called in the inner loops of both the sorting algorithms and the ADP program, using pointers instead of direct function calls or inlined comparisons could result in a noticeable performance drop. Early test versions using function pointers showed an increase in computation time of up to a factor of 10.
To avoid this problem, it is usually recommended to pass objects instead of functions. Ideally, the object is defined constant and contains only static functions, creating predictable jumps at every stage. Such function calls still show an overhead to inlined code, but perform better than function pointers. For the Pareto implementations, as a design goal, individual comparator calls should be minimized over all code. At the same time, comparisons within each individual comparator should be minimized. These goals are inherently contradicting to each other as many algorithms don't always need to compare all dimensions at the same time, but instead need individual tests. On the other hand, whenever a comparison must include all dimensions it is impractical to call them individually. Creating code for all possible combinations over arbitrary dimensions poses a combinatorial problem, as already for 4 dimensions this would create 16 different comparators that would need to be passed and organized, although most of them would not be needed. The mechanism to call the correct function would create multiple conditional jumps, again slowing down the process. As a compromise, only two comparators will be generated in Bellman's GAP: one comparing all dimensions and one comparing a single dimension, its index passed as a parameter.

As established in the last section, candidates are created as binary structures over all products. When passing down substructures along nested function calls, only references to substructures of the original candidates are passed. Effectively, this means that only one global sorting over all candidates can be established, unrelated to the used implementation of ADP. For sorted Pareto implementations, this effectively means that only one such product can exist as two competing sortings cannot be executed on the same list of candidates. Likewise, the sorting must be completed before applying the choice function of the topmost product, regardless if it is a Pareto product or not.
Effectively, comparators therefore need to implement two steps. First, the substructure of the Pareto operator needs to be extracted out of the full candidate structure. Then the elements that are needed for comparison are extracted and compared. For both steps, the product tree serves as a guide. Starting from the global root, in a depth first fashion, the root of a Pareto product is searched, and the path to it is transformed into the operations needed to extract the corresponding candidate substructure. Then, starting at the root of the Pareto product, a breadth first search is executed to extract all dimensions by Theorem 3.1.
Comparators for all dimensions are generated as nested conditionals, only calling the next dimension if the current dimension was not sufficient to determine the order between two objects.
Comparators for individual dimensions take an index of the dimension to compare. For implementation in C++, *if* structures can be slow because of false branch predictions. It is more efficient to rely on switch case, instead of nesting if clauses, as it reduces this risk [28].

3.3 Comparing with Choice Functions

For Pareto products, the underlying choice functions are not directly applied to lists of candidates, as is the case for all other products. Instead, the choice function is used to compare only two candidates at the same time. For this, three basic cases need to be distinguished for any efficient implementation.
In most cases – if not nearly all – choice functions in Pareto products are defined as minimizing or maximizing the candidate score. With such functions, as a standard, GAP-C will generate all choices over a list of candidates, meaning it will take a list as input and return a single result. In order to find the smaller or bigger of two candidates, the two elements would need to be copied into a new list, the list passed to the choice function, and the return value compared to the original values. Irrespective of the kind of objects to be compared, however, the smaller or bigger operators of C++ could be used directly to compare them instead of using the choice function. This saves multiple memory operations and function calls per comparison.
However, when deviating from minimization or maximization, this speed up is no longer possible. Instead, the values to be compared are copied into a new list as described above. Since the choice functions should ideally define an ordering, only one element should be returned that can be taken as the result. Equal operators need to be defined for all objects, so the returned value can always be compared to the original input to find out which one was returned as better.
As a third option, not always just one element is returned by the choice function. In this case, the first element will always be used. It is up to the user to guarantee that this is in fact well defined.

3.4 Interfaces

The most challenging task in the implementation of the generalized interface is to identify the correct positions to insert the calls to *nullary*, *append*, and *choice* in the generated ADP code. We will see that *choice* and *nullary* functions will be fully generated in the code generation process, while *append* functions will be realized via templated function calls to functionality added in a header file.

Of the three functions to insert, the *choice* interface is the easiest to realize, as choice functions are already applied in the original implementation of Bellman's GAP. Naturally, this is done at the end of each production of the tree grammar (if one is set by the user). The generalized implementation can directly profit from this existing architecture. For the current use cases, only minimal change is needed for the choice function generators. In fact, for the sorted implementation no changes are needed at all. For the Pareto eager implementation, the choice function is set up to return the identity. A generator for this already existed hidden within the code.
Generating the *nullary* function is more complicated, although even here the variation is minimal for the needed Pareto implementations. For the sorted implementation, the *nullary* function must return a sorted list. Therefore, it is generated as an identity function, but with an added sorting function that sorts the given lists in place. The global comparator objects described in the last subsection are used for element comparison. In the Pareto eager case, the Pareto front of the candidate lists needs to be computed. This task is solved by applying the original choice function, and the same generation process can be used. Effectively, to generate *nullary* functions, the original choice functions are

duplicated in all algebras and then the specializations for *choice* or *nullary* functions are respectively created during code generation. The most problematic part of the implementation of *nullary* functions is the question of when to apply them. For most grammars, this is not needed at all, as technically signature functions in Bellman's GAP cannot return more than one element. Therefore, all constant functions return exactly one element which can not be influenced by a selective or ordering nullary function. If a nullary function would remove a constant single item, the rule would be useless. It is possible, however, to create candidate lists out of k-ary rules only consisting of moving index boundaries over the input sequence, hence adding a constant element for each index combination to a sublist.
Before answering the question of when to apply *nullary* in full, an analysis of how Bellman's GAP generates product rules is needed. At the same time, we will answer how the *append* operation can be included. In Section 2.3, we saw an example production of a tree grammar rule with alternative functions of different arities. All alternative productions are combined by the combine operator $\oplus$ after building up candidates by the build operator $\otimes$ like in Eq. 19. For the standard implementation of Bellman's GAP, the combine operator does nothing. Instead, the combine steps are integrated into the build steps that are executed in strict order after each other. The $foldr$ structure of the build steps is realized by nested loops, each looping over its respective candidate set, applying the evaluation algebra in the innermost loop to create a new candidate. New elements are individually added to the end of the candidate list. So for our example, the loops for X and Y are nested, and the function f is applied in the middle. Afterwards, a single loop for Z is created.
The arity of each function is directly linked to the number of nested loops, although an important distinction has to be made that is not strictly present in the implementation example. Algebra functions can take both terminal and non-terminal arguments. Terminal arguments are fed as moving boundaries or constant functions. Within Bellman's GAP, loops that are the result of a moving index boundary are always the outermost loops of the nesting, the loops created from subresults forming the inner loops.
The *nullary* function is applied only when the loops of one alternative production exclusively contains terminal production rules, hence exclusively consisting out of moving boundaries. The rationale behind this is as that all elements but the innermost loop are fixed values by Eq. 37. For the generalized implementation, Bellman's GAP assumes all inner non-terminal functions to have a properly defined choice function. Therefore, the sublists can be expected to follow the properties of the implementation, i.e. sorting or Pareto. Additional checks to ensure that the innermost candidate set has indeed a defined (ordering) choice function are conceivable, but ultimately most likely unnecessary with proper tree grammar design.
For the *append* functionality, different changes are needed depending on different implementations of $foldr$. For both, however, functions need to be added after the innermost loop, appending elements that were created within the last execution of the loop – and only those – to the overall solutions.

3.4.1 Step Mode

The step mode is the most literal implementation of $foldr$ without any additional logic. For each non-terminal, a global list of candidates is created and initialized as empty. For each alternative production, additional lists are created that are filled in the innermost nested loops respectively. After each execution of an innermost loop, the *append* function is called, combining the elements of the local list with those of the global list, saving the results again in the global list. The local list is emptied afterwards. Additional parameters, such as comparators, are passed as needed. This mode, however, is potentially problematic for performance. Taking the sorted implementations as an example, let us define M as the number of sorted sublists that are joined. The elements of the first merge are then sorted M times. The elements of the penultimate are sorted $M - 1$ times and so forth. Each element is therefore sorted on average $O(\frac{M(M+1)}{M}) = O(M)$ times. Additionally, in order to add a candidate, it potentially needs to be copied twice to insert it into the respective global and local lists, instead of adding it just once.

3.4.2 Block Mode

To eliminate these problems, a second mode was created with a different interpretation of $foldr$. Instead of directly calling *append* on each new sublist, all sublists are collected first. Like in the original implementation, a global list is created and filled by adding individual candidates to the end of the list, one at a time. However, a second list that is filled with integer indices of the sublist boundaries is kept. More specialized implementations for C++ exist, but in order to keep the possibility of generating code for other imperative languages as well, a universal structure to save list positions was chosen. The index list is filled by function calls after each innermost loop, adding the index of the current list end. This is not an immediate disadvantage, since also in step mode each element needs to be added once to a list and adding does not become much slower for longer lists. Only if little memory is available this becomes critical.
All sublists can then be joined using different algorithms (see Sections 6.1 and 7.2 for details). For example, for the sorted implementation, a list of sorted sublists is passed that can be interpreted as an intermediate step of an ongoing bottom up two way mergesort. If M is the number of sublists, using mergesort, each list element is then only sorted $O(\log(M))$ times. For Bellman's GAP, the switch between step and block mode is implemented as a command line argument.

3.5 GAP-L

Only few changes to the GAP-L language were needed in the course of this work, mainly because no special notation was included for k-ary products (see Section 3.1). Prior to this thesis, the Pareto product was included under the head symbol "ˆ". To the definitions in [6], the rule

```
product:
                product '^' product |
                ...
```

needs to be added as the only change.

3.6 GAP-C

Most of the changes to the framework for this thesis have been done to GAP-C. The compiler consists of three modules as shown in Fig. 1: A frontend, a middle-end, and a backend.

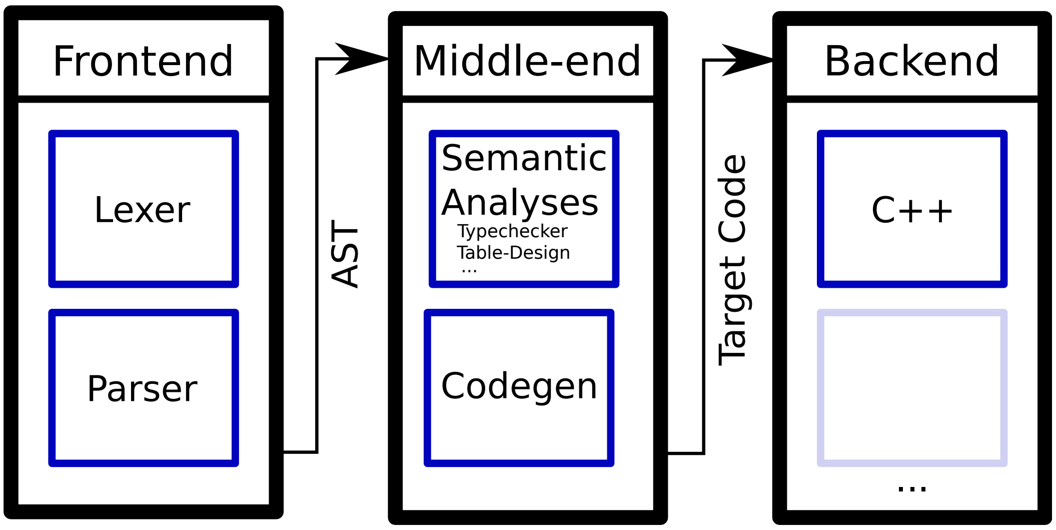


Figure 1: Architecture of GAP-C. Image taken from [6].

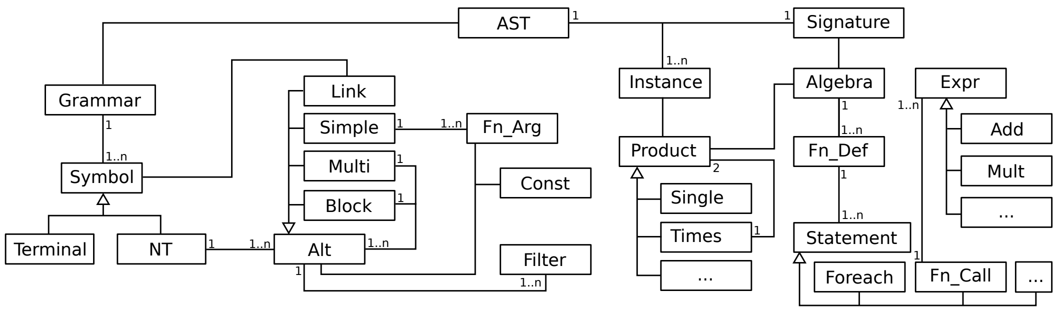


Figure 2: A diagram of the main abstract syntax tree (AST) classes. Image taken from [6].

The frontend interprets the GAP-L definitions and creates an abstract syntax tree (AST) out of them. For this, Flex[2] is used as lexer, Bison[3] serving as parser. For Pareto products, only the rule extension described in Section 3.5 needed to be added to the parser rules, and the head symbol was added in the lexer as a new symbol.
More complex changes were needed for the AST that serves as an object representation of the code to be generated and all contained functionality. The basic layout of the AST is depicted in Fig. 2. The core of the definition is the root *AST* object that combines all other objects and handles all function calls during the code generation process. The grammar is represented on the left side of the graph. The *symbol* class represents all terminals and non-terminals of the definition. Each non-terminal contains a list of *Alt* objects that describe the production rules. Subclasses of *Alt* handle different types of rules:

- *Alt::Link* is used to link back to terminals or non-terminals
- *Alt::Simple* is used to apply evaluation algebra functions
- *Alt::Block* is a grouping of alternatives
- *Alt::Multi* is a grouping for multi track ADP

The right side of the diagram shows the function and product definitions. *Fn_Def* represent final code functions. Ultimately, for each evaluation algebra function, for each product and for each non-terminal respective *Fn_Def* are created. A function object essentially contains a list of function parameters and a list of statements (*Statement*). Statements contain all functionality of the later program, defining everything from loops and conditionals to variable assignments and function calls. A class of expressions (*Expr*) exists that defines calls with return values. Both expressions and statements can reference external functions of GAP-M.
The middle-end modifies the AST and is responsible for the code generation process. While the front end only returns a nested structure of objects in the AST, the middle-end creates the code for each function, i.e. it creates *Fn_Def* objects and fills them with the appropriate lists of *Statement* objects.
Finally, the C++ code is generated out of the AST objects in the backend. This is handled by the *Cpp* object that contains functions to convert *Fn_Def*, *Statement* and *Expr* to C++ code.
In the following paragraphs, the most important changes to the AST and code generation steps are summarized by topic.

3.6.1 Pareto Products

For the support of Pareto products, a new object class was created in the *Product* class. For all implementations of Pareto front operators and for all implementations of ADP, the same class is used. Which kind of Pareto operator and ADP is generated is based on command line parameter options given by the user. For a summary, see Section A in the appendix.

The code generation of products is relatively complex, as multiple objects are involved. For the new Pareto products only the choice function generation had to be changed. Each product is assigned a corresponding algebra object in the binary product and algebra

[2]http://flex.sourceforge.net/ [18.09.2015]
[3]http://www.gnu.org/software/bison/ [18.09.2015]

trees. When the code generation is called on a product, it calls the code generation of its partnering algebra, passing itself as a parameter. The algebra function loops over all functions of the evaluation grammars of its left and right children, represented by *Fn_Def* objects, and generates a new *Fn_Def* for each, passing the children and the product as parameters. In the *Fn_Def* object, finally, the function content is generated depending on different parameters. For the standard implementation of ADP, only the Pareto operator kind is important and is encoded in the product objects as an enumeration of possible values and a boolean operator indicating if multi-dimensional Pareto support is needed (for products of 3 or more dimensions). Both parameters are passed up to the product from the AST object AST root.
For sorted implementations, another piece of information is needed. As already established, the element list can only be sorted at the root of the product tree, otherwise substructures would be modified without context. However, a *Product* object in GAP-C can only know its children, never its ancestors, therefore it cannot know if it is at the root or not. As an easy remedy, the root elements are marked by an object parameter before calling the code generation.

3.6.2 Comparators

Comparators are created whenever a sorting of the search space is needed or other templated functions are created that need such objects, regardless of which formulation of ADP is used. If a root product has been marked accordingly, it will automatically create a corresponding comparator. This functionality is executed in the *Fn_Def* objects while generating the choice functions. The comparison statements are generated as described in Sections 3.2 and 3.3.
To accommodate this, two new classes were introduced to GAP-C. First of all, comparators were assigned a new *Operator* type, meant to generate a general interface for passing functions as parameters. In the current version, *Operator* objects generate a C `struct` that implement an `operator()` function (after which the feature has been named), a construct that makes objects callable like functions and is faster than using function pointers. The general concept of operators, however, is kept broad enough to allow implementations in other programming languages. In C++, operators are defined as global, constant, static objects. The code segment of the operator is defined as a list of *Statement* objects. Additionally, constant values can be added to the operator. This is used, for example, to elegantly pass the available dimensions of a comparator to interfaced functions.
The second new class is a switch-case statement that was defined in the *Statement* class of GAP-C. Cases can be added as tuples containing the case value and a list of *Statement* objects as substatements. So far, this construct is only used within comparators, but it can be added universally throughout GAP-C.

3.6.3 Lists

Prior to implementing Pareto products, candidates were kept in a custom list implementation with limited standard list functionality. In its most basic characterization, the old implementation could be described as a low level deque in the sense that it was constructed as a double linked list of arrays of a fixed size. Elements could be added to the end and accessed in $O(1)$ time. Other than that, no further operations were supported. For various Pareto computations, however, full list support is needed, meaning elements need to be

inserted and removed from arbitrary positions.
Therefore, the original implementation was replaced by a standard deque implementation from the C++ STL library. This has a direct impact on all algorithms. Elements can be accessed in $O(1)$. Adding or removing elements at the end or the start of the list can also be done in $O(1)$. Inserting or removing elements from other position costs $O(N)$ or, more exactly, memory operations for every position to the nearest end of the list. Overall, deques perform well for appending and iterating over them[4], the most common operations performed in ADP. Since deques can be fragmented in RAM, they scale well for programs with a huge demand for memory, a common problem for ADP. Their performance was tested and confirmed superior in practice compared to the old implementation and other types of lists (data not shown in this work).

3.6.4 Floating Point Accuracy

Floating points can be a problematic part of ADP definitions. As a standard, GAP-C generates C++ *double*s for all floating point numbers, disregarding how much precision is actually needed. While this poses a potential performance problem, since unnecessarily large objects are handled and copied at runtime, it is not dangerous to do so. A much greater problem is the accuracy of floating point values.
Many real numbers cannot be precisely represented by floating points. Similarly, some floating point operations cannot precisely represent the true arithmetic operation. Since the definition of algebra functions is left fully to the user, Bellman's GAP has no guarantee that all operations are indeed safe, rendering full comparisons of floating values a rather dangerous operation. This can be, for example, problematic when searching for co-optimals in a set of candidates, when slight variations after the precision threshold separate them into different sets, while in reality they should be the same. Because of this, some existing Bellman's GAP applications, such as the covariance algebra for alignment folding, contain their own types of comparators that allow a certain margin of error in order for two values to be the same. Implementing such a change while remaining oblivious to the internal operations of Bellman's GAP is a dangerous task, however, as the user might either forget to overwrite some operators or overwrite too many. Until recently, this had caused faulty memory operations even in some in-house implementations.
Because of this, a new command line parameter was added to GAP-C that can automatically set the precision for all compare operations in the generated code. This is done by adding a header file that overwrites all needed operators if needed. The precision is set globally by a single constant flag in the generated code. This option should only be used if no prior handling of this issue is present, as competing overwrites of operators might cause undefined behaviour.

3.6.5 Generalized Implementation

The generalized implementation introduced the biggest change to GAP-C, as practically all components are affected. The ADP implementation and the mode of the implementations are passed by the user as command line arguments. For internal usage, all involved options are referenced by the namespace *ADP_Mode*. Names of comparators and involved functions are created in the *AST* object and passed down to all components to keep them consistent without redefinitions throughout the code.

[4]http://baptiste-wicht.com/posts/2012/12/cpp-benchmark-vector-list-deque.html [18.09.2015]

Various parts of the internal code generation are handled in different objects:

- creation of *Fn_Def* objects for each *nullary* function in *AST*
- content generation for *nullary* and *choice* in *Fn_Def*
- integration of *nullary*, *append* and *choice* calls in *Symbol* and *Alt*
- integration of *append* implementations via a library interface

Nullary functions are created as copies of choice functions. In GAP-C, empty choice function objects will be created alongside of parsing the GAP-L definitions file. Afterwards, options needed for the content generation will be passed to the objects by the *AST* root object. For the generalized implementation, an enumerator is set that describes the mode and the kind of ADP implementations. An additional flag is set to the root products defining which comparators/operators need to be generated.
If a code specialization was specified in the command line options, the *AST* object will recursively loop over all products and algebras and duplicate all choice functions. To differentiate between original choice functions and newly copied nullary functions, an enumeration flag is set to the *Fn_Def* objects. When code generation is called on each product, *Algebra* objects will automatically call the code generation of the newly created nullary functions as well. It is then up to the *Fn_Def* objects to generate code for each choice and nullary function.
This behaviour is mostly motivated by the Pareto eager implementation of ADP, where the nullary function effectively is the original choice function. Using this method, the original code generation can be used for the nullary function without any kind of modification. Similarly, for the sorted implementation, the original choice function can be kept without any modifications. As such, problematic re-implementations can be avoided. There is one setback to this approach, however, as one cannot simply exchange implementations by, for example, switching header definitions of needed functions, something that could also be done by the end user to create his own specialized implementation of ADP. So far, GAP-C has to be modified to allow new specializations. Should the need arise for this in the future, a simple switch could be introduced so that *Fn_Def* objects create calls to predefined template calls.
It should be noted that this problem cannot be fully avoided for the generation of comparators/operators, as they practically define the core operation for all interfaced function calls, and therefore are based on the definitions of the underlying algebras and products. New types of ADP implementations likely will require the implementation of new *Operator* types.

