Experimental Evaluation of Parametric Search Algorithms

Ivaylo Ilinkin*

Abstract

This paper provides an experimental report on implementation of parametric search algorithms. The algorithms are due to Matoušek and Cole et al. and answer the following questions about an arrangement of n lines: (a) compute a tangent to the k-level with a given direction d; (b) compute a tangent to the k-level through a given point p; and (c) compute the k-th vertex in the arrangement. The implementations are based on the framework of van Oostrum and Veltkamp, which simplifies considerably the development of parametric search algorithms.

1 Introduction

Parametric search is an optimization technique that has been used to obtain efficient algorithms for a wide range of problems [5]. In the parametric search setting one considers a decision problem, $P(\lambda)$, that is monotone in a real-valued parameter λ, i.e. if $P(\lambda') = T$, then $P(\lambda) = T$ for $\lambda < \lambda'$. The goal is to find the largest value, λ^*, for which $P(\lambda) = T$. During its execution a parametric search algorithm maintains an interval, $(\lambda_{lo}, \lambda_{up})$, where λ^* lies. This interval is refined repeatedly by updating one of the bounds until λ^* is discovered as one of the intermediate computations, or the interval is small enough that λ^* can be computed directly.

Despite its successful applications to a number of problems, parametric search algorithms are considered difficult to implement, and therefore, the technique has been primarily of theoretical interest. Indeed, until recently the only reported implementations of parametric search algorithms were by Schwerdt et al. [6] and Toledo [7]. Interest in the practical aspects of parametric search was renewed with the work of van Oostrum and Veltkamp [8] and more recently with the work of Goodrich and Pszona [3].

This paper discusses the implementation of the following algorithms due to Matoušek [4] and Cole et al. [2], respectively. Given a set of n lines:

Algorithm 1 [4]: Compute a tangent with a given direction, d, to the k-level in the arrangement.

Algorithm 2 [4]: Compute a tangent through a given point, p, to the k-level in the arrangement.

Algorithm 3 [2]: Compute the k-th vertex in the arrangement.

Note that the implementations discussed in this paper have an extra $O(\log n)$ factor, since the framework of van Oostrum and Veltkamp does not include Cole's technique [1]. Future work will consider adapting the implementations to the framework of Goodrich and Pszona [3], which has expected running time that matches Cole's improvement.

2 Parametric Search Framework

The framework of van Oostrum and Veltkamp [8] simplifies considerably the task of implementing parametric search algorithms by requiring that the user provide only a small number of classes:

Polynomials class: This is a representation of the objects that are compared during the search. Here the polynomials are the given lines. The outcome of the comparison depends on a critical value, λ_{ij}, defined by the polynomials ℓ_i and ℓ_j.

Comparison class: This compares two polynomials at the unknown value λ^*. The framework collects independent comparisons in a single batch to be resolved efficiently later once the answers for the associated critical values, λ_{ij}, are known.

Solver class: This solves the decision problem, $P(\lambda)$, for any concrete value, λ, which effectively answers whether $\lambda < \lambda^*$, $\lambda = \lambda^*$, or $\lambda > \lambda^*$. This is an expensive computation and the framework ensures that a number of decision questions are answered together in binary search fashion exploiting the monotonicity property.

For sorting-based parametric search the framework offers an implementation of *quick sort* based on the observation that it allows for convenient batching of comparisons. To this end, *quick sort* is modified so that during the *partition* step it first computes, for each comparison of two polynomials, the critical value λ that determines the outcome of the comparison, but does not evaluate the decision problem, $P(\lambda)$, yet.

*Department of Computer Science, Gettysburg College, iilinkin@gettysburg.edu

Instead, the decision problem is answered for the median of the critical values and this single expensive comparison computation makes it possible to deduce the answer for half of the comparisons. Therefore, all of the $O(n)$ comparisons during the *partition* step can be answered using only $O(\log n)$ expensive evaluations of the decision problem. Thus, the running time per level in *quick sort* is $O(T_S \log n)$, or $O(T_S \log^2 n)$ for the entire sort over the $O(\log n)$ expected levels, where T_S is the time for the *Solver* to answer the decision problem at any concrete value λ.

