A New Modular Parametric Search Framework

Christian Knauer* David Kübel† Fabian Stehn*

Abstract

Parametric search is a technique to develop efficient deterministic algorithms for optimization problems. The strategy requires an algorithm for the decision variant of the problem at hand. Given this decision algorithm, the strategy can be seen as a *black box* that does not really exploit characteristics of the underlying problem: It computes an *optimal* solution by keeping track of the values that appear in comparisons of the decision algorithm.

In this abstract, we present a new parametric search framework written in `Java`. We show how parametric search based algorithms can be implemented and explored with the framework. It allows to quickly specify or change crucial functionalities which influence the specific behaviour (and performance) of the resulting optimization algorithm. We focus on a transparent, simple-to-use, and modular design, and discuss the implementation of a specific algorithm that computes an optimal shortcut of a polygonal curve.

1 Introduction

Parametric search is a powerful and fairly general method to compute the (exact) optimal solution λ_{opt} of an optimization problem P in one variable. Let P be a minimization problem whose objective function is monotone, that is, for the decision variant DEC_P of P there is a value λ_{opt} such that $\text{DEC}_P(\lambda) = \text{TRUE} \Leftrightarrow \lambda \geq \lambda_{opt}$. In oder to apply Megiddo's [6] parametric search technique, two ingredients are required:

1. A sequential algorithm $\mathcal{A}_s$ that solves DEC_P for any value λ in T_s time, and
2. a parallel algorithm $\mathcal{A}_p$ using k processes, solving DEC_P for an unknown value of λ_{opt} in T_p time.

$\mathcal{A}_p$ is an algorithm that uses k (independent) processes whose control-flow depends on the outcome of comparisons. Each comparison can be resolved by relating a value that is derived from the input to the (unknown) value of λ_{opt}. The basic idea is to follow the control-flow of each process until it requires the solution to such a comparison and to collect these comparisons in *batches* (these collected values are usually called *critical values*). A batch hence consists of k values, representing k comparisons to λ_{opt}. Exploiting the monotonicity of P, all comparisons of a batch can be resolved by $O(\log k)$ calls to $\mathcal{A}_s$. After resolving a batch of critical values, each process can continue until a new batch of k comparisons is collected and resolved in the same way.

Throughout this process, an interval $I = (\lambda^-,\lambda^+]$ is maintained, where λ^- is the largest value encountered so far such that $\text{DEC}_P(\lambda^-) = \text{FALSE}$ and λ^+ is the smallest value encountered so far with $\text{DEC}_P(\lambda^+) = \text{TRUE}$; this implies that $\lambda_{opt} \in I$. Through the course of this process, at least one instance of the parallel algorithm $\mathcal{A}_p$ has to carry out a comparison with the actual optimal value λ_{opt}, which implies that λ_{opt} will appear as a critical value. Since P is a minimization problem, we have that $\lambda_{opt} = \lambda^+$ after all processes of $\mathcal{A}_p$ terminated. This gives a combined minimization algorithm $\mathcal{C}$ that computes λ_{opt} in $O(T_p \cdot (k + T_s \cdot \log k))$ deterministic time; see [6] for details.

1.1 Applications

The technique has been applied to a wide range of problems. In the field of computational geometry, e.g., the technique has been used to compute the Fréchet Distance [2] between two polygonal curves. Recently, Große et al. [4] showed how to efficiently compute a diameter-optimal shortcut between vertices of a polygonal curve. Agarwal et al. [1] present several applications in context of geometric optimization.

1.2 Related work

The first implementations of parametric search algorithms are due to Toledo [9] (solving extremal polygon placement problems) and Schwerdt et al. [8] (computing the diameter of moving points) roughly twenty years ago. Van Oostrum and Veltkamp [7] were the first to provide a general framework, written in `C++`. Their framework includes a detailed documentation as well as two reference implementations to compute the *Median of Lines* [6] and the *Fréchet distance between two polygonal curves* [2]. They experimented with different sorting algorithms and showed that Quicksort can replace a parallel sorting network in practice.

*Institut für Informatik, Universität Bayreuth, {christian.knauer, fabian.stehn}@uni-bayreuth.de,

†Institut für Informatik, Abteilung I, Universität Bonn, dkuebel@uni-bonn.de

1.3 Contribution

In Chapter 2, we present a new general parametric search framework. Two (geometric) algorithms that have been realized in this framework are discussed in Chapter 3.