After duplicating choice functions to differentiate between *nullary* and *choice* generators, the names of the respective functions are passed to the *Symbol* objects from the *AST* object. *Symbol* objects will recursively pass down all names to its underlying *Alt* objects. Similarly, generation options for mode and comparator/operator names are passed down to each involved object of the production rules. Unfortunately, the integration of function calls is not a straightforward process. The standard implementation is generated in two steps. Initially, for each alternative of the production rule (*Alt* objects) an individual list variable is generated, but initialised to reference the same list as the global candidate list of the non-terminal. These intermediate variables are then removed in a second optimization step. To generate code for the step mode, this optimization process needs to be disabled, concurrently also removing the assignment of the local lists of each alternative production to reference the global list. At the same time, for block mode, the optimization can remain unchanged. However, while generating the function calls to *append* and *nullary*

functionality in the *Alt* objects, the global list is only indirectly known to each object. To further complicate things, for *Alt::Links* local lists cannot be removed. Even more so, some code that should be generated by *Alt* objects is indeed created by *Symbol*, most likely to avoid passing down information. For the implementation of the generalized interface this effectively means that the code is fragmented heavily across both object definitions.
nullary calls are strictly created in *Alt::Simple*. *choice* calls are strictly created in *Symbol* (unchanged to old versions of GAP-C). The generation of *append* depends on the defined mode. For step mode, shared between *Alt* and *Symbol*, the *append* function is called directly after each time a new sublist is created. If the *append* is within a nested loop structure, another call is added to empty the current local list, so no items are added twice. This is needed as the original local variables that are used are defined in scope right before the nested loop. For block mode, instead of directly calling *append*, a marker function is called. After the code for all alternative productions is created, the *append* function is called on all sublists at the same time.

Marker functions and *append* functions are implemented via template interfaces using the C++ Template functionality that allows generic typing that is deducted to the correct types at compile time. These kinds of interfaces are already used for functions defined in the GAP-M library and therefore do not pose a new restriction to the language system, although templates are not strictly supported for all languages that GAP-L could be used to compile into. Ultimately, this methodology simplifies the switch between different implementations to the point of simply including different header files with the corresponding definitions. For efficiency and consistency, the interface of *append* needs to be strictly defined, again depending on the mode.
For step mode, the interface is:

`append(`global list, new sublist [, operator [, operator [...]]] [, flag keep co-optimals]`)`

The result is returned not as a return value, as this would invoke a full copy of the returned list, but instead, values are directly populated into the global list that is passed by reference. Operators can be added as needed. For the sorted and Pareto eager implementations, the comparators are set as needed. The co-optimal flag is only needed for implementations that reduce the candidate space, such as the Pareto eager implementation. The rationale behind this is described in Section 5.2. Similarly, the block mode is called as:

`join_marked(`list with sublists, list of sublist markers [, operator [, operator [...]]]
[, flag keep co-optimals]`)`

All current realizations of these functions are kept in the GAP-M library.

3.6.6 Backtracing

For the implementations of backtracing, the same basic modifications that were made for forward mode computations apply, however, with two important exceptions.
Unfortunately, it is not possible to correctly deduct the candidate structure for generating comparators or Pareto products with more than two dimensions out of the artificially created backtracing product. Therefore, references to the original product structure have to be passed through the structures together with a marker for the backtracing root that is needed for sorted implementations.
The second exception is one done for optimization. In the most typical use case, when

computing Pareto fronts under backtracing, usually only the Pareto front is created during the forward phase, and candidate representations are added via a lexicographic product in the backward phase. If the non-terminals are tabulated, the backward phase does not need to repeat the expensive Pareto front computations, but can rather fish out the tabulated candidates out of the whole search space, which is done automatically by the lexicographic product. For the standard implementation, this rationale is already implemented. For the generalized implementation this effectively yields a return to the standard implementation, so that for tabulated non-terminals the specialized code generation is simply disabled while backtracing.

3.6.7 Algebra Characteristics

Special attention should be given to possible algebra characteristics in the context of Pareto and the generalized implementation. GAP-C performs a series of optimizations, depending on the kind of algebra that was given, or more specifically, on how the choice function is defined.

Definition 3.1 *(Algebra Roles)* An evaluation algebra A with choice function φ_A and $\varphi_A([]) = []$ is:

- enumerative, if $\varphi_A(X) = X$
- set-valued, if $\varphi_A(X) = set(X)$
- selective, if $\varphi_A(X) \subseteq X$
- synoptic, if $|\varphi_A(X)| = 1$ and $\nexists : \varphi_A(X) \notin X$

Within GAP-C, another level of differentiation is laid over this internally. Algebras can be classified as:

- *synoptic*: if φ_A is synoptic
- *kscoring*: if φ_A is enumerative or selective and $\exists X : |\varphi_A(X)| > 1$
- *scoring*: if φ_A is selective and $|\varphi_A(X)| = 1$
- *pretty*: if φ_A is enumerative
- *classified*: if φ_A is set-valued

Ideally, all underlying choice functions of a Pareto product are *scoring*, but the implementation can also handle *kscoring* and *classified* algebras. *pretty* is used for internal optimization only. *synoptic* algebras will fail for Pareto products! The category of product algebras depends both on the product as well as the underlying algebras. Unlike other products, the category of a Pareto combination is always *kscoring*. This might conflict with internal optimizations within GAP-C.
If the choice function of the root of the product tree is *scoring*, there is no need to keep full candidate lists. Instead, the lists in all recursions will be replaced by a single variable, and choice functions are applied at every step. This effectively constitutes a less general modification of the ADP implementation than is created by the new generalized

implementation. The operators $\otimes, \oplus$ and $\#$ are modified as

$$\begin{aligned} l \# \varphi &= l && (40)\\ l_1 \oplus l_2 &= \varphi([l_1 ++ l_2]) && (41)\\ \otimes(f, X, Y) &= foldr \quad \varphi ++ [] \ [\varphi([f(x,y) \mid x \leftarrow X]) \mid y \in Y] && (42)\\ \otimes(g, X) &= \varphi([g(x) \mid x \leftarrow X]) && (43)\\ h &= \varphi(h), && (44) \end{aligned}$$

φ indicating a scoring choice function. Upon looking at the definition, the reader may ask why this modification cannot be used as a basis of the generalized implementation. The main reason is that this implementation, while seemingly complicated, only needs to modify the list append function and to disable calls to the original choice function. No deep modification is needed. It should be fairly obvious, though, that the generalized implementation is not compatible with these changes, although it could be used to implement just this.

For *classified* choice functions, another level of modification is added, recasting existing list definitions to hashtables with filters [11, 6]. The full implementation is out of the scope of this thesis and will not be fully discussed. However, it should be noted that classification is, unfortunately, currently not supported for use with either the generalized implementation or any use of Pareto products, unless the Pareto product is not a part of the hash definitions. The reason for this is the hard setup of hashtables to only work with k-best choice functions. Allowing solution sets of arbitrary length for each class – as would be needed for Pareto products – is not within the scope of the current design. Ultimately, to allow Pareto products within this optimization, a full reconstruction of the hand-crafted current hashtable implementation in GAP-M would be needed. This was not attempted so far, as it would likely invalidate all current classifying implementations, such as those in the RNA shapes studio [17], as well as a number of handmade filters for these. Classification can still be achieved for Pareto products without this optimization when using the standard implementation.

For now, classified and choice function optimization are disabled for use with the generalized implementation.

3.7 GAP-M

No changes have been made to GAP-M other than to include files implementing marker and *append* functions for all possible variations of sorted and Pareto eager ADP.

4 Computing Pareto Fronts

Recently, Saule and Giegerich identified a series of different algorithms to compute two-dimensional Pareto fronts from unsorted or sorted candidate lists [13]. In their work, Pareto front operators have been implemented and evaluated in Bellman's GAP as filters over the search space, which could only be defined for specific products only. All their described algorithms have been re-implemented as fully functional algebra products prior to this work. As they closely relate to new implementations, and will be used in the final benchmarks, their definitions and features will again be presented. Saule and Giegerich described all algorithms in a functional fashion, but as Bellman's GAP currently operates in an imperative setting, instead, pseudo-code is given that more closely resembles the actual implementations.
Henceforth, we employ the following notation:

- ϵ is an empty list
- $l_{\text{pre}}{:}x{:}l_{\text{suf}}$ denotes an element in the middle of a list $l = l_{\text{pre}}{:}x{:}l_{\text{suf}}$ with l_{pre} containing all elements before x and l_{suf} containing all elements after x
- length(l) returns the number of elements in a list l
- $\leftarrow$ indicates a term rewriting
- $a < b$ means that b is better than a regarding the order of candidates (maximization)

Within the standard implementation of ADP, candidates are generated without any guarantees wrt. order. However, algorithms over sorted input lists can have favourable properties. We will categorize algorithms whether they take sorted or unsorted lists as input and whether the Pareto front they produce is sorted or unsorted.

4.1 Two-Dimensional Products

Algorithm 4.1 pf$_{nosort}$

```
INPUT: a) unsorted or b) sorted list $input$
OUTPUT: a) unsorted or b) sorted Pareto front in $answers$
  $answers \leftarrow \epsilon$
  for all $input_{\text{pre}}{:}(u,v){:}input_{\text{suf}}$ do
      $add \leftarrow$ true
      for all $answers_{\text{pre}}{:}(x,y){:}answers_{\text{suf}}$ do
          if $u \geq x$ and $v \geq y$ then
              $answers \leftarrow answers_{\text{pre}}{:}answers_{\text{suf}}$        ▷ Remove
          else if $u \leq x$ and $v \leq y$ then
              $add \leftarrow$ false
              break
      if $add$ then
          $answers \leftarrow answers{:}(u,v)$        ▷ Add
```

Starting at the definition of Pareto front operators in Eq. 15, the most intuitive implementation of a Pareto front operator is a an all-against-all comparison of all elements of a given input list. We call this method **pf**$_{nosort}$. The implementation is shown in Alg. 4.1. It is interesting to note that when using a sorted list as input, the resulting Pareto front will itself be ordered again. The worst case complexity of this implementation is $O(N^2)$, as is fairly obvious from the nested loops. One of the factors stems from the size of the Pareto front that, in the worst case, is of size N. In the average case the size of the Pareto front is $H(N) \approx \log(N)$, so the algorithm performs in $O(N \log(N))$ average runtime.

Algorithm 4.2 $\mathbf{pf}_{lex}$

```
INPUT: sorted list $input$
OUTPUT: sorted Pareto front in $answers$
  $answers \leftarrow \epsilon$
  for all $input_{\mathrm{pre}}{:}(u,v){:}input_{\mathrm{suf}}$ do
      if $answers == \epsilon$ then
          $answers \leftarrow (u,v)$                                  ▷ Add
          continue
      $answers_{\mathrm{pre}}{:}(x,y){:}\epsilon$
      if $v > y$ then
          $answers \leftarrow answers{:}(u,v)$                        ▷ Add
```

Algorithm 4.3 $\mathbf{pf}_{isort}$

```
INPUT: unsorted list $input$
OUTPUT: sorted Pareto front in $answers$
  $answers \leftarrow \epsilon$
  for all $input_{\mathrm{pre}}{:}(u,v){:}input_{\mathrm{suf}}$ do
      $add \leftarrow$ true
      $erase \leftarrow$ false
      for all $answers_{\mathrm{pre}}{:}(x,y){:}answers_{\mathrm{suf}}$ do
          if $erase ==$ false then
              if $(u == x$ and $y \geq v)$ or $(x > u$ and $y \geq v)$ then
                  $add \leftarrow$ false
                  break
              else if $u > x$ or $(x == u$ and $y < v)$ then
                  $add \leftarrow$ false
                  $erase \leftarrow$ true
                  $answers \leftarrow answers_{\mathrm{pre}}{:}(u,v){:}(x,y){:}answers_{\mathrm{suf}}$     ▷ Insert
          else
              if $y > v$ then
                  break
              else
                  $answers \leftarrow answers_{\mathrm{pre}}{:}answers_{\mathrm{suf}}$          ▷ Remove
      if $add$ then
          $answers \leftarrow answers{:}(u,v)$                        ▷ Add
```

If the input list is already sorted, a linear algorithm can be applied to compute the Pareto front. We call it $\mathbf{pf}_{lex}$ (Alg. 4.2). Saule and Giegerich showed that for two-dimensional Pareto fronts, an increasing sorting of the first dimension results in a decreasing sorting of the second [13]. To confirm this for oneself, the reason behind this is that in order not be dominated by another element, if one dimension increases, the other must decrease. As such, computing the front constitutes choosing all elements that are sorted according to the second dimension, or, in other words, to remove all elements that are not sorted in the second dimension. This can be easily done in a single pass over the input list, thus in $O(N)$. The sorting mechanism used to sort the input list will dominate the complexity when using this operator.

It is interesting to also consider a joined algorithm of $\mathbf{pf}_{nosort}$ and $\mathbf{pf}_{lex}$. From an unsorted list, a sorted front is generated by sorting them while computing the Pareto front (see Alg. 4.3). We call it $\mathbf{pf}_{isort}$. It is executed in two phases. First, the correct position is searched where the new candidate should be inserted. If the new candidate is dominated in this phase, it can be directly rejected. After the candidate has been inserted, on the second dimension now unsorted candidates are removed. Like with the unsorted implementation, the worst case complexity of this implementation is $O(N^2)$, but $O(N\log(N))$ average complexity can be expected.

4.2 Three- and More-Dimensional Products

Algorithm 4.4 pf$_{nosort}$ multi-dimensional

```
INPUT: a) unsorted or b) sorted list $input$
OUTPUT: a) unsorted or b) sorted Pareto front in $answers$
  $answers \leftarrow \epsilon$
  for all $input_{\text{pre}}{:}(u_1, \dots, u_k){:}input_{\text{suf}}$ do
      $add \leftarrow$ true
      for all $answers_{\text{pre}}{:}(x_1, \dots, x_k){:}answers_{\text{suf}}$ do
          if $u_1 \geq x_1$ and ... and $u_k \geq x_k$ then                ▷ Lazy
              $answers \leftarrow answers_{\text{pre}}{:}answers_{\text{suf}}$        ▷ Remove
          else if $u_1 \leq x_1$ and ... and $u_k \leq x_k$ then           ▷ Lazy
              $add \leftarrow$ false
              break
      if $add$ then
          $answers \leftarrow answers{:}(u_1, \dots, u_k)$                 ▷ Add
```

From the two-dimensional, unsorted implementation, an almost equal multi-dimensional case can be derived by adding all dimensions into the same conditionals as before (see Alg. 4.4). The basic mechanics stay the same as before, however, the expected case gets worse with each new dimension. As we recall, one of the factors of $O(N^2)$ stems from the size of the Pareto front. For $d > 2$ the front size is expected as $H^{(d)}(N)$, where $H^{(d)}$ is the generalized harmonic number and closely related to $log^{d-1}(N)$. Thus, the average case is close to $O(Nlog^{d-1}(N))$ in theory.

Algorithm 4.5 pf$_{lex}$ multi-dimensional

```
INPUT: sorted list $input$
OUTPUT: sorted Pareto front in $answers$
  $answers \leftarrow \epsilon$
  for all $input_{\text{pre}}{:}(u_1, \dots, u_k){:}input_{\text{suf}}$ do
      if $answers == \epsilon$ then
          $answers \leftarrow (u_1, \dots, u_k)$                           ▷ Add
          continue
      $add \leftarrow$ true
      for all $answers_{\text{pre}}{:}(x_1, \dots, x_k){:}answers_{\text{suf}}$ do
          if $u_2 \leq x_2$ and ... and $u_k \leq x_k$ then                ▷ Lazy
              $add \leftarrow$ false
              break
      if $add$ then
          $answers \leftarrow answers{:}(u_1, \dots, u_k)$                 ▷ Add
```

For **pf$_{lex}$** the situation is more complicated, as there are no guarantees for ordering anymore. Higher dimensions can break the ordering of lower ones, so that only the first dimension is totally ordered after sorting the candidate set. Let's take the list

$$[(10, 5, 5), (9, 7, 3), (8, 3, 7)]$$

as an example with maximization in all dimensions. It is clearly lexicographically sorted and a Pareto front, however, neither the second nor the third dimensions are ordered. If we now insert a new element $(7, 4, 4)$ that, according to the sorting, follows after all elements that are already in the front, it is no longer sufficient to only consider the last element of the list anymore. We see that $(8, 3, 7)$ does not dominate the new element because of the second dimension. Similarly, $(9, 7, 3)$ does not dominate it because of the third dimension. Only $(10, 5, 5)$ can reject $(7, 4, 4)$. This means for each element to insert, the full front needs to be tested, creating again an $O(N^2)$ algorithm (see Alg. 4.5). The expected case is lower, however, by the same rationale as with **pf$_{nosort}$**. In fact, both algorithms are

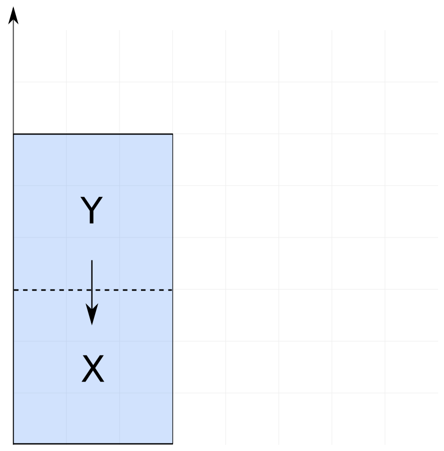


Figure 3: Sketch of the domination in $\mathbf{pf}_{yuk}$. Arrows indicate possible domination between lists. We have only one dimension of domination.

Algorithm 4.6 $\mathbf{pf}_{yuk}$

```
INPUT: unsorted or sorted list input, cut-off size c
OUTPUT: unsorted Pareto front
  if length(input) ≤ c then                                        ▷ Recursion End
      sort input
      return solve with $\mathbf{pf}_{lex}$
  else                                            ▷ Recursion (Divide and Conquer)
      split input in two sets $X$, $Y$ such that $Y$ is superior to $X$        ▷ Divide
      $X' \leftarrow \mathbf{pf}_{yuk}(X)$                                      ▷ Recursion
      $Y' \leftarrow \mathbf{pf}_{yuk}(Y)$                                      ▷ Recursion
      $X'' \leftarrow X'$ without first dimension
      $Y'' \leftarrow Y'$ without first dimension
      $X''' \leftarrow \mathbf{marry}(X'',Y'')$
      return $Y'$:$X'''$
```

now basically the same, with the only difference that in a sorted scenario, elements that already are in the Pareto front can never be removed again and thus, this conditional is missing. Also, the first dimension does not need to be compared anymore.

No attempt was made to transform the $\mathbf{pf}_{isort}$ algorithm to make it suitable for higher dimensions, as it no longer poses any advantage to $\mathbf{pf}_{nosort}$, for the same reason that increased the complexity of $\mathbf{pf}_{lex}$. Instead, a far more complex algorithm was implemented. Following the work of Bentley and Yukish [29, 27], the Pareto front of any set can be computed in guaranteed $O(Nlog^{d-1}(N))$, employing a divide and conquer strategy. Because the algorithm is very complicated and consists of multiple parts, only the basic idea will be given instead of full pseudo-code.
The basic operation of this algorithm is relatively simple. When defining a median element m of a list l, given an ordering induced by Pareto domination, all elements better than m (we call them Y) can dominate elements worse than m (X), but not the other way around. By comparing to m, we guarantee that every element of Y is better in at least the first dimension than every element of X. We say that Y is superior to X. If we compute the Pareto front individually for X and Y, we still need to remove elements from X that are dominated by elements in Y as is shown in Fig. 3. This is done by the **marry** step (Alg. 4.7). Since we already know that Y is better in the first dimension, we can drop it for this task. The algorithm of Yukish and Bentley applies this separation step recursively until all lists are smaller than a certain cut-off value. Lists below this value are first sorted and then the Pareto front is computed with $\mathbf{pf}_{lex}$; this is shown in Alg. 4.6.

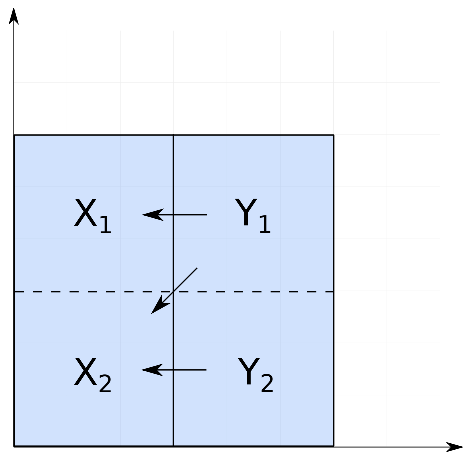


Figure 4: Sketch of the domination in the **marry** step of $\mathbf{pf}_{yuk}$. Arrows indicate possible domination between lists.

Algorithm 4.7 marry

```
INPUT: unsorted Pareto fronts $X$ and $Y$, $Y$ superior to $X$, cut-off size $c$
OUTPUT: $X$ without elements dominated from $Y$
  if 2 dimensions then                                          ▷ Recursion End 2
    return solve with $\mathbf{marry}_{2D}$
  else if length($X$) $\leq c$ or length($Y$) $\leq c$ then      ▷ Recursion End 1
    return solve with $\mathbf{marry}_{brute}$
  else
    choose a cut plane to divide $X$ to $X_1, X_2$ and $Y$ to $Y_1, Y_2$
        st. $X_1$ superior $X_2$, $Y_1$ superior $Y_2$, $X_1$ superior $Y_2$      ▷ Divide
    $X_1' \leftarrow$ marry($X_1, Y_1$)                              ▷ Recursion 1
    $Y_2' \leftarrow$ marry($X_2, Y_2$)                              ▷ Recursion 1
    $X_2'' \leftarrow X_2'$ without first dimension
    $Y_1'' \leftarrow Y_1$ without first dimension
    $X_2''' \leftarrow$ marry($X_2'', Y_1''$)                         ▷ Recursion 2
    return $X_1'$:$X_2'''$
```

The **marry** algorithm is more complicated, as we no longer have to deal with separation along one dimension, but rather along two dimensions. From the last step, we gain the separation of the original list into Y and X. We now separate them again by a new median into Y_1, Y_2 and X_1, X_2 respectively. It is important to use the same median for both lists, as only then we can guarantee that X_1 is superior to X_2, Y_1 is superior to Y_2 and X_1 superior to Y_2. Fig. 4 demonstrates the layout of this scenario, with arrows indicating which lists can dominate elements in each other list. X and Y are already Pareto fronts, so they cannot contain dominating elements within themselves. It is important to note that subsets of Pareto fronts are again a Pareto front, so this property always holds when dividing lists to create smaller subproblems. The **marry** problem is divided along two features.

First of all, the just described separation step creates sub-problems of smaller list lengths. We now need to marry Y_1 with X_1 and Y_2 with X_2. This recursion ends when the lists are below a certain cut-off, and a brute force marry algorithm that works similar to $\mathbf{pf}_{nosort}$ is called, just that only X is compared to Y. The full code is given in the appendix in Alg. B.2.