In essence, the framework abstracts the low-level decisions about coordinating the interactions between the *Solver* and the *Comparison* classes, which typically has been the challenge in implementing parametric search algorithms. Now, one can focus only on the high-level aspects of the problem — implementing an algorithm for the decision problem and a comparison predicate for two polynomials.

3 Algorithm 1

Recall that *Algorithm 1* seeks to find a tangent with a given direction, d, to the k-level in the arrangement of n lines. This algorithm is mentioned only in passing in [4], so we provide the details here for completeness. Its simplicity makes it a good candidate for illustrating parametric search and the use of the framework.

The main idea behind the algorithm is to sort the given lines along the unknown tangent τ^*. While it may seems strange to attempt to sort along the object we are trying to find, nevertheless, if this sorted order were somehow available, then the vertex at which τ^* touches the k-level will be determined by two adjacent lines. In fact, even though τ^* is not known, the order of two lines, ℓ_i and ℓ_j, along τ^* can be determined if it is known whether their intersection, r_{ij}, lies above or below τ^*.

The following two procedures aid in solving the overall problem [4]:

Procedure 1: Given a line q, the vertices of the k-level lying on q can be found in time $O(n \log n)$.

Procedure 1 works by intersecting q with the given lines and sorting the intersections. The level of the left-most intersection is computed directly and the levels of the subsequent intersections are computed via a scan along q updating each level by ± 1 as lines go above/below q The running time is dominated by the time to sort the intersections.

Procedure 2: The order of two lines, ℓ_i and ℓ_j, along τ^* can be determined in $O(1)$ time assuming that it is known *whether r_{ij} lies above or below τ^*.*

Procedure 2 relies on the observation that since the direction for τ^* is fixed, the order of ℓ_i and ℓ_j changes only when r_{ij} is crossed. Thus, if r_{ij} lies below τ^*, we can intersect ℓ_i and ℓ_j with any line with direction d that is above r_{ij} and infer their order along τ^*.

The decision problem, $P(y)$, that guides the sort is: *Does the line with direction d and height y intersect the k-level?*. Notice that the decision problem exhibits the monotonicity property, since as we sweep a line with direction d along the y-axis, the answer is $P(y) = T$ for all $y \leq y^*$, where y^* is the height of τ^*.

Consider a question about the relative order of ℓ_i and ℓ_j along τ^*. Form the critical line, τ_{ij}, through r_{ij} with direction d — if τ_{ij} intersects the k-level, then τ_{ij} is below τ^* and needs to be raised, i.e. $y_{ij} < y^*$; otherwise, if τ_{ij} misses the k-level, then it is above τ^* and needs to be lowered, i.e. $y_{ij} > y^*$. Thus, the answer to the decision problem, $P(y_{ij})$, answers $y_{ij} \lesseqgtr y^*$, which also answers whether r_{ij} is above/below τ^*.

The decision problem can be answered using *Procedure 1*, which makes it possible to use *Procedure 2* to determine the order of ℓ_i and ℓ_j along τ^*. Notice that *Procedure 1* and *Procedure 2* correspond to the *Solver* and *Comparison* classes, respectively, in the framework. Since the *Solver*'s running time is $O(n \log n)$, the running time of *Algorithm 1* is $O(n \log^3 n)$.

Finally, we highlight the genericity of the framework and the style of programming that it offers. Conceptually, one can view the *quick sort* algorithm offered by the framework as an *STL*-like *quick sort*. In fact, the original problem could be solved using an *STL*-like *quick sort* — the lines will be sorted correctly along τ^*, despite the fact that τ^* is not known:

```
bool comp(const Line& l1, const Line& l2)
{
    Point r = intersection(l1, l2);
    result_t  side = Procedure_1(r);
    if (side == ON) { abort(); }
    return Procedure_2(l1, l2, side);
}

std::quick_sort(lines.begin(), lines.end(),
                comp);
```

Of course, this is an inefficient way of solving the problem, since each comparison takes $O(n \log n)$ time for *Procedure 1* for an overall $O(n^2 \log^2 n)$ time over the $O(n \log n)$ comparisons. Nevertheless, it can be helpful to cast the solution in this form, since it ignores the low-level details of parametric search. Without the framework we would have to worry about those details next; instead, we can simply write:

```
vov::quick_sort(lines.begin(), lines.end(),
                Procedure1, Procedure2);
```