The framework can be used as a black box to implement efficient optimization algorithms (based on decision algorithms) with minimal adaptation effort for the task at hand. It is designed to provide the capability to easily exchange not only the sorting algorithm (a central component of the parametric search technique) but also the strategy that is used to solve the comparisons of a batch, or the scheduler that manages the (simulated) parallel execution of the individual processes. Standard performance enhancements, such as "Cole's trick" [3] are built in along with other optimizations. Several parameters can be used to fine-tune the performance of the final algorithm.

From this point of view, our approach is twofold: On the one hand, the framework can be used as a black box that merely requires to provide an implementation of the corresponding decision algorithm and parts of $\mathcal{A}_p$ that determine and organize critical values. On the other hand, it allows a deeper look "under the hood" of the general mechanism of parametric search in order to study, analyse and compare an algorithm and to gain deeper insight into the original problem.

In Chapter 3, we discuss how a recent algorithm for computing shortcuts has been realized with the framework, and we show how the framework allows to study the effect of optimization strategies on the actual computation times. Numerical effects which may arise in practical applications are discussed briefly.

2 The Framework

In this section, we describe the general structure of our framework and discuss where and why it differs from the reference framework by van Oostrum et al. [7]. To keep the implementation effort as small as possible, van Oostrum et al. identify essential parts of the technique so that certain components can be reused in different implementations. Their framework provides Quicksort and Bitonicsort together with an automated organization of the comparisons in a batch. The framework encapsulates and hides the scheduling of comparisons and the management of critical values. In case that $\mathcal{A}_p$ is a parallel sorting algorithm, it is certainly a benefit to hide all this complexity which reduces the implementation effort considerably. If, however, $\mathcal{A}_p$ is not a sorting algorithm, the developer has to use "lower-level facilities for batching comparisons and suspending/resuming computation" instead which enforces the use of a rigid mechanism specified by their framework. This has certain drawbacks: It forces the developer to scatter code to scheduled methods which makes the parallel algorithm even harder to read, understand or debug. Realizing complex parallel algorithms is considerably more involved with this design. Moreover, the developer has no chance to control or influence the parallel steps and the evaluation of batches separately.

With our framework we offer the chance to hook into the scheduling or the handling of critical values, if desired or necessary. Both frameworks share (conceptually) similar components, e.g., a scheduler, processes or a decision algorithm. However, the design of our framework meets additional requirements. The most striking difference is that we separate the scheduling component from the management of critical values. We provide ready-to-use functionalities to reduce the implementation effort; c.f. Section 2.2. Among these are different strategies to resolve critical values of a batch which can be easily exchanged to study their impact on the overall performance. This provides an insight into how critical values are processed in a certain application and may reveal characteristics of the underlying problem.

The core components of the framework have already been released online [10]. The code of the whole parametric search framework together with the demo applications will be released as a part of a larger framework within this year.

2.1 Design Choices

Our framework is written in `Java` and consists of four components. A single responsibility is assigned to each component in order to (re-)use or interchange them independently. In the following, we briefly describe these components and their role within the framework.

One component is the serial decision algorithm $\mathcal{A}_s$ which has to be provided by the developer. The implementation has to support a method to decide DEC_P for any given value $\lambda \geq 0$. The outcome of a call to $\mathcal{A}_s$ is a boolean value; either TRUE (if $\lambda \geq \lambda_{opt}$) or FALSE (if $\lambda < \lambda_{opt}$).

The remaining three components constitute the parallel algorithm $\mathcal{A}_p$. We aim to restrict the access to the decision algorithm during the execution of $\mathcal{A}_p$. Whenever DEC_P needs to be solved for a concrete value λ, the component `Oracle` has to be asked: In contrast to $\mathcal{A}_s$, the oracle may also return the value UNKNOWN, implying that $\lambda \in I$, which forces the calling process to wait. This allows us to delay a single evaluation of the decision problem and to batch several critical values. Of course, at some points during the execution of $\mathcal{A}_p$ it will be necessary to evaluate the decision problem for (some of) the batched values, e.g., when $\mathcal{A}_p$ has to continue with its next parallel step. At this point, the oracle will apply a strategy

to resolve the comparisons of the current batch.

The flow of control of the parallel algorithm branches at certain points. This is realized by instances of `Process`, which allows for pausing and resuming, depending on the outcome to the calls to the oracle.