Secondly, the elements of the list X_2 can also be dominated by elements in Y_1. Since X_2 and Y_1 are separated using the same median, we can again drop the first dimension before marrying them. This recursion ends once we have reached a two-dimensional problem that can be solved by employing a linear time algorithm that works by the same principles as

$\mathbf{pf}_{lex}$. However, it operates on two Pareto fronts. This means elements in X can only be dominated by elements in Y, so while moving through X, the last element of Y is always kept as a reference. The pseudo-code is shown in the appendix as Alg. B.1.
The complexity of $O(Nlog^{d-1}(N))$ can be intuitively explained by the recursive decomposition of the problem along a median similar to Quicksort that is executed until only 2 dimensions remain, adding a factor of $log(N)$ per dimension.

Only $\mathbf{pf}_{yuk}$ was implemented fully as part of this work. $\mathbf{pf}_{nosort}$ and $\mathbf{pf}_{lex}$ existed as test cases for the rudimentary implementation of multi-dimensional Pareto definitions done prior to this work, and were only optimized.

4.3 Optimization

Code optimization is a relatively straightforward process for all implementations except $\mathbf{pf}_{yuk}$. The concepts creating the most slowdown are defined by the basic framework that cannot be easily modified, leaving little room for improvement. The most problematic feature of choice functions is that they must create a new list of candidates instead of being able to modify the input or move elements of the old lists instead of copying them. Each selected candidate is, therefore, duplicated in memory potentially multiple times along the choice function tree. There is, however, no way to amend this problem. As we already established before, lower choice functions only have access to references to substructures of the overall candidate list. Calling a move instead of a copy on such a substructure would therefore destroy the shared list, or at least constitute an unsafe memory operation. The only way to fix this would be to copy all substructures before passing them as arguments, which ultimately would cause a lot more copy operations. The second reason is embedded within the definitions of algebra products, or more specifically, the products of choice functions. Reducing candidates of the search space prematurely will, in fact, break certain product definitions, like, for example, lexicographic products. We will see later how this problem is handled for the Pareto eager implementation where this problem actually arises (Section 5.2).
For the multi-dimensional versions of the algorithms, all comparisons are generated in a way that emulates the lazy evaluation behaviour of binary expressions. If an action can be rejected based on a lower dimension, the comparisons of higher dimensions will not be executed. While this does not change the asymptotic runtime, it drastically reduces the constant factor.
Although we can not easily change this behaviour, a few notes should be given about list operations as well. In the pseudo-code, operations have been marked as Add, Remove, or Insert. The theoretical complexities given in the last section are based on the fact that all three operations can be executed in $O(1)$. By the choice of deques (see Section 3.6.3) this is not the case, causing Remove and Insert to behave in $O(n)$, affecting the runtimes of $\mathbf{pf}_{nosort}$ and $\mathbf{pf}_{isort}$. Nevertheless, the use of deques was confirmed to be the best solution prior to this work, even when including these operations.

For $\mathbf{pf}_{yuk}$, the situation is a lot more complicated as various competing implementation strategies exists. A fair balance between maintainability and optimization has to be kept. As the algorithm is very complicated, and fine structure changes are needed for optimization, it is not created fully by the code generation process. Rather, it is implemented in GAP-M, and only an interface call is generated for the choice function. Thus, one of the bottlenecks of the implementation is the use of comparator functions. Lazy behaviour is achieved by loops over all needed dimensions, comparing each dimension individually and

breaking the loop as early as possible. The compiler might optimize this code using loop unrolling, but this creates (nested) branches and the problem of false branch predictions remains. Loops will generally mispredict at least once on exit if not unrolled. Ultimately, this problem cannot easily be resolved, especially not while implementing code for arbitrary dimensions. Duplicating and unrolling loops for all needed dimensional ranges via pre-compiler macros seems within reach, but the benefits would likely not justify the effort, as we would just create conditional jumps in other parts of the code instead. A full inlining would necessarily require to unroll the full dimensional recursion, drastically increasing the code size, which again would have negative effects on the overall speed. Accepting the overhead of both loops and comparators seems to be reasonable in this context.
The two most pressing questions of the implementations are how to store candidate lists and how to choose the optimal median for splitting lists. We will later also address the optimal choice of the cut-off values for different applications. Currently, two versions of $\mathbf{pf}_{yuk}$ are implemented. In the reference based version, all intermediate lists consist of pointers to elements of the candidate list. This means that while the memory footprint of storage is low and we avoid copying potentially large elements, we first need to transform the input to a pointer list, and in the end, transform the pointers back to a list of real objects. Additionally, having to dereference a pointer at every step might have a negative influence on core operations. Contrasting this, an implementation was realized where full objects are kept in intermediate lists. Obviously, this creates costly copy operations on potentially large objects, although some of the internal lists can make use of more effective move operations when compiled with C++ 11 support.
Experimentally, a version of $\mathbf{pf}_{yuk}$ was created that used the median of medians [1] approach to find a well suited median, then arranging the list regions by memory swaps similar to Quicksort. We can partition the full input list until the cut-off is reached in $O(N \log(N))$. The practical implementation, however, turned out to be painfully slow due to high constant factors induced by this method. For the same reason, usually the median-of-three is recommended for Quicksort [30], using the median of the first, the middle, and the last element. This can be done in $O(1)$ instead of $O(N)$. However, such a median has the potential to create strongly unbalanced splits as we have no guarantee for optimality. As a compromise, we do the following: the lists are first fully sorted by a Quicksort with median-of-three (and some optimizations proposed by Sedgewick [30]). On a sorted list, all splits can be found perfectly balanced in $O(\log(N))$ for any cut-off. This implementation is further motivated by the fact that underlying algorithms like $\mathbf{pf}_{lex}$ and $\mathbf{marry}_{2D}$ also require sorted inputs, so sorting elements below the cut-off does not go to waste in many cases. To avoid the comparator overhead on the first dimension, lists are already passed sorted to $\mathbf{pf}_{yuk}$.
It is critical to avoid copying or moving list elements whenever possible. Most importantly, this can be avoided for splitting up the candidates along the first dimension, as we can keep the splits as index structures with access in $O(1)$, in this case, C++ *Iterators*. The candidates are first modified by $\mathbf{pf}_{lex}$ after the list regions are smaller than the cut-off. Only then new additional lists are constructed, as we no longer can work on the immutable input list. For the reference implementation, here, the reference lists are created and for the copy version elements are just copied to new Pareto lists. Similarly, Y cannot be modified during **marry**, and splits only need to be kept as indices.
The descriptions of both $\mathbf{pf}_{yuk}$ and **marry** show a high level of recursion over different categories, most of which can be avoided by using iterative implementations. This not only serves to avoid the overhead generated by frequent function calls, but also improves the locality of the program. We can, for example, first create all splits and only after that

apply $\mathbf{pf}_{lex}$ and **marry** them, improving the usage of the code-cache.

Aside of these changed, there are general-purpose optimization heuristics that can be employed for $\mathbf{pf}_{yuk}$ as well as any other function, most notable:

1. Whenever possible, objects are passed as references to avoid copying them.
2. Similarly, lists are never returned in a return statement, but rather, references are passed as arguments.
3. Whenever possible, move is used instead of copy for list elements.

The second optimization is kept throughout all code generated by GAP-C, although less visible. Copy operators will indeed be called on lists repeatedly, but as *List_Ref* is just a reference wrapper, only the reference will be copied, not the list itself.

5 Algebra Products and their Influence on Candidate Lists

Up to this point in this work, some implicit assumptions have been made that can no longer be upheld. So far, we have neglected to take the interactions between other algebra products and Pareto products into account. In order for the Pareto operators to work properly, other choice products in the product tree must not interfere with their definitions. In the standard implementation of ADP, Pareto products are surprisingly hard to break. None of the currently implemented products in the Bellman's GAP system can directly fault computations, aside from possible implementation issues with GAP-C. At present, only custom user filters can easily interfere with the process.
The Pareto eager implementation of ADP, as well the lexicographically sorted implementation are a lot more sensitive in this regard. Various products can interact negatively with their definitions. It is best to explain this problem in the terms of an expositional example of one such case. A likely product a user may attempt to create would be one such that $A *_{\text{lex}} (B *_{\text{Par}} C)$, where A is an algebra partitioning the search space into classes, and B, C are either minimizing or maximizing. A practical case of such a definition would be, for example, to compute a Pareto front for each shape class of an RNA folding. We need to pay special attention to the choice function definition of the lexicographic product $*_{\text{lex}}$ (Eq. 14) on what this means exactly. Both A and the Pareto product $(B *_{\text{Par}} C)$ are kscoring (or classifying) in nature. The left part of the choice function definition, Eq. 14a, creates a set of all possible values over the left choice function, here all possible classes. The right side then creates subsets for each class, applying the right hand side choice function to each set separately (Eq. 14b). As a result, we don't compute just one global Pareto front, but rather one individual front per class.

For now, let's remain in the standard implementation and see why this is unproblematic in this case. It is easy to see the unsorted implementations ($\mathbf{pf}_{nosort}$, $\mathbf{pf}_{isort}$, $\mathbf{pf}_{yuk}$) of the Pareto front operator can handle this situation well as they can cope with every input. Instead, let's say we have the following list of candidates

$$[(a,(6,1)),(b,(5,2)),(a,(4,4)),(c,(3,5)),(a,(7,1)),(b,(4,3))],$$

with $\{a,b,c\} \subseteq A$, and B, C maximizing over elements in $\mathbb{N}$.
If we want to use $\mathbf{pf}_{lex}$ in the standard implementation, we must first sort the list (Eq. 24). Please remember that the sorting is done globally on the full candidate lists because of restraints in the datastructures used to represent it. Modifications on this restriction might be possible with additional intelligence during code generation in some instances, but we should not be concerned with such details, as they only potentially influence efficiency of the code, but not general functionality. Globally sorted, our example list turns to

$$[(a,(7,1)),(a,(6,1)),(b,(5,2)),(a,(4,4)),(b,(4,3)),(c,(3,5))].$$

A global sorting also automatically sorts all possible subsets of elements in the list, therefore, when extracting candidates for each class, they are sorted as well, allowing $\mathbf{pf}_{lex}$ to compute the correct Pareto fronts.

$$\begin{aligned}
\mathbf{pf}_{lex}([(a,(7,1)),(a,(6,1)),(a,(4,4))]) &= \quad [(a,(7,1)),(a,(4,4))]\\
\mathbf{pf}_{lex}([(b,(5,2)),(b,(4,3))]) &= \quad [(b,(5,2)),(b,(4,3))]\\
\mathbf{pf}_{lex}([(c,(3,5))]) &= \quad [(c,(3,5))]
\end{aligned}$$

The results are added to the result list in order of the class and the Pareto operator output

$$[(a,(7,1)),(a,(4,4)),(b,(5,2)),(b,(4,3)),c,(3,5))]\,.$$

Before applying $\mathbf{pf}_{lex}$ the next time, the elements will be sorted again. This rigid switch between sorting and "unsorting" of the candidate list might be ineffective, but is very robust. The use of unsorted Pareto operators is still recommended.

5.1 Properties of Candidate Lists

It is now time to see how this affects the Pareto eager and sorted implementation by extending the above example. Let's say we want to merge the lists l_1 and l_2, consisting of the same candidates as before, and they are the only two lists created by the non-terminal before applying the choice function.

$$\begin{aligned} l_1 &= & [(a,(6,1)),(b,(5,2)),(a,(4,4)),(c,(3,5))] \\ l_2 &= & [(a,(7,1)),(b,(4,3))] \end{aligned}$$

Both lists are sorted and Pareto fronts, assuming the initial conditions for the specialised implementations are met, although this would constitute mere coincidence in reality.

Starting out with the sorted implementation, both lists can be merged without any problems, and we get the same sorted list as before.

$$[(a,(7,1)),(a,(6,1)),(b,(5,2)),(a,(4,4)),(b,(4,3)),(c,(3,5))]$$

The problem only arises when applying the choice function that, as already established, separates and orders candidates by classes first and changes the candidate list. The result is the same, as before, clearly unordered result list

$$[(a,(7,1)),(a,(4,4)),(b,(5,2)),(b,(4,3)),c,(3,5))]\,.$$

This violates the condition of intermediate lists having to be sorted as required by the build $\otimes$ definitions.

Pareto merging the two sets would yield a combined Pareto front

$$[(a,(7,1)),(b,(5,2)),(a,(4,4)),(c,(3,5))]\,.$$

While keeping the Pareto condition intact, it is clear that this is not what we wanted to compute.

One could argue that this is not really a problem, as within each class the properties of the respective implementations are in fact conserved. Build $\otimes$ functions need to be created in such a way that they can recognize their context.
Staying with the example product for the sorted implementation, we can keep the original logic, but it needs to be extended in such a way that *merge* does not merge sorted lists, but lists of sorted lists. Keeping the original definition of *merge*, one can change the definitions such that we gain another level of *foldr* to merge the classes within each sublist.

$$\begin{aligned} \otimes(f,X,Y) &= foldr \quad merge \quad [] \\ &\qquad [foldr \quad merge \quad []\ [f(x,y) \mid x \leftarrow X] \mid y \in Y] && (45) \\ \otimes(g,X) &= foldr \quad merge \quad []\ [g(x) \mid x \leftarrow X] && (46) \end{aligned}$$

Of course, one needs to be careful that the definition of select # is adjusted as well to reflect this change. Alternatively, the application of the choice function could be moved before the execution of each *merge* so that each class can be handled individually, posing the question on how the actual choice function definition should be modified so no choice function is executed twice.
For the Pareto eager implementation, we need to effectively reprocess sublists to call the merge on every class individually, or in other words, we need to apply the $*_{\text{lex}}$ operator on the sublists before executing the Pareto merge. In the broader context, this makes sense as we move the application of the choice function to another position in the definition, so we could move all other choice functions in the product tree at the same time.

While sounding well in theory, these changes are particularly hard to implement as they mostly cannot draw on parts of the existing framework. Just taking the sorted implementation, the easiest solution to resolve this issue would be to sort the now unsorted sublists before merging them. This, however, would practically invalidate the whole concept. In a more complex solution, the introduction of the second layer of *foldr merge* could be achieved by modifying the original $*_{\text{lex}}$ to include the merges or to alternatively pass a list of split indices to the calling non-terminal to pass it down to the next build step. While both could be realized within a conceivable amount of effort, they pose a very specific fix for one specific implementation of ADP with one specific product.
Moving the lexicographic product $*_{\text{lex}}$ for execution before the *merge* or Pareto merge respectively, as described above, would pose a more general solution. The original generators for the choice function tree could not be reused, however, as the operators now execute different tasks. The standard definition of $*_{\text{lex}}$ only needs to deal with one solution list. The new definition would need to be able to keep track of two or more solution lists – one for each class – that need to be processed in parallel. At the same time, the product must ensure class sets are correctly joined. Even more complicated, the question remains when to execute potential other products of the tree.

Most sensibly, this case of classified products should be realized by a full reconstruction of hashtables, combined with an optimized and fitted in implementation of the generalized interface, bridging the gap to Section 3.6.7. Such a construct could even be used to treat each class as an individual solution list, avoiding conflicts such as above to a large degree. Leaving aside the possible implementation details of our example case again, it is time to finally return to the general problem raised in this section.

The example was chosen to highlight the high complexity of interaction between product definitions. Even for such a seemingly simple product, multiple conflicting implementation strategies arise, and the question of how a generalized handling strategy within ADP should be implemented remains problematic. It is time to remind us that other such products can be defined, the most obvious being the combination of multiple Pareto fronts. Before investing much work in potentially unneeded features, it is wise to concentrate implementations to the most common use cases of Pareto optimization, which is computing a Pareto front combined with printing out the candidates in the front. More formally, we want to analyse products of the form

$$*_{\text{Par}}\{A_1, \dots, A_k\} *_{\text{lex}} B_1 \dots *_{\text{lex}} B_l, \tag{47}$$

joining a k-ary Pareto product with an arbitrary number of lexicographic products. This product conserves the properties of sorted and Pareto eager ADP. For the sorted implementation, the choice functions are generated unchanged to the standard implementation, so all other products will still be accepted, although the burden of correctness lies with the user. It is not as easy for the Pareto eager implementation, as we will see in the next section.

5.2 Candidate Reduction and Products

For the generalized implementation, the question arises of how and when the choice functions are applied to the candidate list. We could already see in the last section that there is likely no universal solution to this. Instead, in this section, we want to explore the difficulties that arise with the Pareto eager implementation. From the definitions in Section 2.3.2, we have to examine the select $\#$ (Eq. 30) and the build $\otimes$ and combine $\oplus$ steps (Eqs. 31 and 32) in particular. The basic idea of the Pareto eager implementation is to move the application of the Pareto front operator from the select $\#$ to the build $\otimes$ step. The question remains of what happens to all other choice functions in the tree.
The first intuition is that only the Pareto operator is moved, while all other choice functions remaining in the select step. This would mean we can generate the choice function tree as before, just changing the Pareto operator to the identity function in the chain. To see why this will not always work, we need to remind ourselves that the right hand side algebra of a lexicographic product can either just return one candidate per front element or possibly more (co-optimals). Let's regard a possible list of candidates for the product $(A *_{\text{Par}} B) *_{\text{lex}} C$, where C is an algebra printing candidate representations and A, B are maximizing. Let's assume we have a candidate list as follows produced as the only elements before the choice function:

$$[((2,1),a),((1,2),b),((1,2),c),((2,0),d)]\,.$$

In the standard implementation, the full list is passed to the choice function of the lexicographic product. The list will be broken down to only the left element of each tuple and passed to the Pareto operator, which will return the list

$$[(2,1),(1,2)]\,.$$

The lexicographic product then maps back to the full candidate list and returns

$$[((2,1),a),((1,2),b),((1,2),c)]\,.$$

By the way of this example, we can see that co-optimals are retained in the lexicographic product, after seemingly being eliminated by the Pareto operator. This differentiation is important when repeating this example for the Pareto eager implementation.
We join two Pareto fronts consisting of the same elements as before.

$$\begin{aligned} l_1 &= [((2,1),a),((1,2),b)] \\ l_2 &= [((1,2),c),((2,0),d)] \end{aligned}$$

Following the definition of Pareto of Eq. 15, the merge will eliminate all co-optimals. So we have two possible results, but we will just stick to one.

$$[((2,1),a),((1,2),b)]$$

We see that the candidate $((1,2),c)$ has been removed, although it should be part of the final result. Applying the choice function of the lexicographic product cannot return the element, as it no longer exists in the list to process.

Two general solutions exist for this problem. To emulate the behaviour of the standard implementation, we can keep two separate candidate lists, one computing the Pareto list on the run, the other collecting all candidates as before. The choice function then takes both lists as arguments. There is nothing inherently wrong with this technique. However, the idea of Pareto eager ADP relies on reducing the size of candidate lists as early as possible. In the worst case doubling the effort of keeping candidate lists seems to be counterproductive.
As an alternative, all choice functions can be moved to $\otimes$ and $\oplus$ to be applied at the same time as the Pareto front operator. This method, however, is potentially dangerous for performance. The lexicographic operator performs in $O(N^2)$, higher than the cost of merging Pareto fronts in two dimensions. So, instead of paying this cost once at the end of each non-terminal, we move it to the innermost loop of each alternative production. In other words, we create costs with a higher complexity for each merge.

For the lexicographic product, a third alternative exists, where we combine the lexicographic product with the Pareto product to one single function. This can be done at no costs at all. We only need to redefine the Pareto operator so that it retains co-optimals.

Definition 5.1 *(Co-optimal retaining Pareto front operator)* We define D and E with relations $>_D, >_E$ as before. A Pareto front operator that allows co-optimal front elements is then defined as

$$\begin{aligned}\mathbf{pfc}_{>_D,>_E}(X) \quad = \quad & \{(d,e) \in X | \nexists\, (d',e') \in X \setminus \{d,e\} \text{ with} \\ & d \leq_D d', e <_E e' \text{ or } d <_D d', e \leq_E e'\}. \end{aligned} \tag{48}$$

Like before, we will drop all information about arity or ordering for the rest of this work and use a more general **pfc** where clear. This change can be easily included into all Pareto front operators defined in the last section as we will see in Section 7.

Theorem 5.1 *(**pfc** is a combination of **pf** and $*_{lex}$)* Given algebras A_1, A_2 with minimizing or maximizing choice functions and B with $\varphi_B(X) = X$ as choice function, then the choice function of the product $(A_1 *_{\text{Par}} A_2) *_{\text{lex}} B$ can be computed as

$$\begin{aligned}&\varphi_{(A_1 *_{\text{Par}} A_2) *_{\text{lex}} B}[((a_{1,1}, a_{2,1}), b_1), \dots, ((a_{1,m}, a_{2,m}), b_m)] = \\ &\qquad\qquad \mathbf{pfc}[((a_{1,1}, a_{2,1}), b_1), \dots, ((a_{1,m}, a_{2,m}), b_m)].\end{aligned} \tag{49}$$

Proof. We only give a sketch of the proof. We define $a_i := (a_{1,i}, a_{2,i})$. The choice function then results to

$$\begin{aligned}\varphi_{(A_1 *_{\text{Par}} A_2) *_{\text{lex}} B}&[((a_{1,1}, a_{2,1}), b_1), \dots, ((a_{1,m}, a_{2,m}), b_m)] \\ &= [(l, r) \mid \\ & l \in \mathbf{pf}[a_1, \dots, a_m], \\ r \leftarrow [r' | (l', r') &\leftarrow [(a_1, b_1), \dots, (a_m, b_m)], l' = l]]\end{aligned}$$

We need to show that the correct values for r are chosen in the last part of the equation.
"$\Rightarrow$":
Per definition of **pf** and **pfc** we know

$$\mathbf{pfc} \setminus \mathbf{pf} = [(a_1^x, b_1^x), \dots, (a_n^x, b_n^x)] \text{ st. } \forall (a_i^x, b_i^x) \exists (a^y, b^y) \in \mathbf{pf} \text{ with } a^y = a_i^x$$

So for each (a_i^x, b_i^x) an $a^y = l$ exists such that b_i^x is chosen.

"$\Leftarrow$":
If $l \in$ **pf** then also $l \in$ **pfc**. Per definition **pfc** keeps all elements
$(l', r') \leftarrow [(a_1, b_1), \ldots, (a_m, b_m)]$ where $l = l'$ as equal elements cannot dominate. $\square$

Using Theorem 3.1, the theorem can be extended to multi-dimensional Pareto definitions.