The above is a simplification and used only as a conceptual illustration. The actual code for setting

Table 1: Results for *Algorithm 1* (times in millisecs).

n	100	200	400	800	1000	10000
$k=1$	4.9	11.0	24.7	59.2	78.1	2188
$k=\frac{n}{8}$	4.9	10.9	24.3	57.1	78.5	1997
$k=\frac{2n}{8}$	4.7	10.6	25.6	59.1	77.3	2156
$k=\frac{3n}{8}$	4.6	10.6	24.8	61.5	82.6	2372

Table 2: Results for *Algorithm 2* (times in millisecs).

n	100	200	400	800	1000	10000
$k=1$	8.1	16.3	34.6	75.9	101.6	1806
$k=\frac{n}{8}$	8.0	15.7	33.9	76.8	101.6	1850
$k=\frac{2n}{8}$	7.3	15.6	33.7	77.5	101.0	1855
$k=\frac{3n}{8}$	7.0	15.4	34.3	78.5	102.7	1889

up the call to *quick sort* has a few more steps that are fairly routine. The main effort in using the framework is in implementing *Procedure1* and *Procedure2*, i.e. the *Solver* and *Comparison* classes, respectively.

Table 1 presents the execution times for different values of n and k. The experiments were run on a MacBook Pro with a 2.7 GHz Intel Core i7 processor and 8GB of RAM in a virtual machine running Linux Mint 15. The lines in each test case form a trellis configuration that is perturbed slightly, so that no two lines are parallel and each pair of lines intersects. (The algorithms are implemented to handle parallel lines, but these represent easy cases, and are not included in the experiments.) For each (n, k) pair in Table 1 the algorithm was run with 10 directions and each run was repeated 50 times averaging the results over all directions and trials. (Since the framework's implementation of *quick sort* randomizes the input to avoid unfavorable configurations, identical runs of *Algorithm 1* will have different actual running times.)

Note that the value k does not have appreciable effect on the running time, since ultimately the algorithm must sort the given n lines along τ^* regardless of the level of interest. The algorithm terminates prematurely if it happens to compare the lines ℓ_i and ℓ_j that define the vertex at which τ^* touches the k-level, which depends only on the initial ordering of the lines.

4 Algorithm 2

This algorithm seeks to find a tangent through a given point p to the k-level in the arrangement of n lines. The algorithm actually computes a tangent ray to the right of p and at the end ensures that its supporting line does not intersect the k-level using *Procedure 1*.

Again, the algorithm works by sorting the lines along the unknown tangent τ^*, but the decision problem, $P(\tau)$, that guides the sort is: *Does the line through p with the given slope, τ, intersect the k-level?* (τ will be used interchangeably to mean a line and its slope). This decision problem also exhibits the monotonicity property, since as we rotate a line τ around p, the answer $P(\tau) = T$ for all $\tau \leq \tau^*$.

To determine the order of two lines, ℓ_i and ℓ_j, along the unknown tangent τ^*, we form a critical line, τ_{ij}, through p and the intersection r_{ij} of ℓ_i and ℓ_j. If τ_{ij} intersects the k-level (to the right of p), then it needs to be rotated up around p, i.e. $\tau_{ij} < \tau^*$, which also implies that r_{ij} is below τ^*; otherwise, if τ_{ij} misses the k-level, it needs to be rotated down, i.e. $\tau_{ij} > \tau^*$ which also means that r_{ij} is above τ^*.

Unlike *Algorithm 1*, however, knowing the position of r_{ij} relative to τ^* is not enough to determine the order of ℓ_i and ℓ_j along τ^* — it is also necessary to know the relative order of the slopes of τ^*, ℓ_i, and ℓ_j. The following procedure can be used to find the range (τ_{lo}, τ_{up}) where τ^* lies.

Procedure 3: The range of slopes (τ_{lo}, τ_{up}) for τ^* in Algorithm 2 *can be found in time $O(n \log^2 n)$.*

Procedure 3 works by creating n lines through p with the slopes of the given lines, sorting them by increasing slope, and performing binary search. If the line with the median slope intersects the k-level, the lines with smaller slopes can be eliminated from consideration and τ_{lo} can be updated; otherwise, the other half of the lines can be eliminated and τ_{up} can be updated. There are $O(\log n)$ applications of *Procedure 1* resulting in $O(n \log^2 n)$ overall run time.