The component that manages the (virtual) parallel execution of $\mathcal{A}_p$ is the `Scheduler`: It assures that the oracle and all processes are triggered when necessary. After a certain number of parallel steps, all processes will terminate; as soon as the last processes becomes inactive, the scheduler and with it the entire algorithm will terminate.

2.2 Functionality and Oracle Strategies

With the framework we provide a parallel sorting algorithm based on a bitonic sorting network. It can be used to implement sorting based parametric search algorithms. The `Scheduler` is realized by a simple serial round-robin strategy. To experiment with different strategies of the `Oracle` component, we implemented the following strategies:

1. *Brute force.* This strategy does not store λ and solves the decision problem, at once. We do not maintain I. Consequently every request causes an evaluation of $\mathcal{A}_s$.
2. *Monotonicity.* This strategy only exploits the monotonicity of DEC_P: The decision problem is only evaluated if $\lambda \in I$. Otherwise, the outcome is determined according to the position of λ according to I in constant time.
3. *Parametric Search.* If λ lies in I, we store it in a list and return UNKNOWN. When the oracle is triggered via the method *resolveCollectedValues*, all stored values are resolved in a binary search fashion as suggested by Megiddo [6].
4. *Cole (unweighted).* In contrast to the previous strategy, the oracle evaluates DEC_P only for the median the of stored values. Afterwards, half of the critical values can be resolved in constant time. The corresponding processes are called to produce additional critical values.
5. *Cole (weighted).* To guarantee that no processes has to wait for a result for too long, a critical value is stored together with a certain weight. In contrast to the previous strategy, the decision problem is now evaluated for the weighted half of the stored values and not just for the median.

Depending on the specific application at hand, other strategies to handle and resolve batches can be realized or combined with these strategies. As a measurement for the performance we suggest to look at the total number of evaluations of $\mathcal{A}_s$.

3 Computing Optimal Shortcuts

In this section we briefly discuss two proof-of-concept implementations. Due to space limitations, we merely state the first implementation, the computation of the Fréchet Distance between two polygonal curves (c.f. [2]). For this implementation, Bitonicsort has been chosen to realize the parallel part of the algorithm; the corresponding decision problem was solved in a standard fashion via the use of free space diagrams.

The initial motivation that led to the design of this new framework is the problem of computing a *diameter-optimal shortcut* of a polygonal curve: A shortcut (a non-edge between two vertices of the path) is considered diameter-optimal if no other shortcut added to the curve results in a graph with smaller diameter. Große et al. [4] present a parametric search algorithm for this problem. In contrast to the computation of the Fréchet distance, their approach does not involve a sorting algorithm for $\mathcal{A}_p$.

Some Details

Let λ_{opt} denote the diameter induced by an optimal shortcut. The main idea of the concrete decision algorithm $\mathcal{A}_s$ is to check for every possible start vertex s of the shortcut whether there exists a feasible end vertex e. The vertex range of candidates for e is efficiently restricted through binary searches. $\mathcal{A}_s$ returns TRUE exactly if a feasible pair (s,e) was found.

Große et al. suggest to implement the decision algorithm in a generic and parallel fashion to get $\mathcal{A}_p$. This implies that each comparison in a binary search has to be deduced from the result of the decision problem at a critical value. For every candidate start vertex s, the search for a feasible e is independent and can be assigned to a separate parallel process. Consequently, Cole's weighting scheme [3] can be applied to reduce the theoretical worst-case running time by a log-factor.

We generated 1000 polygonal curves of 2^{10} vertices each, where the coordinates of the vertices were chosen uniformly at random within a square. For each curve, we computed the diameter-optimal shortcut using each of the five oracle strategies discussed above. As stated earlier, the number of calls to $\mathcal{A}_s$ is used to measure the performance of the individual strategies. Table 1 lists the outcome of the experiments, ordered by oracle strategy as discussed in Section 2.2. As expected, all strategies but the first (brute force), require a small number of evaluations of $\mathcal{A}_s$. With regard to the average number of calls, Strategy 2 performs best by only exploiting the monotonicity of the decision problem. As the last row of Table 1 reveals, Strategy 5, known as "Cole's trick" (which reduces the theoretical worst-case runtime by a log-factor), requires more calls to the decision problem than Strate-

	Strategy of the oracle				
	1	2	3	4	5
μ	$\sim$ 13,872	19.43	19.66	20.32	20.05
σ	$\sim$ 658	3.88	2.28	1.94	2.96
max	18,538	36	26	25	59
min	12,965	12	12	12	12

Table 1: Experimental results by oracle strategy. μ: average number of calls to $\mathcal{A}_s$; σ standard deviation; **max** (**min**): maximum (minimum) number of calls to $\mathcal{A}_s$.

gies $2-4$ for some instances.