The actual implementation is twofold. So far, we have omitted backtracing from the discussion. Ideally, with backtracing, our example product will be split up into a forward phase computing $(A *_{\text{Par}} B)$ and a backtracing phase computing $(A *_{\text{Par}} B) *_{\text{lex}} C$. In Tab. 1 all possible combinations of options are given. With backtracing, the forward computation does not incorporate any other products, so the standard Pareto front operator definition can be used. In the backward phase, we then need to differentiate between tabulated and untabulated non-terminals. For untabulated cases, we need to recompute the Pareto front, and will therefore rely on the co-optimal retaining Pareto front operator. For the tabulated case, we already have the front, so we should not recompute it, and can instead directly apply the choice function with the tabulated values. This leaves somewhat of a pragmatic after-taste, as for sorted two-dimensional Pareto products, we can compute the Pareto front and implicitly the lexicographic product without changing the complexity compared to simply listing all candidates, while the lexicographic product takes $O(N^2)$. List sizes are always kept as small as possible using **pfc**. However, Pareto products bring in a large factor for each merge, compared to a single application of the choice function. There is also further merit for using the choice function than potential speed. In a way, this realises the first general solution of above – where we keep the Pareto front, and a candidate list in parallel – with the additional benefit of backtracing, thus without duplicated candidate lists in comparison to the standard implementation. Should the need arise to use Pareto eager ADP with a combination other than the lexicographic product, this can be done by tabulating all non-terminals.

	co-opt	**no co-opt**	**co-opt, backtr.**	**no co-opt, backtr.**
forward	**pfc**	**pf**	**pf**	**pf**
backward tabulated	-	-	$*_{\text{lex}}$	$*_{\text{lex}}$
backward untabulated	-	-	**pfc**	**pf**

Table 1: Overview of options and generated operators for Pareto eager ADP. If backtracing (backtr.) is not defined, co-optimal (co-opt) candidates are handled in the forward computation. With backtracing, co-optimals only have to be considered in the backward phase. If the results of the forward phase were tabulated, the Pareto front is not recomputed, but the normal choice function is called using the tabulated front.

6 Lexicographically Sorted ADP

Now that we have defined all theoretical, and some practical, components needed for the implementation of lexicographically sorted ADP, it is time to combine them and fill in the missing parts.
In Section 2.3, we have seen two different strategies to create sorted lists. In Section 4, we have seen the definition of $\mathbf{pf}_{lex}$ that expects and produces sorted lists and can, therefore, find use as a choice function in Eq. 24 or 25.
What is left for us to analyse are different implementations of sorting algorithms, corresponding either to *sort* in Eq. 24, or *merge* in Eqs. 26, and 27.
It is interesting to note that also $\mathbf{pf}_{nosort}$ can be used with all sorted implementations, as it keeps the sorting intact, but there is no practical benefit for doing this. While $\mathbf{pf}_{yuk}$ would benefit greatly from an already sorted input list, it is virtually incompatible with the the sorted implementation of ADP, as it produces unsorted Pareto fronts.

Pareto and sorting operations on lists of intermediate results in a dynamic programming setting are executed for each entry of the dynamic programming matrix (or matrices) and in the innermost loops of the algorithm. Careful attention needs to be placed not only on the asymptotics of each algorithm in order to create good code, but also other properties such as memory usage and constant factors in the implementation.

6.1 Sorting Algorithms

Sorting algorithms are traditionally classified along different criteria. Next to runtime complexity, the memory usage of the algorithms is very important in the context of dynamic programming. An algorithm can either operate *in place*, taking $O(log(N))$ or less in additional memory, while algorithms that are not in place need worst case $O(N)$ memory or more. Here, we also introduce a third category of algorithms that will not move around elements that are already sorted, but technically are not in place, contrasting with implementations that will copy or move every element once regardless of position. Another interesting aspect is the adaptiveness, as even non-adaptive algorithms can show adaptive behaviour. Most of the following algorithms are adaptive in the sense that they work on already known sorted sublists. Other factors, such as stability, are not important for this work and will therefore not be discussed.

S1 *Quicksort:* If nothing is known about the content of the list, Quicksort is widely recognized to be one of the fastest algorithms for comparison based sorting. The C++ STL library (the definitions of GNU were used) contains an implementation of Quicksort with a cut-off and Insertion sort for smaller problems as proposed by Sedgewick [30]. Median-of-three is used to find cut elements. It can be considered to run in place with an average runtime complexity of $O(N\log(N))$, while the worst case is $O(N^2)$. Empirically, Quicksort can be shown to be adaptive to pre-sorted sublists [31], shortening computation times with increasing numbers of elements that are not inverted, hence are already in the right order relative to each other. For ease of use, this property will be addressed as *sortedness* for the rest of this thesis.

It is important to note that while we might not know where the sorted sublists are in the standard implementations of ADP, when using a Pareto front operator that creates sorted lists, the candidate list will still contain pre-sorted areas. As such, Quicksort is a very good candidate for *sort* in Eq. 24 and sets a fast reference for all other implementations. It is also suitable for the use in Eq. 29.

In the lexicographically sorted implementations of ADP, already sorted sublists are known within the overall lists. Depending on the mode of Eq. 27 (see Sections 3.4.1 and 3.4.2) sorted lists are handled differently. Ignoring all other properties of both modes for a moment, the difference for the *merge* algorithms is how many sublists can be merged at once. All implementations that support the block mode automatically are suitable for the step mode, but not vice versa. Effectively, the same implementations could be used for both modes, potentially removing the overhead for the step mode in some cases.
In this work, however, the step mode will not be supported for the use with sorted ADP. In Section 3.4.1, the inferior complexity for merging more than two lists compared to the block mode was already addressed. This behaviour can only be justified if the successive application of a function is advantageous to first collecting all elements. Arguably, this can be the case when the *append* function reduces the number of elements during execution, thus preventing lists from growing large early on. Sorting, however, is only a reordering of candidates that does not modify any other properties of the candidate lists. Neither space nor memory operations can be saved. The only conceivable advantage of the step mode in this case is that no index structures need to be kept to keep track of sorted sublists. When the number of sorted sublists is small, index lists are easy to keep and cost near to no additional effort. It is only when the number of sublists comes close to the number of elements in the candidate list that they become a noticeable factor. However it is exactly this case that is also the most problematic for the step mode, the bad complexity having the most effect. Preliminary tests on step-wise sorting confirmed this in practice, even on very small testsets, increasing computation times by a large factor.

Following this, we will only regard the more general case defined by the block mode, where we define N to be the total number of all elements over all sorted sublists and M the number of sublists that have to be merged. Merging two sublists of length l_1 and l_2 results in a sorted list of length $l = l_1 + l_2$, w.l.o.g. $l_1 \leq l_2$.

***S2** List-Join:* The most intuitive algorithm to merge M sorted lists is to iteratively build a new list, for each element testing which element of the M sublists needs to be added next. This gives a trivial complexity of a guaranteed runtime of $O(N \cdot M)$ and space complexity of $O(N)$. Disregarding initial position, every element needs to be moved to the new list during the merge.
***S3** Queue-Join:* The behaviour of *List-Join* can be improved by using a sorted queue to find the next element to be added. Inserting into a sorted container has a complexity of $O(\log(M))$, effectively reducing the complexity of the whole merge to $O(N \log(M))$, while the space complexity remains unchanged.
***S4** In-Join:* The In-Join algorithm was developed to reduce the number of elements that have to be moved in memory during the merge. Its basic functionality is the same as *List-Join*, but each element is written directly to the correct position of the input list if necessary, otherwise, it is left untouched. This, however, creates the need for a temporary element queue of displaced candidates. Even worse, elements do not need to be inserted to the queue in order, as can be seen in step 4 in the example in Fig. 5. This adds a factor of $O(\log(N))$ to some operations, creating a worst case runtime of $O(N \cdot M \cdot \log(N))$ while space complexity remains at $O(N)$. Pseudo-code for a full implementation can be found in the appendix as Alg. B.3.

Step	**List**			**Queue**
1	$3 \quad \underset{\uparrow}{4}$	$\underset{\uparrow}{1}$	$0 \quad 2 \quad \underset{\uparrow}{\dot{5}}$	ϵ
2	$3 \quad \underset{\uparrow}{4}$	$\underset{\uparrow}{1}$	$0 \quad \underset{\uparrow}{\dot{2}} \quad 5$	ϵ
3	$\underset{\uparrow}{3} \quad \_$	$\underset{\uparrow}{1}$	$\underset{\uparrow}{\dot{0}} \quad 4 \quad 5$	2
4	$\_ \quad \_$	$\underset{\uparrow}{\dot{1}}$	$3 \quad 4 \quad 5$	2, 0
5	$\_ \quad \_$	$\underset{\uparrow}{\dot{2}}$	$3 \quad 4 \quad 5$	1, 0
6	$\_ \quad 1$	2	$3 \quad 4 \quad 5$	0
7	$0 \quad 1$	2	$3 \quad 4 \quad 5$	ϵ

Figure 5: Short example of sorting a list with three sublists using the S4 In-Join algorithm. A dot $\dot{x}$ indicates the current element to sort. $\uparrow$ shows current ends of sublists to consider for sorting. *Step 1:* 5 is left untouched because it is already sorted. *Step 2:* 4 is worst element, so 2 is displaced to queue. *Step 3:* 0 is displaced to queue. First list is now exhausted. *Step 4:* 2 from queue is the worst and 1 is displaced. Adding 1 to end of queue would create faulty order, so sorted insert is needed. *Step 5-6:* Write queue to current element. *Step 7:* Sorting finished.

Instead of joining all sublists at the same time, a two way mergesort can be utilized, the used merge step for two lists characterizing the process.
S5 *Merge A:* A merge can be achieved in guaranteed l comparisons and maximally l swaps ($2l$ moves) if additional space is available. *Merge A* starts at the right-most (worst) elements of both sublists. At each step, the biggest (worst) current element is written to the current index of the right sublist. If the right element was already the worst, nothing happens, otherwise the element is displaced to a temporary queue. This element is guaranteed to be bigger or equal than the next element of the right list, and elements are inserted in order (by the fact that the right list is sorted). Hence, if elements in the temporary list are present, only one comparison between the current element of the left list and the next of the queue is needed. In the worst case, the queue gets as long as the smaller of both sublists. Although this algorithm is not in place, it tries to minimize both the size of additional memory as well as memory operations. Due to the linear merge, the whole sorting process has a complexity of $O(N \log(M))$. An example of a sorting process using *Merge A* is shown in Fig. 6. Pseudo-code is given in the appendix in Alg. B.4.
S6 *Merge B:* The previous algorithm can be extended to move all elements, from the right list, smaller than the first elements of the left list in a consecutive block. This trades in additional comparisons in the hope of gaining a speed-up from the more efficient memory operation. The complexities remain unchanged.
S7 *Merge In-Place:* Finally, the merge of two sorted lists can be done in place. The C++ STL package defines a function *inplace_merge* that defines two sub-algorithms called depending on the availability of an additional (constant) amount of memory. Both algorithms are based on Recmerge of Dudzinski and Dydek [32] that recursively reduces the size problem using rotations around central elements. If additional memory is available, it is used as a temporal storage to sort elements in blocks for smaller subproblems, reducing the complexity from $O(l_1 log(l_2/l_1 + 1))$ comparisons and $O((l_1 + l_2)\log(l_1))$ swaps to a linear behaviour. The whole sort takes $O(N \log(N) \log(M))$ or $O(N \log(M))$, respectively. We can assume for the experiments that the fastest case is called almost all the time.

Step	**List**									**Queue**
1	2	6	8	$\underset{\uparrow}{9}$	|	3	4	5	$\dot{7}$	ϵ
2	2	6	$\underset{\uparrow}{8}$	⑦	|	3	4	$\dot{5}$	9	7
3	2	$\underset{\uparrow}{6}$	⑤	⑦	|	3	$\dot{4}$	8	9	7, 5
4	2	$\underset{\uparrow}{6}$	⑤	④	|	$\dot{3}$	7	8	9	5, 4
5	$\underset{\uparrow}{2}$	③	⑤	$\dot{④}$	|	6	7	8	9	5, 4, 3
6	$\underset{\uparrow}{2}$	③	$\dot{⑤}$	5	|	6	7	8	9	4, 3
7	$\underset{\uparrow}{2}$	$\dot{③}$	4	5	|	6	7	8	9	3
8	$\underset{\uparrow}{2}$	3	4	5	|	6	7	8	9	ϵ

Figure 6: Short example of merging two list with Merge A. A dot $\dot{x}$ indicates the current element to sort. ↑ shows the rightmost element of the left list to sort. Circled elements ⓧ show which element was sent to the queue by freeing this position. *Step 1:* 9 is worse than 7, so 7 is displaced to queue. *Step 2:* 8 is worse than first element of the queue 7, so 5 is displaced. *Step 3:* First element of queue 7 is worse than 6, so 7 from queue displaces 4. *Step 4:* 6 is worse than 5, so 3 is displaced to queue. *Step 5-7:* Empty queue. *Step 8:* Sorted, because right list is processed and queue is empty.

6.2 Optimization

The degree of optimization needed in order to perform well varies greatly between algorithms. All implementations are implemented as part of the GAP-M library or the C++ STL library. As such, they are called over templated interfaces and employ comparator objects for comparisons. No comparator calls for individual dimensions are needed, but rather all dimensions are tested in a single function, comparing higher dimensions only if needed.
As a general rule, sorting can only be done on the highest level of candidate lists. This means, for the normal implementation, the sorting is applied before the top-most choice function. In the generalized implementations, the candidate list itself is directly modified in non-terminal recursions. It is this fact that allows the usage of in place algorithms in the first place. As a welcome side effect, if a C++ 11 compatible compiler is available, we can use the move function over the more expensive copy operation. This change is automatically detected and employed by all of the above algorithms. The output of all algorithms is returned in the pointer of the input list that is passed as reference, although the returned list is not necessarily the same. This avoids the application of the copy operator as would be the case by using a return statement.

Algorithms S1 and S7, as part of the GNU C++ STL package, can already be regarded as fully optimized. The formulations for S2 and S3 do not leave much room for improvement. S2 does not keep track of exhausted sublists to avoid costly lists constructions and element modifications that are likely not needed most of the time. For the sorted queue in S3 a *vector* is used as recommended by the library. A *vector* can be seen as a consecutive array in RAM with higher end functionality. *Vectors* are considered to be very fast when operating on small elements, in this case, integer-sized indices. The number of sublists is known, so memory for all is reserved before initially filling it.

The sorted queue of S4 is set up the same way. The rest of the implementation is straightforward: find the worst element on all sublist ends, send the current element to queue if needed, and write the worst element to the current index.

For S5 and S6, an observant reader may ask why the implementation cannot be done fully in place, in other words, keeping the queue within the original elements of the list. After all, each time an element is displaced to the queue, another position within the list is freed. Even more so, elements are displaced in order as the sublists are sorted, and freed regions are guaranteed to be connected. A problem only arises when elements from the queue displace new elements. In a free queue implementation, the first element can be removed, and a new element be added in $O(1)$. In Fig. 6, an example of a sorting with Merge A is shown, also indicating which positions were freed when displacing candidates. In step 4, the queue order is inverted, meaning the queue is no longer read from right to left, but now from left to right. In step 5, we completely lose order within the queue. Longer examples that fragment the queue area even more can be easily constructed. Following this rationale, while the algorithm can be made in place, it would cost a factor of $O(l)$ – for shifting all existing queue elements to the right when an element is removed from the queue – to keep the queue sorted.
Instead of implementing a single loop containing all intelligence, S5 and S6 were broken down to several smaller loops. This is done to minimize faulty branch predictions for repeated branches within the main loop. Spreading the logic over multiple loops reduces the need for branching within each loop to a minimum, although some code is duplicated.

7 Pareto eager ADP

Unlike with the sorted implementation, we are still missing essential parts of the definitions of Pareto eager ADP. As the *nullary* function, every definition from Section 4 can be used (see Eq. 34). Co-optimals are retained by the product tree if defined and needed (see Section 5.2). The *choice* function is set up as the identity (Eq. 30) regardless of defined products in forward mode, or is generated without change to the standard implementation for tabulated non-terminal functions. Most complicated is the implementation of the Pareto merge operation $\overset{p}{\vee}$ for Eqs. 31 and 32. Saule and Giegerich identified an algorithm for $\overset{p}{\vee}$ that can join sorted Pareto fronts for two dimensions in $O(N)$ [13], but left open practical issues and possible alternative implementations for higher dimensions.

The biggest concern is that we need two definitions of $\overset{p}{\vee}$, one that merges by the definition of **pf** and one by the definition of **pfc**. This difference has not been addressed in any work so far.

There are more issues we need to address before introducing different algorithms that can solve both versions, however.

7.1 Floating Point Operations and Pareto Front Inaccuracies

In Section 2.3.2, we saw the definition of the build $\otimes$ steps in Eqs. 32 and 33. One of the key properties of the build phase is that applying an algebra function to a Pareto front will again result in a Pareto front, as all such functions are required to be strictly monotone [13]. The proof of correctness for individual definitions is left to the user by the Bellman's GAP system. In Section 3.6.4, we already discussed the problem of unsafe floating point operations as a general issue. We will now see how this can be critical for Pareto eager implementations.
Let's assume we have two values representing the for this example arbitrarily chosen number 1.04. Due to floating point inaccuracies, they are represented as, say 1.04 and 1.039, hence posing different values for a full comparison. Let's say we are maximizing over a two-dimensional Pareto front, and the two values are paired with other values as follows:

$$[(1.04, 2), (1.039, 3)].$$

If both values were the same – as they should be ideally – the Pareto front would only keep one element. However, due to the inaccuracies, both elements will be kept. For the standard implementation and the sorted implementation, this behaviour does not pose a massive problem, as they have a self-correcting tendency. More formally, all elements are considered for computing the Pareto front every time, so if an element is retained for one front and the accuracy error is (by chance) amended later, the element will still be removed. So if a function shows behaviour that is not strictly monotone – for our example, the 1.039 value catching up to the 1.04 one – the standard and the sorted implementation will amend for that. For the sorted case, this is less obvious, as technically the sorting might be invalidated by rounding errors. But since elements are technically the same, they remain sorted on an abstract level.

The Pareto eager implementation does not always have this property. Since the Pareto merge operates on the knowledge of existing fronts, fronts don't need to be re-evaluated by the elements in them. So for the example above, once the two elements are in a front, they will not leave it except if elements from another front dominate them. So, if the strict monotonicity is violated, the faulty elements will remain in the front.

There is a relatively simple, but yet not completely safe, solution for this: restricting comparisons to act only within a certain level of accuracy that has to be chosen to be greater than the level of floating point accuracy. This choice is left to the user who can define it using the parameter introduced in Section 3.6.4.

7.2 Pareto Merge Algorithms

All algorithms of this section can be modelled after Pareto operators defined in Section 4. Accordingly, they can be defined with varying arities in mind and work on sorted or unsorted Pareto fronts. Algorithms are designed in a way that they can compute the Pareto front according to **pf** and **pfc** depending on a binary switch.

Algorithm 7.1 drop for $\overset{p}{\vee}_{lex}$

```
INPUT: compare element $e$, index to advance $it$, list $l$,
       binary variable $coopt$ (true: keep co-optimals)
OUTPUT: modified $it$
  $(-, x_2) \leftarrow e$
  $(-, y_2) \leftarrow l[it]$
  if $coopt == $ true then                                  ▷ Coopt
      while $x_2 > y_2$ and $it < \text{length}(l)$ do          ▷ Compare Dim. 2
          $it \leftarrow it + 1$
          $(-, y_2) \leftarrow l[it]$
  else
      while $x_2 \geq y_2$ and $it < \text{length}(l)$ do        ▷ Compare Dim. 2
          $it \leftarrow it + 1$
          $(-, y_2) \leftarrow l[it]$
  return $it$
```

We will start with an algorithm inspired by the two-dimensional definition of $\mathbf{pf}_{lex}$ that can join sorted Pareto fronts. Its basic properties and definition were described by Saule and Giegerich [13]. To better highlight implementation details, we will use index based list access in the pseudo-code:

- $l[i]$ accesses the i'th element of a list l, starting at index 0

We split up the definition of $\overset{p}{\vee}_{lex}$ in two functions, Algs. 7.1 and 7.2. While the definition as pseudo-code is relatively lengthy, the basic idea is easy to describe. We keep markers on both lists indicating which elements have to be merged next. On each iteration, the two next elements are compared to find which one needs to be added next (on the first dimension), at the same time ensuring order on the second dimension. The nested cases cover all combinations of comparisons for two dimensions. The **drop** subfunction iterates over elements that are unsorted in the second dimension. In order to retain co-optimals, only minor changes needed to be made. When we merge two lists of length l_1 and l_2, the complexity of this algorithm is clearly $O(l_1 + l_2)$. In the block mode, we have to merge M sublists at once, which can be done utilizing a two way merge. In this case we gain a runtime of $O(N \log(M))$.

As with $\mathbf{pf}_{lex}$, the good complexity cannot be kept for higher-dimensional merges, but we can reuse the multidimensional definition the same way as for the two-dimensional case, merging two sorted lists and computing the Pareto front at the same time. Unlike for the two-dimensional case, the similarity is easy to highlight like in Alg. 7.3. Grey areas

Algorithm 7.2 $\overset{p}{\vee}_{lex}$

```
INPUT: sorted Pareto fronts $l_1$ and $l_2$,
       binary variable $coopt$ (true: keep co-optimals)
OUTPUT: sorted Pareto front in $answers$
  $it_1 \leftarrow 0$
  $it_2 \leftarrow 0$
  $answers \leftarrow \epsilon$
  while true do
    if $it_2 == \text{length}(l_2)$ then                              ▷ End 1
      write rest of $l_1$ to $answers$
      return
    if $it_1 == \text{length}(l_1)$ then                              ▷ End 2
      write rest of $l_2$ to $answers$
      return
    $(x_1, x_2) \leftarrow l_1[it_1]$
    $(y_1, y_2) \leftarrow l_2[it_2]$
    switch compare $x_1$, $y_1$ do                                 ▷ Compare Dim. 1
      case $x_1 < y_1$
        switch compare $x_2$, $y_2$ do                             ▷ Compare Dim. 2
          case $x_2 < y_2$
            $it_1 \leftarrow it_1 + 1$
            $it_1 \leftarrow$ drop $l_2[it_2], it_1, l_1, coopt$
          case $x_2 == y_2$
            $it_1 \leftarrow it_1 + 1$
        $answers \leftarrow answers : (y_1, y_2)$                         ▷ Add
        $it_2 \leftarrow it_2 + 1$
      case $x_1 == y_1$
        switch compare $x_2$, $y_2$ do                             ▷ Compare Dim. 2
          case $x_2 < y_2$
            $it_1 \leftarrow it_1 + 1$
            $it_1 \leftarrow$ drop $l_2[it_2], it_1, l_1, coopt$
            $answers \leftarrow answers : (y_1, y_2)$                     ▷ Add
            $it_2 \leftarrow it_2 + 1$
          case $x_2 == y_2$
            $answers \leftarrow answers : (x_1, x_2)$                     ▷ Add
            if $coopt ==$ true then                                 ▷ Coopt
              $answers \leftarrow answers : (y_1, y_2)$                   ▷ Add
            $it_1 \leftarrow it_1 + 1$
            $it_2 \leftarrow it_2 + 1$
          case $x_2 > y_2$
            $it_2 \leftarrow it_2 + 1$
            $it_2 \leftarrow$ drop $l_1[it_1], it_2, l_2, coopt$
            $answers \leftarrow answers : (x_1, x_2)$                     ▷ Add
            $it_1 \leftarrow it_1 + 1$
      case $x_1 > y_1$
        switch compare $x_2$, $y_2$ do                             ▷ Compare Dim. 2
          case $x_2 == y_2$
            $it_2 \leftarrow it_2 + 1$
          case $x_2 > y_2$
            $it_2 \leftarrow it_2 + 1$
            $it_2 \leftarrow$ drop $l_1[it_1], it_2, l_2, coopt$
          $answers \leftarrow answers : (x_1, x_2)$                       ▷ Add
          $it_1 \leftarrow it_1 + 1$
```

Algorithm 7.3 $\overset{p}{\vee}_{lex}$ multi-dimensional

```
INPUT: sorted Pareto fronts $l_1$ and $l_2$,
       binary variable $coopt$ (true: keep co-optimals)
OUTPUT: sorted Pareto front in $answers$
  $answers \leftarrow \epsilon$
  $it_1 \leftarrow 0$
  $it_2 \leftarrow 0$
  while $it_1 <$ length($l_1$) or $it_2 <$ length($l_2$) do
      if $it_1 <$ length($l_2$) then
          $(x_1, \ldots, x_k) \leftarrow l_1[it_1]$
      if $it_2 <$ length($l_2$) then
          $(y_1, \ldots, x_k) \leftarrow l_2[it_2]$
      if $it_2 ==$ length($l_2$) or ($it_1 \neq$ length($l_1$) and (
            $y_1 < x_1$ or ($y_1 == x_1$ and $y_2 < x_2$) or ... or
            ($y_1 == x_1$ and ... and $y_{k-1} == x_{k-1}$ and $y_k < x_k$))) then        ▷ Lazy
          $(u_1, \ldots, u_k) \leftarrow l_1[it_1]$
          $it_1 \leftarrow it_1 + 1$
      else
          $(u_1, \ldots, u_k) \leftarrow l_2[it_2]$
          $it_2 \leftarrow it_2 + 1$
      if $answers == \epsilon$ then
          $answers \leftarrow (u_1, \ldots, u_k)$                                          ▷ Add
          continue
      if $coopt ==$ true then                                                            ▷ Coopt
          $answers_{\text{pre}}$:$(t_1, \ldots, t_k)$:$\epsilon$
          if $t_1 == u_1$ and ... and $t_k == x_k$ then                                  ▷ Lazy
              $answers \leftarrow answers$:$(u_1, \ldots, u_k)$                          ▷ Add
              continue
      $add \leftarrow$ true
      for all $answers_{\text{pre}}$:$(z_1, \ldots, z_k)$:$answers_{\text{suf}}$ do
          if $u_2 \leq z_2$ and ... and $u_k \leq z_k$ then                              ▷ Lazy
              $add \leftarrow$ false
              break
      if $add$ then
          $answers \leftarrow answers$:$(u_1, \ldots, u_k)$                              ▷ Add
```

remain unchanged. Blue areas were modified from the original implementation to allow the algorithm to work on two inputs. Instead of iterating over one sorted list, we now iterate over two sorted fronts. So at each step, we need to test which element comes next in the sorting. Once this element is established, the algorithm continues as before. The test for co-optimals can be done in $O(1)$, as, due to the sorting, all equal elements follow each other and only the last element of the answers front has to be checked. Overall, complexity is $O(l \cdot l)$. For block mode, applications can be arranged in a two way merge like before, exhibiting a complexity of $O(N^2)$ by Master's theorem. The influence of M is subsumed by the influence of N.

The modification of $\mathbf{pf}_{nosort}$ is an especially interesting case of the Pareto eager implementations. All other implementations require to create new answer lists containing the Pareto front. $\overset{p}{\vee}_{nosort}$ in Alg. 7.4 allows us to work directly on the input lists. Since both inputs are already Pareto fronts, we can declare the longer one to be the answers list, and insert the candidates of the second front into it. In other words, we compare all elements of one list to all elements of the other, making changes to the first list accordingly. Again, grey areas remain unchanged and blue areas were modified. The biggest problem is to handle co-optimals effectively. To avoid calling a comparison twice, the original branching has been modified so that all dimensions are tested in one loop, marking properties to new variables as they occur. This procedure was set up in a way that comparisons are stopped

Algorithm 7.4 $\overset{p}{\vee}_{nosort}$ all dimensions

```
INPUT: sorted Pareto fronts $l_1$ and $l_2$,
       binary variable $coopt$ (true: keep co-optimals)
OUTPUT: sorted Pareto front in $l_1$
  for all $l_{2,\text{pre}}$:$(u_1,\dots,u_k)$:$l_{2,\text{suf}}$ do
      $add \leftarrow$ true
      for all $l_{1,\text{pre}}$:$(x_1,\dots,x_k)$:$l_{1,\text{suf}}$ do
          $less \leftarrow$ false
          $better \leftarrow$ false
          for $j = 1; j \le k; j = j + 1$ do
              switch compare $u_j$, $x_j$ do                              ▷ Compare Dim. $j$
                  case $u_j < x_j$
                      $better \leftarrow$ true
                  case $u_j > x_j$
                      $less \leftarrow$ true
              if $better$ == true and $less$ == true then                 ▷ Lazy
                  break
          if ($better$ == true and $less$ == false) or ($better$ == false and
                $less$ == false and $coopt$ == false) then                ▷ Coopt
              $add \leftarrow$ false
              break
          else if ($better$ == false and $less$ == true) then
              $l_1 \leftarrow l_{1,\text{pre}}$:$l_{1,\text{suf}}$                               ▷ Remove
      if $add$ then
          $l_1 \leftarrow l_1$:$(u_1,\dots,u_k)$                                    ▷ Add
```

once everything we need is established. Allowing or disallowing co-optimals then boils down to just one additional condition in an existing branch. The complexity for merging two lists is $O(l_1 \cdot l_2)$. Both using the step mode as well as using the block mode, we will perform no more comparisons or memory operations as by applying $\mathbf{pf}_{nosort}$. Joining M fronts in block mode accordingly costs $O(N^2)$, employing a two way merge.

Algorithm 7.5 comarry

```
INPUT: unsorted Pareto fronts $X$ and $Y$, cut-off size $c$
OUTPUT: $X$ without elements dominated by $Y$, $Y$ without elements dominated
  by $X$
  if length($X$) ≤ $c$ or length($Y$) ≤ $c$ then                    ▷ Recursion End
      $X' \leftarrow \mathbf{marry}_{brute}(X, Y)$
      $Y' \leftarrow \mathbf{marry}_{brute}(Y, X)$
      return $(X', Y')$
  else
      choose a cut plane to divide $X$ to $X_1, X_2$ and $Y$ to $Y_1, Y_2$ st. $X_1$ superior $X_2$,
          $Y_1$ superior $Y_2$, $X_1$ superior $Y_2$, $Y_1$ superior $X_2$          ▷ Divide
      $(X_1', Y_1') \leftarrow \mathbf{comarry}(X_1, Y_1)$                        ▷ Recursion
      $(X_2', Y_2') \leftarrow \mathbf{comarry}(X_2, Y_2)$                        ▷ Recursion
      $X_2'' \leftarrow \mathbf{marry}(X_2', Y_1')$
      $Y_2'' \leftarrow \mathbf{marry}(Y_2', X_1')$
      return $(X_1' : X_2'', Y_1' : Y_2'')$
```

Lastly, a modification of the $\mathbf{pf}_{yuk}$ implementation can be adapted to merge two unsorted Pareto fronts. Of course the fronts could be simply appended and then $\mathbf{pf}_{yuk}$ applied to them. This, however, would practically result in the same algorithm as applying $\mathbf{pf}_{yuk}$ with the standard implementation. Instead, an algorithm can be conceived as the combination of the **marry** step and the entry function of $\mathbf{pf}_{yuk}$; we call it **comarry**, Alg. 7.5. The idea is to recursively split up both input lists with the same median until a cut-off size is reached. Since subsets of Pareto fronts are also Pareto fronts, we don't need to call $\mathbf{pf}_{lex}$ after the cut-off as before; Rather we need to check for dominating elements between

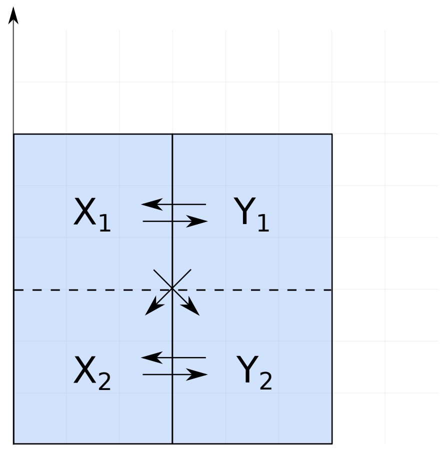


Figure 7: Sketch of the domination in **comarry**. Arrows indicate possible domination between lists. We have only one dimension of domination. Horizontal arrows are solved recursively by $\mathbf{marry}_{brute}$, vertical arrows by **marry**.

both inputs lists, which can be done using $\mathbf{marry}_{brute}$ (see Alg. B.2). Similarly, we don't need to check for domination within each front, but rather across splits for both. This situation is shown in Fig. 7. For the vertical splits, we can reuse the **marry** algorithm as before, only now it has to be applied twice. $\overset{p}{\vee}_{yuk}$ only needs to call **comarry** and append both answer lists into one answer. Merging two Pareto fronts has the same complexity as creating a new one, so the merge takes place in $O(l\log^{d-1}(l))$ if we have d dimensions. However, this means for merging M fronts by a two way merge, we gain a factor of $\log(M)$, resulting to a total complexity of $O(N\log^{d-1}(N)\log(M))$.

On first glance, it seems that none of the above algorithms improve the asymptotic behaviour of their counterparts of the standard implementation. Even worse, $\overset{p}{\vee}_{yuk}$ deteriorates the overall complexity. For the multi-dimensional $\overset{p}{\vee}_{lex}$, we can assume additional runtime factors compared to the multi-dimensional $\mathbf{pf}_{lex}$, brought in by added memory operations and the two way merge. For $\overset{p}{\vee}_{nosort}$, it is not as clear what assumptions can be made.
Only for the two-dimensional case of $\overset{p}{\vee}_{lex}$, we can actually enhance complexity. Because of the sorting step prior to applying the two-dimensional $\mathbf{pf}_{lex}$, it has a total complexity of $O(N\log(N))$, while $\overset{p}{\vee}_{lex}$ has $O(N\log(M))$. Since $M \leq N$, this is an improvement. Constant factors, e.g. by additional memory operations, might still invalidate this method, however.

Returning for a moment to the problem of floating point accuracy, it is interesting to note that, contrasting to all other algorithms, the multi-dimensional implementation of $\overset{p}{\vee}_{lex}$ has self correcting potential. For all other implementations, faulty elements can remain in the front.

7.3 Optimization

Optimizations for all algorithms except $\overset{p}{\vee}_{yuk}$ are relatively straightforward. A few general rules for all implementations in C++ and the same general rules as before apply. The choice to pass *coopt* as a parameter for co-optimality was done to keep manual code duplication low, therefore increasing maintainability. The performance cost for this is low. All functions are marked for inlining and the parameter is passed as a constant value, so

the compiler is likely to duplicate the function itself and, even if not, the level of faulty branch predictions is kept minimal. Pareto fronts are not returned by a return statement, but are passed as reference to avoid unnecessary calls to the copy operator. If C++ 11 is available, the less expensive move function can be used instead of copy (marked as Add), as we work directly on the global candidate list.
For the two-dimensional $\overset{p}{\vee}_{lex}$, switch case is used instead of branching with *if* to reduce false branch predictions and speed up code jumps [28]. Comparator calls can be reduced to exactly one call for each dimension.
For the multi-dimensional $\overset{p}{\vee}_{lex}$, we need two levels of comparisons: one to find the next element according to the sorting, and one for comparison within the front. The first comparison always includes all dimensions, so a single comparator comparing all dimensions at once is used. All other comparisons are done by comparator calls over single dimensions.
For $\overset{p}{\vee}_{nosort}$, all comparisons are made within one loop so that comparators are called at most once for each dimension. The comparison loop is likely to be unrolled since the dimension is small and constant. It should be noted that since lists are implemented as deques, removing elements takes $O(N)$ as with $\mathbf{pf}_{nosort}$.
Except for the entry functions, the same subfunctions are used for $\overset{p}{\vee}_{yuk}$ as for $\mathbf{pf}_{yuk}$. The median is chosen by the same method as well. While **comarry** is defined recursively, it is implemented iteratively like all other functions of the Yukish implementations to improve locality. The Yukish merge was only implemented so that intermediate lists contain pointers to elements in the global input list. When merging M fronts at once, references for candidates are not created for each merge individually, but globally once so that no candidate has to be transformed more than once.

The most pressing question for optimality is whether to use implementations in step mode or block mode. For $\overset{p}{\vee}_{lex}$ and $\overset{p}{\vee}_{yuk}$, the answer is simple, since for each call, each element of the front needs to be moved to a new list. The inferior complexity of the step mode (see Section 3.4.1) will therefore cause a lot of unnecessary memory operations. For $\overset{p}{\vee}_{nosort}$, the bigger front is always modified by elements of the smaller front. There is no significant difference between the step mode and the block mode in terms of memory operations or comparisons. Instead, the mechanism of the two-way merge in block mode creates a large, unnecessary overhead. To confirm this effect in practice, we will test both modes in the next section.

8 Benchmarks

A number of different experiments were performed to evaluate different aspects of the various implementations. In a dynamic programming application, intermediate results are generated from a search space with particular properties and will be far from random. Therefore, it is important to test all algorithms not just on artificial data, but also on real world scenarios. Due to the nature of Bellman's GAP, existing descriptions of bioinformatics applications that were already coded in GAP-L were used for this work without modification.

Due to the many different strategies and concepts described in this work so far, benchmarks will be split up into multiple steps. First, we will describe the basic setup that was used for all tests. Since the defined test cases vary in the number of dimensions of the Pareto products and floating point accuracy is critical for some of these, we will have a closer look at the influence of both on the programs. The Pareto operator $\mathbf{pf}_{yuk}$ has multiple competing implementations, and efficiency highly depends on the chosen cut-off size. Likewise, lexicographically sorted ADP and Pareto eager ADP will be examined individually before all implementations will be combined into one final test.

We will set a special focus on how the different components of the ADP definition, namely evaluation algebras, tree grammars and the input, influence the effectiveness of implementations.

8.1 Setup

All experiments were run on a cluster system using an Intel(R) Xeon(R) E7540 CPU clocking at 2.00 GHz. Each task was restricted to only one CPU and maximally 100 GiB RAM. Only RAM that was physically bound to the CPU was allowed for computations to minimize memory access times. Applications were coded in GAP-L. All C++ code was fully automatically produced by the latest in-house version of Bellman's GAP and not modified manually, except for adding additional functions needed for time profiling. Existing GAP-L files were not modified. All variations are created solely by different parameters of GAP-C.
Tests are executed in two steps when applicable. To achieve accurate comparisons, implementations are tested against each other on the level of individual calls on intermediate candidate lists, excluding the runtime of the surrounding code, e.g. the iteration through the search space. Only when the ideal implementation was determined, full running times, including the full ADP program, were measured. Individual runtimes were determined using the Boost Timer Library[5], for measurements within a program, or the Unix time command, for total runtimes, respectively. Times were averaged over at least two datapoints to ensure accuracy and correctness.
The generated code was compiled using g++ version 4.8.3 with C++ 11 Standard support and *-O3* for optimization. For code generation, the option *–kbacktrace* was given, separating the computations into a forward phase, computing the Pareto front, and a backtracing phase to compute string representations of all solutions in the Pareto front, in part recomputing it. This means that all measurements contain computation times from varying object sizes.

[5]http://www.boost.org/libs/timer/ [18.09.2015]

Application	Name	Dim.	Element Size (byte)	Dataset
RNA Alignment Folding	Ali2D	2	12	Rfam seed alignments[33]
RNA Alignment Folding	Ali3D	3	16	Rfam seed alignments[33]
Gotoh's algorithm	Gotoh2D	2	8	BAliBASE 3.0 [34]
RNA Folding	Fold2D	2	12	Rfam seed alignments[33]
RNA Folding	Fold3D	3	16	Rfam seed alignments[33]
RNA Folding	Prob2D	2	12	RMDB [35]
RNA Folding	Prob3D	3	20	RMDB [35]
RNA Folding	Prob4D	4	28	RMDB [35]

Table 2: Summary of test cases with essential properties and used databases as input. Element sizes are estimates only for the forward phase of the computation and are only valid for the used test system. They may vary on other systems or differ in reality due to memory padding. For the backward phase, string representations of candidates are added, so no general estimate can be made on their size.

8.2 Definition of ADP Tests

To achieve a both varied and representative benchmark, we tested, in total, four real world applicable ADP applications of biosequence analysis. Each uses a two-dimensional Pareto operator and, if suitable, also a three or four-dimensional operator, totalling seven main test cases shown in Table 2. The applications Ali, Fold and Prob execute the same basic task of RNA structure prediction, using the same tree grammar, but varying in the specified evaluation algebras. Gotoh performs sequence alignment under an affine gap model. More complete definitions are given in the following subsections. For each application, realistic test sets were downloaded, and a subset of inputs of different sizes were manually extracted. For detailed analysis, a number of different inputs of varying sizes were tested. Where applicable, only representative results of preferably large inputs will be shown to lower the amount of data presented in this work. Only inputs were chosen for which profiling data could be extracted within 5 days of computation time. While 5 days seems to be a rather large, and for many applications sufficient, time frame, this poses a strong restriction especially on computations with more than two dimensions, as we will see shortly.

The asymptotic complexity of the tested ADP applications can be described in terms of the initial input length n, as well as the runtime p and the space s of the Pareto computations per subproblem. The Pareto front is computed once per table entry, therefore, complexities multiply. The fold programs' asymptotic complexity is $O(n^3p)$ in time and $O(n^2s)$ in space. The alignment program's asymptotic complexity is $O(n^2p)$ in time and $O(n^2s)$ in space. The input length will be given as the number of bases (characters) in the input sequence for the remainder of this work.

Ali and Fold were tested on the the same inputs taken from Rfam seed alignments [33] that contains alignments of sequences known to produce stable structures. No special attention was given to the nature of the optimal structures, only to input length. Chosen alignments are *RF01064* (input length 100, 4 sequences), *RF02187* (input length 200, 24 sequences), *RF00216* (input length 302, 23 sequences), *RF01675* (input length 404, 19 sequences), *RF01478* (input length 509, 9 sequences) and *RF01271* (input length 608, 5 sequences).

The sequences and reactivities used to evaluate Prob originate from the RMDB [35]. The selected dataset contains 146 sequences, but only three were chosen under the condition of length and that data for three different reactivities was available.

Gotoh2D was evaluated with Balibase 3.0 [34]. Again, sequences were chosen solely on length.

8.2.1 Gotoh's Algorithm

The test case Gotoh2D is an implementation of Gotoh's algorithm [16] that solves the pairwise alignment problem for affine gap costs. For Gotoh2D, the gap init cost is combined with the gap extension cost in a Pareto product, and the results are printed as an alignment string via a lexicographic product. A more complete presentation of this grammar will be given in a yet unpublished work.

8.2.2 RNA Folding

The RNA folding space is defined by the overdangle grammar first described by Janssen et al. [36]. Depending on the application and the number of dimensions in the Pareto product, the folding space is evaluated with different algebras. For Fold2D, we use the Minimum Free Energy (MFE), according to the Turner energy model [37], combined with the Maximum Expected Accuracy (MEA) which consists in the accumulation of base pairs probabilities computed with the partition function of the Turner model. For Fold3D, the maximization of the number of predicted Base Pairs (BP) is added to the Pareto product. In Prob we study the integration of reactivity data in RNA secondary prediction with Pareto optimization. The design of algebras is based on distance minimization between probing reactivities (SHAPE [35, 38, 39], CMCT [40] and DMS [41, 42]) and their computed centroids. For Prob2D, the MFE was combined with SHAPE. Prob3D combines MFE, SHAPE, and DMS and Prob4D MFE, SHAPE, DMS, and CMCT, accordingly. A dot bracket string representation and a RNAshape representation of the candidates of the front are printed via a lexicographic product. A more detailed presentation of probing algebra products will be given in a yet unpublished work. As probing reactivities are handled as doubles in the used definitions, comparisons will be handled as exact to 4 decimal places if not noted otherwise. 4 decimal places marks the experimentally chosen highest possible setting before problematic candidates arise for the Pareto eager implementation (see Section 7.1).

8.2.3 RNA Alignment Folding

As Ali, we tested the behaviour of Pareto fronts in an implementation of RNAalifold [18, 19]. We analysed structure prediction with the MFE and MEA algebras and a covariance model algebra COVAR following the definitions of [43]. For Ali2D, only MFE will be combined with COVAR. For Ali3D, MFE, MEA, and COVAR will be combined into the Pareto product. Like for the single sequence RNA folding, a dot bracket string representation and a RNAshape representation of the Pareto front candidates are added via a lexicographic product. It should be mentioned that alignment folding is defined over the same overdangle grammar as the single sequence case, only now, the input alphabet is columns of aligned characters, including gap symbols. Accordingly, for this case, the Rfam seed alignments were left intact, while for Fold only the first sequence was used.