The fact that Strategy 2 performs well is probably due to the order in which critical values are computed by $\mathcal{A}_p$. All distances from the first vertex to other vertices along the curve are critical values of the first batch (see [4] for details). Consequently, the interval I is already narrowed down right after the first batch has been resolved. The higher number of evaluations for Cole's weighing scheme in some instances might be due to the fact that this strategy calls $\mathcal{A}_s$ for values of λ that are far from λ_{opt}, as critical values of previous parallel steps are favoured over critical values of processes that have proceeded further.

It turns out that for the problem of computing an optimal shortcut, the critical values of the first batch are the key to achieve a small overall number of evaluations. The order in which critical values are computed and resolved plays an important role for the actual performance of a parametric search algorithm.

3.1 Numerical Issues

As for the built-in components of our framework, critical values are currently represented as double-precision numbers. The following problem can arise when representing critical values as finite-precision floats: The floating point representation of the optimal solution λ_{opt} is slightly smaller than the actual value of λ_{opt}. As a consequence, $\mathcal{A}_s$ rejects this value and it will be stored as the lower bound of $I = (\lambda^-,\lambda^+]$. If the algorithm outputs λ^+ as suggested above, the error can be huge. To address this problem, we can perform a final call to $\mathcal{A}_s$ with the center of I. If the center is a valid solution, we conclude that λ^- is closer to λ_{opt} than λ^+.

We are currently working on a solution to integrate and provide types that allow comparisons with arbitrary precision. Note that the general framework does not depend on specific numerical data-types. The specification and treatment of critical values has to be handled by the developer in a concrete application.

4 Conclusion & future work

With the presented framework it is possible to take a closer look into the details and the specific behaviour of parametric search algorithms. Different oracle strategies allow for quantified experiments under different optimization variants with no additional implementation effort. The outcome of these experiments might lead to a deeper understanding of the structure of the underlying problem.

Future versions of the framework might include the option to trace and visualize critical values throughout the parallel steps. In case of the shortcut problem, this would help to study the distribution of critical values around λ_{opt} and to understand under which conditions which parallel step calls $\mathcal{A}_s$ with the optimal value.

References

[1] Pankaj K. Agarwal, Micha Sharir and Sivan Toledo. *Applications of Parametric Searching in Geometric Optimization*, J. of Algorithms 17(3):292–318, 1994.

[2] Helmut Alt and Michael Godau. *Computing the Fréchet distance between two polygonal curves.* Int. J. of Comp. Geom. & App. 5:75–91, 1995.

[3] Richard Cole. *Slowing down sorting networks to obtain faster sorting algorithms.* J. ACM 34(1):200–208, 1987.

[4] Ulrike Große, Joachim Gudmundsson, Christian Knauer, Michiel H. M. Smid and Fabian Stehn. *Fast Algorithms for Diameter-Optimally Augmenting Paths.* Proceedings, Part I, ICALP (1) 2015: 678–688.

[5] Prosenjit Gupta, Ravi Janardan and Michiel H. M. Smid. *Fast Algorithms for Collision and Proximity Problems Involving Moving Geometric Objects.* Comput. Geom. 6: 371-391, 1996.

[6] Nimrod Megiddo. *Applying Parallel Computation Algorithms in the Design of Serial Algorithms.* J. ACM 30(4): 852–865, 1983.

[7] René van Oostrum and Remco C. Veltkamp. *Parametric search made practical.* Comput. Geom. 28(2-3): 75–88, 2004.

[8] Jörg Schwerdt, Michiel H. M. Smid and Stefan Schirra. *Computing the Minimum Diameter for Moving Points: An Exact Implementation Using Parametric Search.* Proceedings of the Thirteenth Annual Symposium on Computational Geometry, pages 466-468, 1997.

[9] Sivan Toledo. *Extremal Polygon Containment Problems and other issues in parametric searching.* Master's Thesis, Tel-Aviv University, 1991.

[10] Source Code: *Modular Parametric Search Framework (in Java)* https://davidkuebel@bitbucket.org/-davidkuebel/modularparametricsearchframework.git