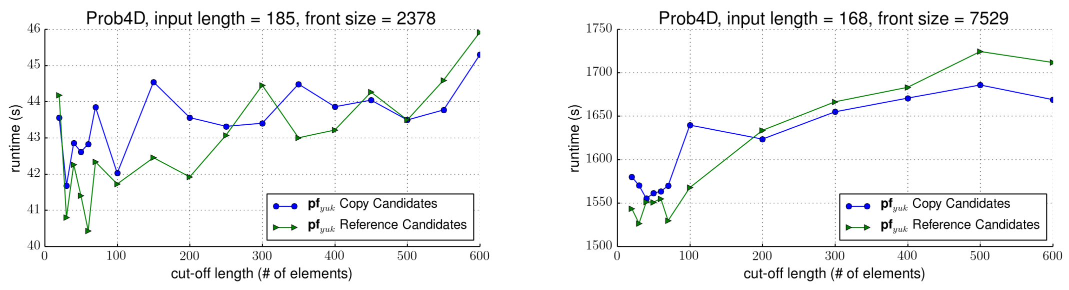
Name	**Compared Decimal Places**							
	Front Size				**Runtime (s)**			
	1	**2**	**3**	**4**	**1**	**2**	**3**	**4**
Input Size 96								
Prob2D	9	16	16	16	0.07	0.09	0.09	0.09
Prob3D	23	27	27	27	0.28	0.31	0.31	0.31
Prob4D	183	188	177	183	1.20	1.36	1.40	1.44
Input Size 185								
Prob2D	8	8	8	8	1.79	2.06	2.08	2.08
Prob3D	103	143	145	146	37.73	52.80	53.85	54.02
Prob4D	1307	2000	2211	2327	504.38	1013.06	1054.85	1214.16

Table 3: Pareto front sizes and runtimes for varying Pareto dimensions and floating point accuracy settings. Runtimes were computed in the standard implementation of ADP with $\mathbf{pf}_{nosort}$ for two-dimensional products and $\mathbf{pf}_{yuk}$ for all other products. Pareto front sizes indicate only the size of the final result. Both runtime and front sizes show an exponetial increase with increasing dimension. Lowering floating point accuracy can lower computation times.

8.3 Influence of Pareto Dimensions and Floating Point Accuracy

Before executing any benchmarks, it is important to explore and understand the influence of parameters other than the implementation of the Pareto front operators and the ADP definition on the computation properties. While it is almost impossible to analyse every component that can influence Pareto front sizes and with that running times, the goal of this section is to at least give the reader a broad idea of what it means in practice to increase the dimension of the Pareto operator. At the same time, we will see what influence different floating point accuracy settings have on the computations.

As Prob is the only case that allows the definition of a four-dimensional product, we will restrict analysis to this application for this section. All possible combinations of products with 2-4 dimensions (Prob2D, Prob3D and Prob4D) and floating point accuracies between 1 to up to 4 exact decimal places were tested using two inputs of different sizes. A summary of the results is shown in Table 3. For all tests, Pareto front sizes increased exponentially with increasing dimensions. Accordingly, computation times increased exponentially as well. Already for the three-dimensional Pareto product, runtime increased by a factor of over 25 for the longer input, less for the smaller input. For input size 185, the two-dimensional front could be computed in about 2 seconds, while the four-dimensional case already took up to more than 20 minutes. Anecdotally, even longer inputs for Prob4D could not finish within multiple days of computation time. Similarly, for Ali3D, the longest fully testable input finished within a few seconds, while the next available test input did not finish at all. As such, theory and practice converge well for this example. It should be clear from these results alone that dimensions should not be added without merit. Careful tests should be set up by the user to ensure problems stay within a sensible range for new applications.
However, the data also shows the strong influence of floating point settings on the runtime, and thus, a possible heuristic to lessen the negative effects for some applications. The bigger the overall Pareto front becomes, the stronger the influence of the floating point is. For the longest input in Prob4D, more than half of the runtime and a bit less than half of the elements of the Pareto front could be saved when only comparing up to one decimal place instead of four. For Prob3D, the improvement is still about one third. The overall behaviour is fairly expected, as with lower accuracy, individual elements can easier dominate other elements in the list. This effect becomes stronger the more elements stand



(a) **(b)**

Figure 8: Comparisons of the Reference (pointer) based and the Copy based implementations of $\mathbf{pf}_{yuk}$. The graphs show the runtime of each implementation for varying length of the cut-off value. Graphs show results for an input of length a) 185 with a final Pareto front of 2378 elements and b) 168 with a final Pareto front of 7529 elements. In both cases, the Reference based version performs better for the best cut-off.

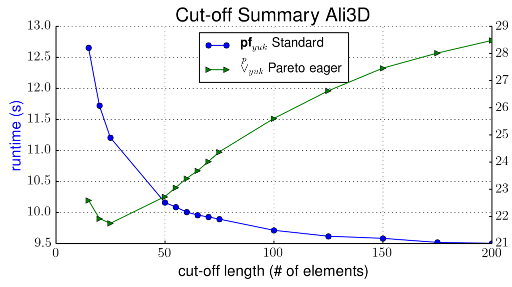
in concurrence to each other. It should be noted that while improvement can be always expected, it is plausible to assume that the high level of improvement we saw for Prob in this test case can not be reproduced for most other algebras, as the effectiveness is highly linked to the spectrum of produced candidate values.
As a further thought, the ultimate goal of the Pareto front should not be forgotten. Reducing candidates might not be a desired effect for all ADP problems, although likely to be uncritical for Prob.

8.4 Optimizing $\mathbf{pf}_{yuk}$ and $\overset{p}{\vee}_{yuk}$

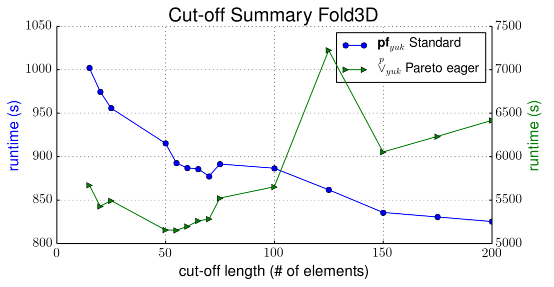
The implementations of $\mathbf{pf}_{yuk}$ and $\overset{p}{\vee}_{yuk}$ can be optimized along multiple criteria. As noted in Section 4.3, a number of different implementation strategies could already be dismissed. In this section, we will analyse the best choices for the two biggest open parameters for performance.

As described in previous sections, the internal lists of the Yukish-inspired algorithms can be implemented in different strategies. As a first choice, elements are copied in full for each intermediate list (Copy). Alternatively, intermediate lists can contain memory pointers to the elements of the input list, thus containing only references to elements (Reference). To determine the best choice between both, a preliminary test was executed on two inputs on Prob4D, guessing a range of possible cut-off values. The results of this test are shown in Fig. 8. Overall, the results seem to vary with different choices of cut-offs. However, as a general trend, the reference based approach seems to be superior for most cases. In both tests, the shortest runtime was achieved using reference lists, with no copy value coming near a similarly good time. While this test is not comprehensive, it serves as a good indicator that when nothing else is known, the reference based method should be used. As such, GAP-C will automatically include the functions of this implementation, while the copy approach can be used by a manual switch in the code. For the Pareto eager implementation $\overset{p}{\vee}_{yuk}$, consecutively only a reference based implementation was created and can be used.

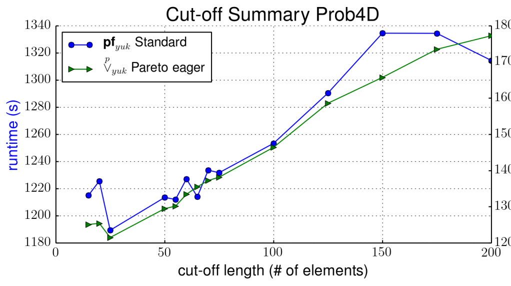
A vital factor for performance is the choice of optimal cut-off values for different ADP applications. As no definition of Gotoh with more than two dimensions was used for this work, it was excluded from the trial. The tested range of cut-off values was determined



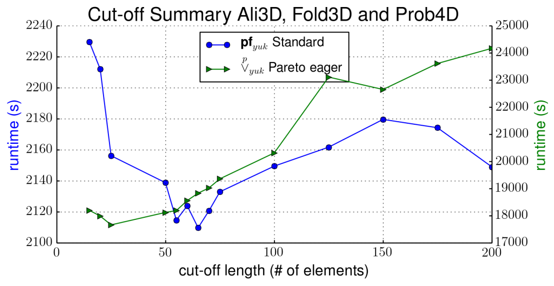
(a)



(b)



(c)



(d)

Figure 9: Runtimes of $\mathbf{pf}_{yuk}$ and $\overset{p}{\vee}_{yuk}$ for different cut-off values. $\mathbf{pf}_{yuk}$ and $\overset{p}{\vee}_{yuk}$ are set in different scales to ease the comparison of general trends. Graphs show summarized runtimes for multiple inputs for a) Ali3D b) Fold3D c) Prob4D and d) all available data. Different scales are used for $\mathbf{pf}_{yuk}$ (left) and $\overset{p}{\vee}_{yuk}$ (right) as they exhibit strongly different runtimes.

prior to the exhaustive test by single test calls to minimize computation effort. The results for all applications are summarized in Fig. 9. Before analysing the data in depth, it is necessary to recall that all applications have been defined over the same grammar. For Ali3D and Fold3D, even the same data has been used as the basis of the input. Yet, applications show strongly different behaviour. Both Ali3D and Fold3D show improvement with rising cut-off values. Fold3D additionally shows a small spike at 65 elements. Most interestingly, for Prob4D the whole trend is inverted, with bigger value causing higher computation time. Taking the equal grammar definitions into consideration, these changes can be attributed to differences in algebra functions. It is possible that even better cut-offs exist for Ali3D and Fold3D by further increasing the cut-off size. Combining the times of all individual tests into one graph (Fig. 9d), however, shows a clear winner at a cut-off at 65 elements. This result is somewhat biased towards computations with longer overall runtime, but one might argue that a good cut-off is especially valuable for these cases. To keep trials simple, 65 was set as a standard value in Bellman's GAP and for all further tests.

As another result of these tests, the strong benefits that can be achieved by varying settings should be noted. As a general rule, the cut-off should be calibrated to fit individual applications if feasible and necessary. It is very possible to save multiple minutes of computation times by different choices even for single inputs.

Analysing the graphs for $\overset{p}{\vee}_{yuk}$ reveals slightly different results. All individual tests exhibit generally low computation times for small cut-off values with computation times increasing more or less smoothly with increasing cut-off values. Except for Fold3D, where the optimum is at 55, the optimal computation time is at 25 elements. Unsurprisingly, the combined plot of all values shows the same trends. As a result, 25 was set as the default value of $\overset{p}{\vee}_{yuk}$. Please note that different scales were used for $\mathbf{pf}_{yuk}$ and $\overset{p}{\vee}_{yuk}$, as the Pareto

Algorithm	Avg. Comp. Calls	Gain (s)	Separator
Trial 1			
S1 Quicksort	26,692.7		
S2 List-Join	1,137,920.0	-	no significant gain
S3 Queue-Join	21,780.1	29.0	Always use Alg. 3
S4 In-Join	716,920.0	-	S4 never performs consistently better
S5 Merge A	**18,043.2**	11.3	use S5 when $\frac{N}{M} \geq 4$
S6 Merge B	18,700.0	7.8	use S6 when $\frac{N}{M} \geq 6$
S7 Merge In-Place	**18,043.2**	13.4	use S7 when $\frac{N}{M} \geq 3$
Trial 2			
S1 Quicksort	222,480.0		
S2 List-Join	50,668,600.0	-	no significant gain
S3 Queue-Join	181,884.0	33.2	when $\frac{N}{M} \geq 8$
S4 In-Join	31,480,100.0	-	S4 never performs consistently better
S5 Merge A	**157,102.0**	44.9	use S5 when $\frac{N}{M} \geq 4$
S6 Merge B	161,502.0	28.1	use S6 when $\frac{N}{M} \geq 7$
S7 Merge In-Place	**157,102.0**	62.7	use S7 when $\frac{N}{M} \geq 3$

Table 4: Maximal runtime gain of S2-7 compared to S1 for randomized uniform trial sets 1 and 2. The number of comparator calls was averaged over all data points, ignoring separators. Separators indicate where an algorithm becomes superior to S1. Separators for runtime gain can operate solely on the total length of the input list N and the number of known sorted sublists M. They are estimated exact to the nearest integer for linear separation and to two decimal places for logarithmic separation, in both cases assuming the simplest form without any offset variables.

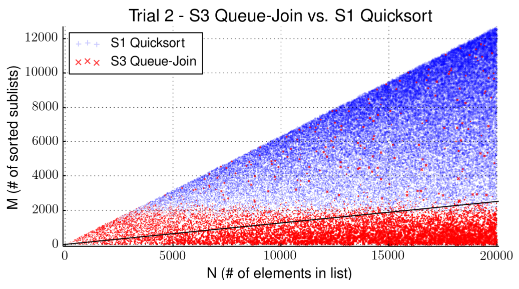
eager merge always performed strongly worse than the standard implementation. This gives us already a strong indicator that $\overset{p}{\vee}_{yuk}$ has no value in practice. It will remain in the further tests as a point of reference, however.

8.5 Lexicographically Sorted ADP

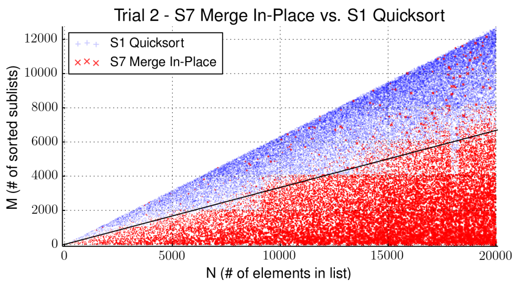
For the lexicographically sorted implementation of ADP, two types of experiments were performed, looking at artificial data and real world scenarios. Experiment 1 uses two random trials to evaluate performance of Pareto front computations based on the sorting algorithms S1-7 defined in Section 6.1. The outcome is of interest for programmers who consider Pareto optimization, but not necessarily in a dynamic programming context. It will also serve as a reference for the second experiment.
In a dynamic programming application, intermediate results are generated from a search space with particular properties. Such candidate lists will be far from random and generally unsorted. In Experiment 2, we therefore test the algorithms on the applications defined at the beginning of this section.

8.5.1 Randomized Trial

Two randomized trials were conducted to confirm the viability of all implementations. For this, we uniformly sample datapoints for $1 \leq N \leq 3,000$ (Trial 1) and $1 \leq N \leq 20,000$ (Trial 2) list elements over $1 < M < N$ sorted sublists. Trial 1 was ended at $200,990$ tests, Trial 2 at $150,505$. All list elements have a size of 22 bytes, bigger than most forward computation, but smaller than all backtracing elements in the next experiment set. The times of S2-7 were compared against the corresponding times of S1, both in total, for all points, as well as utilizing a separator on N and M between the sets. All but one algorithm perform either in $O(N\log(M))$ or $O(N \cdot M)$, compared to $O(N\log(N))$ of S1,



(a)



(b)

Figure 10: Runtime gain of individual data points for a) S3 Queue-Join and b) S7 Merge In-Place against S1 Quicksort in Trial 2. Points are plotted when one algorithm performed better than the other with point sizes relative to the gained time. Maximally $50,000$ uniformly chosen points are plotted per graph. Separators are indicated by a black line.

thus linear or logarithmic separators should be applied respectively. A summary of the results is presented in Table 4.
The clear winner of the first trial is S3 (Queue-Join), outperforming Quicksort across all tested combinations, followed by S7 and S5 with linear separators. In the second trial, S3 moves down to third place, overtaken by S7 and S5. The reason for this is the bad scaling behaviour of S3 that becomes apparent when comparing graphs for S7 and S3. Both algorithms gain over Quicksort when M is small relative to N and sublists hence are long. Fig. 10 shows that S7 (Merge In-Place) gains more than S3 and hence performs better on the larger data set.
For S4 and S2, no noticeable gain could be found in any trial, which likely can be attributed to their inferior complexity, both showing a drastic increase in comparator calls compared to S1. All other algorithms average below S1 on comparator calls. S6 consistently performs worse than S7, the additional tests not yielding any performance boost. In Trial 1, S5-7 clearly outperformed S3 regarding comparator calls while showing longer runtimes. This difference can be explained by the number of memory operations executed by each algorithm. In a randomized setting, only few elements will be initially placed correctly in the list. S3 performs in guaranteed N moves – whereas S5-7 take asymptotically $O(N\log(M))$ moves for this case – amortising the time needed to allocate new memory and destroy the old list on small data. S7 and S5 perform best regarding comparator calls, showing the same average value. Looking at computation times alone, S7 is always slightly faster, however.

8.5.2 Structured Trial

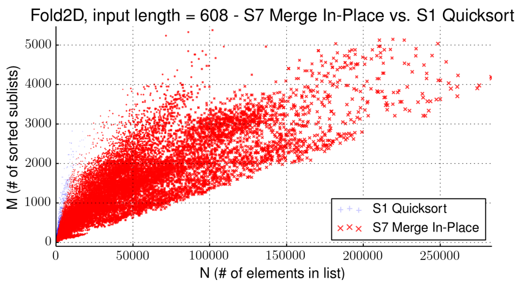
The tests in this section were executed in 2 steps. To achieve an accurate comparison of the sorting algorithms, first, all sorted implementations were tested, excluding the runtime of the surrounding code, e.g. the iteration through the search space. After identifying the algorithm with the biggest time gain, possibly using a separator, a second test was performed, comparing the full running times of all applications, including the full ADP programs. For now, we will restrict the analysis to only sorted implementations. We will see in a later section how they relate to other implementations. A summary of all results is presented in Table 5.

Name	**Input Size**	**Front Size**	**Gain (s)**	**Algorithm**	**Sort (s)**	**Spec. Sort (s)**
Ali2D	509	487	59.1	S7	579.8	530.0
Ali3D	200	446	0.1	S7	5.0	4.9
Gotoh2D	249	209,744	0	S7	29.4	30.3
Fold2D	608	13	304.7	S7	2220.2	1833.3
Fold3D	404	165	40.7	S7	847.0	806.7
Prob2D	185	9	0	S7	3.64	3.6
Prob4D	185	2327	125.1	S7	2518.8	2469.8

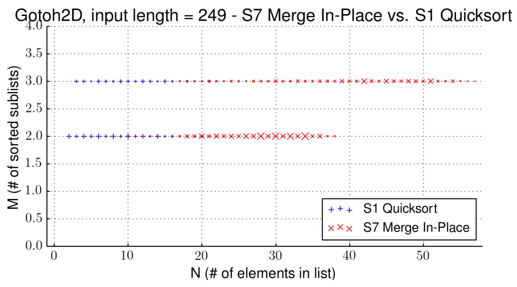
Table 5: Summary of computation times and meta-data of sorted test cases. Left: Isolated sorting time gains over S1. Input length and the final size of the resulting Pareto front are given. Right: Sort and Spec. Sort show the full running times of the basic sorted (S1, $\mathbf{pf}_{lex}$) and the specialized sorted (S7, $\mathbf{pf}_{lex}$) case respectively.

Most noticeably, comparing algorithms S2-7 to S1, a speed up over S1 could be achieved for most test cases. Exceptions are Ali3D and Prob2D, both of which have a very low overall computation time, and Gotoh2D. In all cases, S7 (Merge In-Place) was the best performing algorithm. Also, in contrast to the randomized trials, no separator was needed to optimize runtimes. In all cases, S7 consistently scored better or equal to S1. S5 performed second without any exceptions in the tested sets. While S3 appeared promising in the randomized trials, only for Prob4D a separator could be found such that S3 could achieve a runtime gain, even though it was significantly lower than that of S7. Like in the randomized trials, S6 performed consistently worse than S5. S4 and S2 never achieved any gain. Comparing the estimated gains of the direct comparison of the sorting algorithms to the full ADP tests reveals no significant discrepancies.
The discrepancies between randomized and realistic test cases can be well explained by the nature of ADP and the non-random character of intermediate lists. Unlike in the random trials, lists become very long very quickly and long intermediate lists dominate the runtime, explaining the bad performance of S3. Separators for S7 become unneeded as on average more than three elements – as determined by the separator in the random trial – are generated per sublist. This is not unexpected as lists are created by combining subresults that themselves are likely to contain more than three elements, even for small inputs. This property can likely be reproduced for most other grammars that contain at least one rule generating sublists proportional in number to the input length, as is the case for the folding grammar.
The analysis, however, becomes more complicated when trying to explain the variation between the different ADP applications.
Neither input size nor front size seem to be a good indicator on which algorithm works best. Most noticeably, the largest gain over S1 was achieved with a final front size of 13 for Fold2D. For Ali3D, there was nearly no gain, despite a much larger final front size of 446. Prob4D and Ali3D were executed on similarly sized inputs, but only one could achieve a significant gain. The reason for these observations is that no direct statements can be made about intermediate list and front sized by input size or final front size alone.

Most informative is the layout of the search space. In Fig. 11, the scatter plots for Fold2D, showing the best improvement, and Gotoh2D, with no improvement, are presented. In its moderate computation time and large final front size, Gotoh2D shows only very limited variance for intermediate results. The final Pareto front consists mostly of co-optimals that are only added in the last step of the computation and are not present for the sorting phases. Comparing the full computation times of Gotoh2D, the overall time variance is minimal, which can be explained by the small intermediate lists. The ultimate reason for this is the problem decomposition of Gotoh2D. At most 3 sorted sublists are combined



(a)



(b)

Figure 11: Runtime gain of individual data points of S7 against S1 for the test cases a) Fold2D and b) Gotoh2D. Points are plotted when one algorithm performed better than the other with point sizes relative to the gained time. Gotoh2D shows only very little variance in list sizes contrary to Fold2D, explaining the performance differences.

Name	Input Size	Front Size	Gain (s)	Separator	$\mathbf{pf}_{lex}$ (s)	$\overset{p}{\vee}_{lex}$ (s)
Ali2D	509	487	220.46	-	579.77	**356.11**
Ali3D	200	446	-	-	**4.97**	8.04
Gotoh2D	249	209,744	-	-	29.42	29.83
Fold2D	608	13	762.16	-	2220.24	**1054.39**
Fold3D	404	165	-	-	**841.64**	5810.36
Prob2D	185	9	0	-	3.64	**2.76**
Prob4D	185	2327	-	-	**2556.83**	14,552.12

Table 6: Summary of computation times and meta-data of Pareto eager test cases. Left: Estimated time gains for $\overset{p}{\vee}_{lex}$ with S1 over $\mathbf{pf}_{lex}$. Input length and the final size of the resulting Pareto front are given. Right: Full runtimes are given.

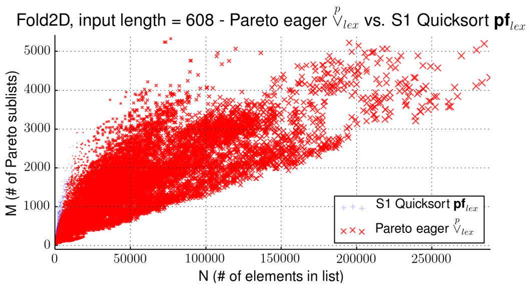
by the tree grammar, while the evaluation algebra also does not allow much variation. In contrast, the grammars for the other applications have at least one rule that generates sublists of length proportional to the input length. We will later revisit this analysis when comparing the runtime of all algorithms.

As a result of the benchmarks of this section, Bellman's GAP was set to use no separators as default. Should the need arise for separators for later applications, the integration into GAP-C is trivial by extending existing functionality. Calls to all sorting algorithms but S1, for the standard implementation, and S7, for lexicographically sorted implementation, have been removed. Their implementations remain in the header files, however, and they can be added manually.

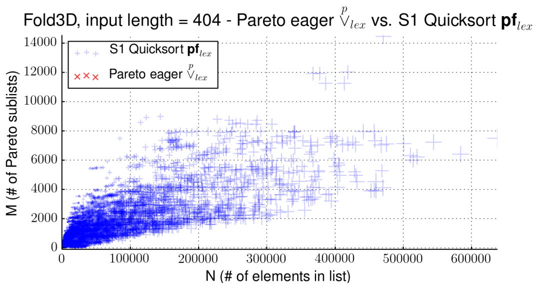
8.6 Pareto eager ADP

Much like for lexicographically sorted ADP, we want to test the behaviour of different Pareto eager implementations before comparing them to all other methods. The main goal is to determine possible separators, indicating when one algorithm performs better than the other, between the standard implementation with $\mathbf{pf}_{lex}$ and the Pareto eager implementation with $\overset{p}{\vee}_{lex}$. We have already seen the poor performance of $\overset{p}{\vee}_{yuk}$ compared to $\mathbf{pf}_{yuk}$ in Section 8.4, hence no separators can be found between both methods. We will also keep the analysis of $\overset{p}{\vee}_{nosort}$ and $\mathbf{pf}_{nosort}$ to the next section, as they represent the same algorithm – only called on differently sized lists – and therefore no useful separators are likely to exist.

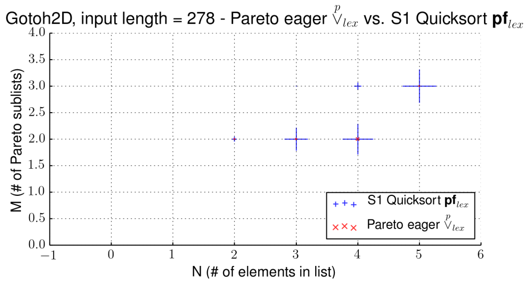
As randomized uniform Pareto sets are hard to construct, we will restrict analysis to the



(a)



(b)



(c)

Figure 12: Runtime gain of individual data points of $\mathbf{pf}_{lex}$ against $\overset{p}{\vee}_{lex}$ for the test cases a) Fold2D, b) Fold3D and c) Gotoh2D. Points are plotted when one algorithm performed better than the other with point sizes relative to the gained time. For the two-dimensional Fold2D, $\overset{p}{\vee}_{lex}$ performs best. For the three-dimensional Fold3D, $\mathbf{pf}_{lex}$ always performed better. For Gotoh, the plot shows only limited variation and no winning algorithm.

ADP applications only. Again, the tests were executed in 2 steps, first comparing the runtimes at the level of intermediate subproblems, excluding the runtime of the surrounding code, afterwards comparing the full running times of the best implementations. As separators, linear and exponential separation based on values of N and M are considered, motivated by the asymptotic complexity of the compared implementations that differ in factors of these two variables. A summary of the results is shown in Table 6.
Looking at the data, one may notice two things. First of all, no separators are given for any case. The reason for this is that in no case any improving separators could be found. In all cases, one algorithm performs best for all data points. Secondly, with the exception of Gotoh2D, for two-dimensional definitions $\overset{p}{\vee}_{lex}$ always performed best, whereas for higher dimensions, $\mathbf{pf}_{lex}$ is superior without fail. The explanations for both are not entirely unrelated. The difference between dimensions is of course a result of the different underlying product definitions.

The two-dimensional case is very similar to the sorted case, only that we apply Pareto operators as well as sorting the elements. In a way, $\overset{p}{\vee}_{lex}$ works similar to a normal two way merge sort with a merge step definition that is not in place, reflected by the complexity of $O(N\log(M))$. By the same arguments as before, $\overset{p}{\vee}_{lex}$ is likely to perform better than executing a sorting in $O(N\log(N))$ when $M << N$. Reversely, if the number of sublists M is near the number of elements N, and sublists are very short, $\mathbf{pf}_{lex}$ will perform better. As intermediate lists are created by combining subresults, the length of individual subresults are likely to stay over a certain threshold that seems to be high enough in order for $\overset{p}{\vee}_{lex}$ to perform well.
For higher dimensions, the situation changes, as now both algorithms perform in $O(N^2)$, the complexity of the Pareto operations now dominating the sorting aspects of both imple-

	Mode	Alg.	Sort Alg.	Complexity	Move	Comperator
Standard		$\mathbf{pf}_{nosort}$	-	$O(N^2)$	no	-
		$\mathbf{pf}_{lex}$ 2D	S1	$O(N + N\log(N))$	no	-
		$\mathbf{pf}_{lex}$ 3D+	S1	$O(N^2 + N\log(N))$	no	-
		$\mathbf{pf}_{isort}$	-	$O(N^2)$	no	-
		$\mathbf{pf}_{yuk}$	-	$O(N\log^{d-1}(N))$	partial	one dim., all dim.
Sort.	**Block**	$\mathbf{pf}_{nosort}$	S7	$O(N^2 + N\log(M))$	only in sort	all dim.
		$\mathbf{pf}_{lex}$ 2D	S7	$O(N + N\log(M))$	only in sort	all dim.
		$\mathbf{pf}_{lex}$ 3D+	S7	$O(N^2 + N\log(M))$	only in sort	all dim.
Pareto eager	**Block**	$\overset{p}{\vee}_{nosort}$	-	$O(N^2)$	yes	one dim.
		$\overset{p}{\vee}_{lex}$ 2D	-	$O(N\log(M))$	yes	one dim.
		$\overset{p}{\vee}_{lex}$ 3D+	-	$O(N^2)$	yes	one dim., all dim.
		$\overset{p}{\vee}_{yuk}$	-	$O(N\log^{d-1}(N)\log(M))$	yes	one dim., all dim.
	Step	$\overset{p}{\vee}_{nosort}$	-	$O(N^2)$	yes	one dim.

Table 7: Summary of all implemented methods for computing Pareto fronts. Algorithms are classified by implementation (standard implementation, lexicographically sorted ADP, Pareto eager ADP) and mode (step, block). Move indicates whether the C++ 11 function move can be used to avoid copying elements. Comparators are generated either for testing all dimensions in one call (all dim.), or to test each dimension individually (one dim.). $\overset{p}{\vee}_{nosort}$ is the only Pareto operator that does not create new lists for results.

mentations. However, the additional factors brought in by the different scenarios strongly favour the use of S1 and a single operator step, compared to multiple smaller operator steps, each in $O(N^2)$. This seems to be true already for small N and accordingly, only grows worse for longer candidate lists. The scatter plots in Figs. 12a and 12b confirm the theories for both cases.
Like in the sorted scenario, Gotoh2D shows hardly any variation in the computation times. Of course, the same rationale as before applies to explain this behaviour. As we can see in Fig. 12c, due to the Pareto condition, intermediate lists are even shorter and allow even fewer variations and with even less potential for better performing algorithms.

Following the results of this section, no separators will be generated within Bellman's GAP.

8.7 Putting it all together

Now that all individual components have been established, it is time to execute a final benchmark that compares all competing scenarios. To ease the reader into the analysis, a summary of all implementations with their most important properties are presented in Table 7.
For this test, only full runtimes were measured, including all recursions of the ADP programs. A summary of all runtimes over all applications is described in Table 8. We will analyse the data in multiple steps, referencing results from previous subsections as needed.

The most noticeable observation that can be made from the data is that $\mathbf{pf}_{nosort}$ performs best for most cases, both for two-dimensional and higher-dimensional cases. This replicates the results of the preliminary implementations done in [13]. It is likely that this good behaviour can be attributed to a positive randomization effect. When operating on sorted lists, extremal points of the Pareto fronts that are maximal in one dimension, but minimal in others, will be tested first. Such a point is unlikely to create domination over a new element, as it dominates only a rather small volume in the system of possible val-

Name	Input	Front	Standard ADP				Sorted ADP		Pareto eager ADP			
							Block		Block			Step
			pf$_{nosort}$	**S1 pf**$_{lex}$	**pf**$_{isort}$	**pf**$_{yuk}$	**S7 pf**$_{nosort}$	**S7 pf**$_{lex}$	$\overset{p}{\vee}_{nosort}$	$\overset{p}{\vee}_{lex}$	$\overset{p}{\vee}_{yuk}$	$\overset{p}{\vee}_{nosort}$
Ali2D	100	51	**0.39**	0.43	0.41	-	0.53	0.44	0.57	0.42	-	0.42
	200	44	0.5	0.5	0.51	-	0.56	0.52	0.59	0.55	-	**0.48**
	302	125	71.1	76.4	103.6	-	131.4	74.7	147.9	**59.46**	-	82.1
	404	121	6.17	5.73	9.06	-	10.59	5.24	11.63	**4.53**	-	6.23
	509	487	667.84	579.77	1320.43	-	1647.64	529.98	1720.65	**356.11**	-	775.77
Ali3D	100	176	**3.92**	4.33	-	4.72	4.78	3.95	11.74	11.94	15.27	4.56
	200	446	5.43	4.97	-	4.93	5.59	4.88	6.48	8.04	8.09	**4.85**
Gotoh2D	249	209,744	29.19	29.42	29.87	-	29.77	30.32	**28.75**	29.83	-	**28.75**
Fold2D	100	1	0.43	0.47	**0.42**	-	0.5	0.49	0.53	0.52	-	0.43
	200	7	**3.19**	3.75	3.31	-	4.1	3.93	4.42	3.81	-	3.39
	302	2	**17.0**	20.77	18.29	-	23.08	21.93	26.57	21.06	-	18.43
	404	20	**36.98**	50.83	38.95	-	52.25	49.53	68.72	46.7	-	40.07
	509	14	**154.12**	334.53	164.55	-	335.23	312.28	614.09	216.54	-	228.29
	608	13	**631.43**	2220.24	705.97	-	1940.5	1833.31	6041.04	1054.39	-	1279.03
Fold3D	100	6	**0.5**	0.65	-	0.84	0.66	0.65	1.08	1.12	1.52	0.56
	200	88	**5.92**	10.29	-	11.69	10.38	9.96	24.16	24.3	30.03	8.97
	302	181	**91.88**	186.16	-	196.21	185.67	180.51	829.8	796.33	814.9	174.7
	404	165	**314.52**	841.64	-	704.03	868.05	806.67	5816.37	5810.36	4583.56	813.2
Prob2D	96	16	**0.09**	0.12	**0.09**	-	0.16	0.15	0.16	0.15	-	0.1
	185	8	**2.05**	3.64	2.2	-	4.08	3.6	4.4	2.76	-	2.36
Prob4D	96	183	**0.82**	1.08	-	1.48	0.94	0.94	1.69	2.37	4.09	0.94
	185	2327	1774.88	2556.83	-	**1227.13**	2469.8	2449.83	9990.08	14,552.12	13,814.64	2535.11

Table 8: Summary of benchmarks for all applications. A representative set of inputs were tested for each test case. Algorithms are classified by implementation (standard implementation, lexicographically sorted ADP, Pareto eager ADP) and mode (step, block). No implementation performed best for all tests. **pf**$_{nosort}$ performed best in most cases. For Ali2D, $\overset{p}{\vee}_{lex}$ consistently achieved the shortest runtimes. For Prob4D, **pf**$_{yuk}$ is the fastest implementation.

ues. On unsorted lists, non-extremal points are on average encountered earlier and thus, domination can be established earlier in the computations. A good indicator for this can be found in the data when comparing the tests of $\mathbf{pf}_{nosort}$ and $\mathbf{pf}_{lex}$ for the sorted implementation. For both tests, the same sorting mechanism was used and hence, the times for sorting are the same. For no test under these conditions, $\mathbf{pf}_{nosort}$ performed better than $\mathbf{pf}_{lex}$. Without utilizing the randomized effect, $\mathbf{pf}_{nosort}$ becomes nothing more than a less effective implementation of $\mathbf{pf}_{lex}$, testing unnecessary conditions, regardless of the number of dimensions. As another possible factor, of course, $\mathbf{pf}_{nosort}$ benefits from not having to rearrange all elements in a sorting, especially for strongly unsorted candidate lists.
The use of $\mathbf{pf}_{nosort}$ in sorted ADP, as well as the implementations of $\overset{p}{\vee}_{nosort}$ for Pareto eager ADP in both modes, show no significant improvement. In contrast, except for two cases with very small overall computation times – where step-wise $\overset{p}{\vee}_{nosort}$ was best – and Gotoh2D, all other uses show a significant increase in running times. Block mode $\overset{p}{\vee}_{nosort}$ is always drastically slower than using the step mode, as was expected by the overhead of the two-way merge structure. The bad performance in the sorted case could already be explained in the last paragraph. The reason why step-wise merge performs worse than in the standard implementation is not directly apparent. At least in theory, the Pareto eager implementation should benefit from three factors:

- candidates do not need to be copied
- omission/integration of the lexicographic product as an optimization
- candidate lists are reduced in size early on

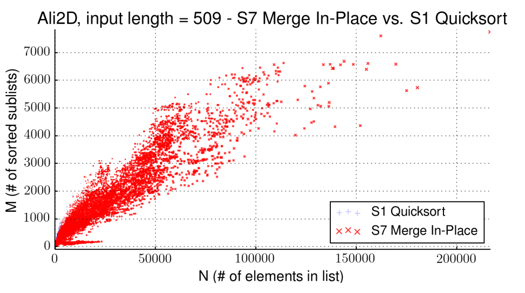
These, however, seem to be counterbalanced by:

- overhead of additional function calls
- overhead of the use of comparators
- increased level of false branch predictions due to breaking up the work of one loop into multiple calls

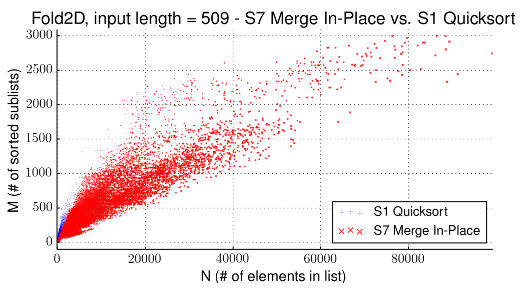
Only the second problem could be fully removed when creating inlined code instead of using the template interface of the current implementation.

As expected by its bad complexity, $\overset{p}{\vee}_{yuk}$ could not perform well in any case, in fact increasing runtimes by a factor of more than 10 for all cases. While not as dramatic in outcome, $\mathbf{pf}_{isort}$ also never performs better than $\mathbf{pf}_{nosort}$ and can therefore be discarded. The reason for this can likely be found in the loss of the randomization effect due to sorting the list, as well as the use of expensive insertion operations.

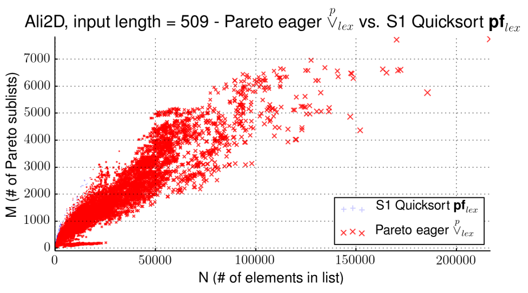
This leaves open $\mathbf{pf}_{yuk}$, the sorted implementations as well as the Pareto eager implementation using $\overset{p}{\vee}_{lex}$, all of which have a specific use-case.
Contrasting to most tested applications, for Ali2D and Ali3D, $\mathbf{pf}_{nosort}$ could not perform best. For Ali2D, instead $\overset{p}{\vee}_{lex}$ was consistently better. At the same time, $\mathbf{pf}_{lex}$, for both the sorted as well as the standard implementation, performed better than $\mathbf{pf}_{nosort}$ as well. This is a unique case within all other test cases. The two facts are not without a deeper connection, however. We already saw that for all cases, the sorted implementation with S7 can improve the runtime compared to the standard implementation with S1. In fact, the strongest improvement can be found in Fold2D. Yet, for no application the increase was enough to overtake the standard $\mathbf{pf}_{nosort}$. Only for Ali, where $\mathbf{pf}_{lex}$ already performs better than $\mathbf{pf}_{nosort}$, the sorted implementation can increase the overall implementation time even more. Here, however, then the two-dimensional case Pareto eager $\overset{p}{\vee}_{lex}$ becomes



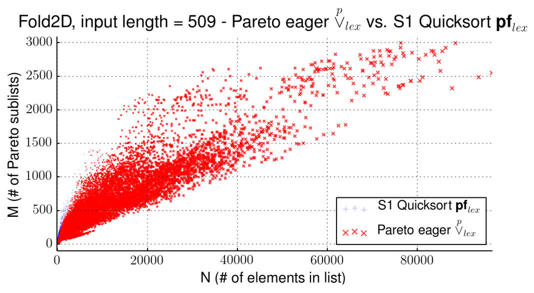
(a)



(b)



(c)



(d)

Figure 13: Runtime gain of individual data points for Top) S7 against S1 and Bottom) $\mathbf{pf}_{lex}$ against $\overset{p}{\vee}_{lex}$ for the test case a, c) Ali2D and b, d) Fold2D. Points are plotted when one algorithm performed better than the other with point sizes relative to the gained time. Both algorithms show similar distributions as they were defined over the same grammar using the same input.

the best implementation.
The reason for this is that $\mathbf{pf}_{lex}$ and $\overset{p}{\vee}_{lex}$ profit from the same properties of the search space that allow an effective merge of presorted lists. At least in this case, the Pareto eager implementation seems to benefit the strongest, although sorted ADP and Pareto eager ADP have similar advantages and disadvantages. The early reduction of candidate lists is the most likely reason for the good performance of $\overset{p}{\vee}_{lex}$.
This does not mean that sorted ADP is without merit, however. Although step-wise Pareto eager $\mathbf{pf}_{nosort}$ performed best for Ali3D, the good effect of the sorted implementation compared to the standard $\mathbf{pf}_{nosort}$ can be seen here as well. Due to the increased complexity, $\overset{p}{\vee}_{lex}$ no longer is a well-performing candidate. For larger inputs, sorted ADP with $\mathbf{pf}_{lex}$ can be expected to be the best option for Ali3D and similar cases. Likewise, if products are needed for the two-dimensional case that are currently not supported by Bellman's GAP in Pareto eager implementations, the sorted implementation can serve as a fall-back option.

While explaining the relation between the different algorithms fairly well, the analysis, so far, lacks an explanation of why only Ali behaves differently than all other implementations. For this, it is time to return to the layout of the search space. In Fig. 13, the plots of intermediate lists for the sorted and the Pareto eager implementations are shown. Sorted and Pareto eager plots show the same data, only for different time gains. The plots are created against the same base, and the same scaling factor was used to compute the size of each marker. Therefore, all plots are inter-comparable. The time gains for the Pareto eager implementation are visibly greater than those of the sorted implementation, as is the case for all two-dimensional products.
Ali2D and Fold2D were computed out of the same data and use the same tree grammar, employing the same number of evaluation algebras. The difference in the distribution

of data points is therefore caused by the qualitative influence of the evaluation algebras. Fold2D generates smaller lists, with, in relation, fewer sublists compared to Ali2D. This alone, however, can not explain why for Ali2D consistently $\overset{p}{\vee}_{lex}$ performs best, while for Fold2D, the standard unsorted case is considerably faster. This discrepancy most likely can be related to the order and sortedness in which candidates are created, sortedness addressing how many elements are inverted in the lists upon creation.

In a way, one can hypothesise a dichotomy between the sortedness and competing unsortedness of generated candidate lists, $\overset{p}{\vee}_{lex}$ and $\mathbf{pf}_{nosort}$ standing at the ends of this spectrum. The more initially sorted a candidate list is, the more effective $\overset{p}{\vee}_{lex}$ and $\mathbf{pf}_{lex}$ can work, while the randomization effect for $\mathbf{pf}_{nosort}$ is at the weakest. With decreasing sortedness the situation tilts in the direction of $\mathbf{pf}_{nosort}$. To fully substantiate this theory a close analysis of intermediate lists should be executed.

The strong effect of different grammar definitions becomes apparent when taking another look at Gotoh2D. We already saw in the last section that Gotoh2D varies strongly from all other cases that were defined over the same grammar, limiting the variation within intermediate candidate lists (see Fig. 11b and 12c). This reflects on the very limited time difference for not just the sorted and Pareto eager implementation, but all tested algorithms.

As a last case, we need to take a closer look at Prob4D. Of all applications, Prob4D was the only case to define a Pareto product over four dimensions and because of this, it is also exhibiting very long intermediate lists. As such, Prob4D was the only case where the superior asymptotic complexity of the unsorted case with $\mathbf{pf}_{yuk}$ resulted in an actual performance increase. It is interesting to note that Fold3D also exhibited similarly-sized intermediate lists for the largest input, but instead of improving runtimes, $\mathbf{pf}_{yuk}$ doubled the effort. In its complicated definition, $\mathbf{pf}_{yuk}$ employs high internal factors for runtime that can only be overcome for large inputs. Higher-dimensional products are more likely to fulfil this property. It should also be noted that – similar to $\overset{p}{\vee}_{lex}$ and $\mathbf{pf}_{lex}$ – the effectiveness of $\mathbf{pf}_{yuk}$ depends on how well each dimension can be partitioned, in other words, how well the search space can be sorted. Therefore, applications that can benefit from the unsortedness the most with $\mathbf{pf}_{nosort}$, might as well show a disadvantage of $\mathbf{pf}_{yuk}$.

All tested implementations will be kept in the Bellman's GAP system. The optimal choice is left to the user. Based on the results of this section, users are advised to follow the following rules when creating new applications with Pareto optimization in Bellman's GAP:

- $\mathbf{pf}_{nosort}$ should be considered first for all implementations, as it performs best in most cases, and the formulations of standard ADP allow the greatest flexibility.
- If $\mathbf{pf}_{nosort}$ does not perform fast enough and a two-dimensional Pareto front is employed, Pareto eager $\overset{p}{\vee}_{lex}$ or sorted ADP $\mathbf{pf}_{lex}$ should be tried, depending on the surrounding products.
- If $\mathbf{pf}_{nosort}$ does not perform fast enough and a higher-dimensional Pareto front is used, $\mathbf{pf}_{yuk}$ or sorted ADP $\mathbf{pf}_{lex}$ should be tested.
- When using $\mathbf{pf}_{yuk}$ and the time is critical, the cut-off value should be optimized for the application.

9 Conclusion and Outlook

9.1 Conclusion

The contributions of this work are manifold. Bellman's GAP allows the deep integration of Pareto front operators for the use with arbitrary products. For the first time, a deep integration of Pareto into an ADP framework was realized, replacing existing specific or only superficial implementations in dynamic programming. This facilitated the creation and analysis of a collection of fully functional, entirely automatically generated Pareto front operators that could be benchmarked against each other, creating novel insights on their relations. It is now possible for users to optimize their applications by choosing from a pool of ready to use, efficient implementations.

In this context, a systematic interface for changes to the standard implementation of ADP has been proposed and realized for the Bellman's GAP system. For this generalized implementation major changes have been introduced to the framework. This novel concept allowed the implementation of previously untested algorithms. Using the interfaces created by the generalized implementation, new ADP definitions were implemented for the use with Pareto optimization.

In lexicographically sorted ADP, intermediate candidate lists are kept sorted at all times. Six algorithms with similar complexity but different properties that can exploit this property for sorting were implemented and tested. The results were compared against a standard Quicksort implementation. An in place two way merge strategy we call *S7 Merge In-Place* could be established that performs better than all other sorting algorithms, both for random as well as realistic datasets. For the use with sorted ADP, the Pareto front operator $\mathbf{pf}_{lex}$ could be established as the best choice. If applicable, sorted ADP should always be preferred to sorting candidates in standard ADP.

In Pareto eager ADP, intermediate candidate lists are always kept as Pareto fronts. For this, several new Pareto merge operators have been introduced and implemented, reinterpreting strategies of Pareto front operators used by the standard implementation of ADP. The operator of $\overset{p}{\vee}_{lex}$ for two-dimensional products could be established as the only effective implementation.

For the standard implementation of ADP, a new algorithm $\mathbf{pf}_{yuk}$ was introduced and optimized that performs well for Pareto fronts with more than two dimensions.

Among other changes, a series of new general concepts, such as comparators and multi-dimensional products, have been created for future and current use. To efficiently handle floating point accuracy throughout all implementations, a central comparison framework was created for this purpose.

All implementations were tested under different real world applications of biosequence analysis. We could show that a "naive" implementation using $\mathbf{pf}_{nosort}$ in standard ADP performs best in most cases. At the same time, we could show strong variations between different applications of ADP, sometimes favouring Pareto eager $\overset{p}{\vee}_{lex}$, if candidate lists profit from sorting, sometimes favouring no specific implementations. $\mathbf{pf}_{yuk}$ was shown to perform well for long candidate lists on high-dimensional products. The effects of the layout of the search space responsible for these differences have been highlighted. A guideline for optimizations was established.

For users of ADP, the use of Pareto optimization is now literally just a single keystroke away. All implementations of Pareto operators created in this work can be easily accessed using command line parameters. Although some optimizations have been left open, the implementation of Pareto products in Bellman's GAP can be regarded to be fully functional. This stands as a strong propositions for the use of Pareto optimization with ADP. The use in Bellman's GAP is both simple and effective, regardless of modifying old or creating new dynamic programming applications.

9.2 Outlook

To address the future of Pareto optimization in Bellman's GAP, a few limitations of the current implementations should be highlighted.

The biggest drawback of this work poses the restricted support of certain algebra product definitions for sorted and Pareto eager implementations. We could demonstrate problematic formulations like classified dynamic programming, at the same time highlighting general strategies on how to solve this problem in future. With the great performance of $\mathbf{pf}_{nosort}$, however, there is no immediate need for extending both strategies and further changes should only be attempted if absolutely needed. Nonetheless, a full reimplementation of the hashtable design of Bellman's GAP is recommended, taking into account candidate lists of arbitrary lengths and the requirements of the new generalized interface.

As a second big limitation, we saw that front sizes quickly become too big to compute higher-dimensional products. Although this problem cannot be fully amended, we propose the use of parallelization to lessen the impact. So far, only limited support exists for parallelizing the recursions of the generated ADP programs, i.e. the iteration through the search space. This will likely not change in near future due to the high complexity of the programs. Instead, we suggest to only parallelize Pareto front operators. For $\mathbf{pf}_{yuk}$, simple and intuitive strategies could be devised along its divide and conquer formulation.

So far, the generalized implementation of ADP was not fully disclosed to the users of the framework. New implementations of ADP are also likely to invoke new code changes to GAP-C. As such, the interfaces serve more as a guideline for future developers than for the extension by end users. Although unlikely, should the need arise for this in future, the interfaces could be opened by giving the users a handful of special generation options.

Both sorted ADP $\mathbf{pf}_{lex}$ and standard ADP $\mathbf{pf}_{yuk}$ showed strong promise for special applications. Should the need for even faster strategies arise in future, so far untested optimizations could be attempted. Examples for this are the use of a k-way merge [44] instead of a fixed binary strategy, and the use of different median strategies for $\mathbf{pf}_{yuk}$, respectively.

As a last further optimization, motivated by the superior behaviour of $\mathbf{pf}_{nosort}$, new implementation strategies can be devised that try to maximize the positive effect of the randomizations. One such algorithm would sort the candidate lists prior to calling $\mathbf{pf}_{nosort}$ by the volume they span in the search space, therefore relative by how likely they are to dominate another candidate. Candidates that can likely dominate other candidates will hence be processed guaranteed as early as possible.

It should be noted that this work was only concerned with the computation of Pareto fronts, not, however, with the analysis and correctness – in the sense that they produce meaningful results – of them. The analysis of especially large Pareto fronts remains an open problem that will be addressed in so far unpulished work.

Even after all the work done on this thesis, Pareto optimization never ceases to amaze with its abundance of competing implementation strategies. The observations of this work are expected to be also valid for the framework of inverse coupled rewrite systems [45], which generalizes ADP to tree-structured data. A proof for this is open, however.

Many applications that could benefit from the use of Pareto optimization remain yet to be re-evaluated.

References

[1] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd ed., 2001.

[2] R. Bellman, *Dynamic Programming*. Princeton, NJ, USA: Princeton University Press, 1 ed., 1957.

[3] R. Giegerich, C. Meyer, and P. Steffen, "A discipline of dynamic programming over sequence data," *Science of Computer Programming*, vol. 51, no. 3, pp. 215–263, 2004.

[4] R. Giegerich and P. Steffen, "Implementing Algebraic Dynamic Programming in the Functional and the Imperative Programming Paradigm," in *Mathematics of Program Construction* (E. Boiten and B. Möller, eds.), vol. 2386 of *Lecture Notes in Computer Science*, pp. 1–20, Springer Berlin Heidelberg, 2002.

[5] P. Steffen, *Compiling a domain specific language for dynamic programming*. PhD thesis, Bielefeld University, 2006.

[6] G. Sauthoff, *Bellman's GAP: A 2nd Generation Language and System for Algebraic Dynamic Programming*. PhD thesis, Bielefeld University, 2011.

[7] G. Sauthoff and R. Giegerich, "Yield grammar analysis and product optimization in a domain-specific language for dynamic programming," *Science of Computer Programming*, vol. 87, July 2014.

[8] G. Sauthoff, M. Möhl, S. Janssen, and R. Giegerich, "Bellman's GAP – a Language and Compiler for Dynamic Programming in Sequence Analysis," *Bioinformatics*, vol. 29, no. 5, pp. 551–560, 2013.

[9] C. H. zu Siederdissen, "Sneaking Around concatMap: Efficient Combinators for Dynamic Programming," in *Proceedings of the 17th ACM SIGPLAN international conference on Functional programming*, ICFP '12, (New York, NY, USA), pp. 215–226, ACM, 2012.

[10] P. Steffen and R. Giegerich, "Versatile and declarative dynamic programming using pair algebras," *BMC Bioinformatics*, vol. 6, no. 1, p. 224, 2005.

[11] B. Voß, R. Giegerich, and M. Rehmsmeier, "Complete probabilistic analysis of RNA shapes," *BMC Biology*, vol. 4, no. 1, 2006.

[12] J. Branke, K. Deb, K. Miettinen, and R. Slowinski, eds., *Multiobjective Optimization, Interactive and Evolutionary Approaches [outcome of Dagstuhl seminars]*, vol. 5252 of *Lecture Notes in Computer Science*, Springer, 2008.

[13] C. Saule and R. Giegerich, "Pareto optimization in algebraic dynamic programming," *Algorithms for Molecular Biology*, vol. 10, p. 22, 2015.

[14] R. Durbin, S. R. Eddy, A. Krogh, and G. Mitchison, *Biological sequence analysis*. Cambridge University Press, 1998.

[15] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of Molecular Biology*, vol. 48, pp. 443–453, Mar. 1970.

[16] O. Gotoh, “An Improved Algorithm for Matching Biological Sequences,” *Journal of Molecular Biology*, vol. 162, pp. 705–708, 1982.

[17] S. Janssen and R. Giegerich, “The RNA shapes studio,” *Bioinformatics*, vol. 31, no. 3, pp. 423–425, 2015.

[18] I. L. Hofacker, S. H. F. Bernhart, and P. F. Stadler, “Alignment of RNA base pairing probability matrices,” *Bioinformatics*, vol. 20, no. 14, pp. 2222–2227, 2004.

[19] S. Bernhart, I. L. Hofacker, S. Will, A. Gruber, and P. F. Stadler, “RNAalifold: improved consensus structure prediction for RNA alignments,” *BMC Bioinformatics*, vol. 9, no. 1, p. 474, 2008.

[20] J. Reeder and R. Giegerich, “A Graphical Programming System for Molecular Motif Search,” in *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, GPCE ’06, (New York, NY, USA), pp. 131–140, ACM, 2006.

[21] 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*, vol. 34, no. 1, pp. 27–47, 2000.

[22] S. Sitarz, “Pareto optimal allocation and dynamic programming,” *Annals of Operational Research*, vol. 172, pp. 203–219, 2009.

[23] T. Schnattinger, U. Schoening, A. Marchfelder, and H. Kestler, “Structural RNA alignment by multi-objective optimization,” *Bioinformatics*, vol. 29, no. 13, pp. 1607–1613, 2013.

[24] T. Schnattinger, U. Schoening, A. Marchfelder, and H. Kestler, “RNA-Pareto: interactive analysis of Pareto-optimal RNA sequence-structure alignments,” *Bioinformatics*, vol. 29, no. 23, pp. 3102–3104, 2013.

[25] R. Libeskind-Hadas, Y.-C. Wu, M. Bansal, and M. Kellis, “Pareto-optimal phylogenetic tree reconciliation,” *Bioinformatics*, vol. 30, no. 12, pp. i87–i95, 2014.

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

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

[28] A. Fog, “Optimizing software in C++: An optimization guide for Windows, Linux and Mac platforms,” 2004.

[29] J. L. Bentley, “Multidimensional Divide-and-conquer,” *Commun. ACM*, vol. 23, pp. 214–229, Apr. 1980.

[30] R. Sedgewick, “Implementing Quicksort Programs,” *Commun. ACM*, vol. 21, pp. 847–857, Oct. 1978.

[31] G. S. Brodal, R. Fagerberg, and G. Moruz, “On the Adaptiveness of Quicksort,” *J. Exp. Algorithmics*, vol. 12, pp. 3.2:1–3.2:20, Aug. 2008.

[32] K. Dudzinski and A. Dydek, "On a Stable Storage Merging Algorithm," in *Information Processing Letters 12*, pp. 5–8, 1981.

[33] E. P. Nawrocki, S. W. Burge, A. Bateman, J. Daub, R. Y. Eberhardt, S. R. Eddy, E. W. Floden, P. P. Gardner, T. A. Jones, J. Tate, and R. D. Finn, "Rfam 12.0: updates to the RNA families database," *Nucleic Acids Research*, vol. 43, no. D1, pp. D130–D137, 2015.

[34] J. D. Thompson, P. Koehl, and O. Poch, "BAliBASE 3.0: latest developments of the multiple sequence alignment benchmark," *Proteins*, vol. 61, pp. 127–136, 2005.

[35] P. Cordero, J. B. Lucks, and R. Das, "An RNA Mapping DataBase for curating RNA structure mapping experiments," *Bioinformatics*, vol. 28, no. 22, pp. 3006–3008, 2012.

[36] S. Janssen, G. Schudoma, Christian Steger, and R. Giegerich, "Lost in folding space? Comparing four variants of the thermodynamic model for RNA secondary structure prediction.," *BMC Bioinformatics*, vol. 429, no. 12, 2011.

[37] T. Xia, J. J. SantaLucia, M. E. Burkard, R. Kierzek, S. J. Schroeder, X. Jiao, C. Cox, and D. H. Turner, "Thermodynamic parameters for an expanded nearest-neighbor model for formation of RNA duplexes with Watson-Crick base pairs," *Biochemistry*, vol. 37, pp. 14719–14735, 1998.

[38] S. Mortimer, C. Trapnell, S. Aviran, L. Pachter, and J. Lucks, "SHAPE-Seq: High-Throughput RNA Structure Analysis," *Current Protocols in Chemical Biology*, vol. 4, no. 4, pp. 275–297, 2012.

[39] D. Loughrey, K. E. Watters, S. A. H., and L. J. B., "SHAPE-Seq 2.0: systematic optimization and extension of high-throughput chemical probing of RNA secondary structure with next generation sequencing," *Nucleic Acid Research*, vol. 42, no. 21, 2014.

[40] W. A. Ziehler and D. R. Engelke, "Probing RNA Structure with Chemical Reagents and Enzymes," *Current Protocols in Nucleic Acid Chemistry*, 2001.

[41] S. E. Wells, J. M. Hughes, A. H. Igel, and M. J. Ares, "Use of dimethyl sulfate to probe RNA structure *in vivo*," *Methods Enzymology*, vol. 318, pp. 479–493, 2000.

[42] J. Talkish, G. May, Y. Lin, J. L. Woolford Jr., and C. J. McManus, "Mod-seq: high-throughput sequencing for chemical probing of RNA structure," *RNA*, vol. 20, pp. 713–720, 2014.

[43] S. Janssen and R. Giegerich, "Ambivalent covariance models," *BMC Bioinformatics*, vol. 178, no. 16, 2015.

[44] W. Greene, "k-way merging and k-ary sorts," in *Applied Computing, 1991., [Proceedings of the 1991] Symposium on*, pp. 197–, Apr 1991.

[45] R. Giegerich and H. Touzet, "Modeling Dynamic Programming Problems over Sequences and Trees with Inverse Coupled Rewrite Systems," *Algorithms*, vol. 7, pp. 62–144, 2014.

A Summary of New GAP-C Options

All new options of the Bellman's GAP compiler are listed in the following table:

Option	**Description**
-P [--pareto-version] arg	The Pareto front operator to use for computations: • 0: $\mathbf{pf}_{nosort}$ (Standard) / NOSORT • 1: $\mathbf{pf}_{lex}$ with S1 Quicksort / SORT(1) • 2: $\mathbf{pf}_{isort}$ • 3: $\mathbf{pf}_{yuk}$ / NOSORT(B)
--no-coopt-class	with backtrace, only print one candidate per element of the front
--multi-dim-pareto	set this token if a Pareto product with more than two dimensions is defined
-c [--cut-off] arg	set the cut-off size of $\mathbf{pf}_{yuk}$, if unset 65 is used
-S [--specialized-adp] arg	The ADP implementation to use: • 0: Standard ADP (Standard) • 1: Sorted ADP with S7 Merge In-Place / SORT(7) • 2: Pareto eager ADP If Pareto eager ADP is chosen, the meaning of -P is modified: • 0: $\overset{p}{\vee}_{nosort}$ / EAGER(unsorted) • 1: $\overset{p}{\vee}_{lex}$ / EAGER(sorted) • 2: - • 3: $\overset{p}{\vee}_{yuk}$
--step-mode arg	Set the mode of the implementation if -S > 0: • 0: block mode (standard) • 1: step mode
-f [--float-accuracy] arg	the number of decimal places to use for comparing floating points

B Algorithms

B.1 Yukish and Bentley

Algorithm B.1 marry$_{2D}$

```
INPUT: unsorted Pareto fronts $X$ and $Y$, $Y$ superior to $X$
OUTPUT: $X$ without elements dominated from $Y$ (in $X'$)
  sort $X$ and $Y$
  $X' \leftarrow \epsilon$
  $i \leftarrow 0$
  $j \leftarrow 0$
  $(y_1, y_2) \leftarrow Y[0]$
  while $i <$ length($X$) do                                  ▷ Compare 1st Dimension
      $(x_1, x_2) \leftarrow X[i]$
      if $y_1 \geq x_1$ then
          break
      $X' \leftarrow X'{:}(x_1, x_2)$                                          ▷ Add
      $i \leftarrow i + 1$
  $r \leftarrow 0$
  while $i <$ length($X$) do                                  ▷ Compare 2nd Dimension
      $(r_1, r_2) \leftarrow Y[r]$
      $(y_1, y_2) \leftarrow Y[j+1]$
      $(x_1, x_2) \leftarrow X[i]$
      if $j + 1 <$ length($Y$) and ($y_1 > x_1$ or ($y_1 == x_1$ and $y_2 \geq x_2$)) then
          $j \leftarrow j + 1$
          if $y_2 \geq r_2$ then
              $r \leftarrow j$
      else
          if $x_2 > r_2$ then
              $X' \leftarrow X'{:}(x_1, x_2)$                                  ▷ Add
          $i \leftarrow i + 1$
  return $X'$
```

Algorithm B.2 marry$_{brute}$

```
INPUT: unsorted Pareto fronts $X$ and $Y$, $Y$ superior to $X$
OUTPUT: $X$ without elements dominated from $Y$ (in $X'$)
  $X' \leftarrow \epsilon$
  for all $X_{\text{pre}}{:}(x_1, \ldots, x_k){:}X_{\text{suf}}$ do
      $add \leftarrow$ true
      for all $Y_{\text{pre}}{:}(y_1, \ldots, y_k){:}Y_{\text{suf}}$ do
          if $y_1 \geq x_1$ and ... and $y_k \geq x_k$ then                ▷ Lazy
              $add \leftarrow$ false
              break
      if $add$ then
          $X' \leftarrow X'{:}(x_1, \ldots, x_k)$                                ▷ Add
  return $X'$
```

B.2 Sorting

Algorithm B.3 S4 In-Join

```
INPUT: unsorted list $l$ of length $N$, array of list ends $e$ pointing to $M$ sublists,
  array of list starts in $s$
OUTPUT: sorted list in $l$
  $sub \leftarrow M - 1$                                         ▷ Lists to pass
  $queue \leftarrow$ empty sorted queue
  $it \leftarrow N - 1$                                          ▷ Global iterator
  while true (unsorted) do
      $i \leftarrow -1$                                          ▷ Next worst element
      for $j = 0; j \leq sub; j = j + 1$ do                      ▷ Find worst element
          if $e[j] \neq s[j]$ then
              if $i == -1$ then
                  $i \leftarrow j$
                  continue
              if $l[e[i]] > l[e[j]]$ then                        ▷ Full comparator
                  $i \leftarrow j$
      if $i == -1$ then                                          ▷ No sublists
          write queue to current positions $it$ and left of it
          return
      if $queue$ empty or $l[e[i]] <$ next element in $queue$ then
                                        ▷ List element worst, Full Comparator
          if $it \neq e[i]$ then
              if $e[sub] == it$ then                             ▷ Store displaced element
                  push $l[it]$ to $queue$
                  $e[sub] \leftarrow e[sub] - 1$
              $l[it] \leftarrow e[i]$
          $e[i] \leftarrow e[i] - 1$
      else                                                       ▷ Queue worst
          if $e[sub] == it$ then                                 ▷ Store displaced element
              push $l[it]$ to $queue$
              $e[sub] \leftarrow e[sub] - 1$
          $l[it] \leftarrow$ next element of $queue$
      $it \leftarrow it - 1$                                     ▷ Next elements
      if $it == s[sub]$ then
          $sub \leftarrow sub - 1$
      if $it < 0$ then
          return
```

Algorithm B.4 S5 Merge A, merge step

```
INPUT: unsorted list $x$ of length $l$, index of last element of first list $m$
OUTPUT: sorted list in $x$
  $it \leftarrow l - 1$                                  ▷ Global iterator
  $cl \leftarrow m$                                      ▷ Left iterator
  $q \leftarrow$ empty queue                             ▷ Temporary Queue
  while $m \neq it$ and $x[m] > x[it]$ do                ▷ Full comparator
    $it \leftarrow it - 1$                               ▷ Next element
  if $m == it$ then                                      ▷ End 1
    return
  push $l[it]$ to $q$
  $l[it] \leftarrow l[cl]$
  $it \leftarrow it - 1$
  $cl \leftarrow cl - 1$
  if $cl < 0$ then                                       ▷ End 2
    while $m \neq it$ do
      push $l[it]$ to $q$
      $l[it] \leftarrow$ next in $q$
      $it \leftarrow it - 1$
    $l[it] \leftarrow$ next in $q$
    return
                                                         ▷ $l[m]$ worse than $l[it]$
  while $m \neq it$ do
    if $l[cl] >$ next of $q$ then                        ▷ Full comparator
                                                         ▷ Next in $q$ worst
      push $l[it]$ to $q$
      $l[it] \leftarrow$ next in $q$
      $it \leftarrow it - 1$
    else                                                 ▷ Left iterator worst
      push $l[it]$ to $q$
      $l[it] \leftarrow l[cl]$
      $it \leftarrow it - 1$
      $cl \leftarrow cl - 1$
      if $cl < 0$ then                                   ▷ End 2
        while $m \neq it$ do
          push $l[it]$ to $q$
          $l[it] \leftarrow$ next in $q$
          $it \leftarrow it - 1$
        break
                                                         ▷ Right list sorted
  while $cl \geq 0$ and $q$ has element do
    if $l[cl] >$ next of $q$ then                        ▷ Full comparator
                                                         ▷ Next in $q$ worst
      $l[it] \leftarrow$ next in $q$
      $it \leftarrow it - 1$
    else                                                 ▷ Left iterator worst
      $l[it] \leftarrow l[cl]$
      $it \leftarrow it - 1$
      $cl \leftarrow cl - 1$
  write $q$ to $it$ and left of it
```

Declaration of Authorship

I hereby certify that this thesis has been composed by me and is based on my own work, unless stated otherwise. No other person's work has been used without due acknowledgement in this thesis. All references and verbatim extracts have been quoted, and all sources of information, including graphs and data sets, have been specifically acknowledged.

Bielefeld, November 10, 2015