We can pick a fixed slope in the range (τ_{lo}, τ_{up}) and use it to form a test line through p. The order of ℓ_i and ℓ_j along the test line will be the same as the order along τ^* (as if it were rotated by a small amount).

The running time of *Algorithm 2* is the same as that of *Algorithm 1*. *Procedure 1* is again used as the *Solver* and *Procedure 2* as the *Comparison*. *Procedure 3* is run only once before the algorithm, so its run time is dominated by the overall run time of $O(n \log^3 n)$.

Table 2 shows the results for the same configurations as in *Algorithm 1*, but 10 points are chosen, instead of 10 directions, per test configuration.

5 Algorithm 3

The last algorithm considered in the paper seeks to find the k-th vertex in an arrangement of n lines. An optimal solution using parametric search and an approximation idea was given by Cole et al. [2], who also gave a sub-optimal, but less complicated, algorithm that relies only on parametric search. The latter algorithm is considered here.

The algorithm works by sorting the given lines at the vertical line, τ^*, through the unknown vertex, r^*. Again, we need to define two procedures that aid in determining the relative order of two lines ℓ_i and ℓ_j

along τ^*. The *Comparison* procedure uses the fact that as the arrangement is swept from left to right with a vertical line, τ, each pair of lines ℓ_i and ℓ_j changes its order along τ only at the pair's intersection, r_{ij}. Thus, the order of ℓ_i and ℓ_j along τ^* can be determined *if it is known* whether r_{ij} is left/right of τ^* — if r_{ij} is to the left of τ^*, then the line with the smaller slope comes first in the sorted order; otherwise, the line with the larger slope comes first.

In general, after the k'-th vertex, exactly k' pairs have switched their order along τ, i.e. there are exactly k' inversions. We define a decision problem $P(x)$: *Does the sorted order along the vertical line at x have fewer than k inversions?* and seek the largest value, x^*, for which $P(x) = T$. Once the sort is complete, the desired vertex, r^*, is the intersection of two adjacent lines along the vertical line at x^*.

The *Solver* procedure for the decision problem can be implemented using a modified version of *merge sort* which counts the elements that are out of order during the *merge* step. The *Solver* can be used to answer $P(x_{ij})$, i.e. $x_{ij} \lesseqgtr x^*$, which answers whether r_{ij} is left/right of τ^*, and therefore, the *Comparison* can determine the order of ℓ_i and ℓ_j along τ^*. Since the *Solver* runs in $O(n \log n)$, the overall algorithms has $O(n \log^3 n)$ running time.

Table 3 shows the results for the same configurations as in Algorithm 1. The times shown for each configuration represent searching for the first, last, and middle vertex in the arrangement, as well as the vertices that are one third away from either end (each search averaged over 50 trials). Unlike the previous algorithms, where the value of k does not have appreciable effect on the results, here the worst-case occurs when searching for the vertices at the extremes and the best case is when searching for the middle vertex.

Table 3: Results for *Algorithm 3* (times in millisecs; $N = \frac{n(n-1)}{2}$ number of vertices in the arrangement).

n	100	200	400	800	1000	10000
$k = 1$	2.0	4.6	11.2	25.8	34.4	752.6
$k = \frac{N}{3}$	1.8	3.4	8.2	18.8	27.2	452.6
$k = \frac{N}{2}$	0.4	0.8	2.2	4.6	6.0	131.0
$k = \frac{2N}{3}$	1.6	4.0	8.0	19.6	24.2	454.6
$k = N$	1.8	3.6	10.4	23.4	30.6	729.0

6 Conclusion

This paper has presented an experimental evaluation of parametric search algorithms and a discussion of their high-level implementation using the framework of van Oostrum and Veltkamp [8]. Long considered to be only of theoretical interest, parametric search has seen renewed attention with the work of van Oostrum and Veltkamp [8] and Goodrich and Pszona [3]. This paper provides further evidence about the practical aspects of parametric search and the hope is that it could inspire other experimental work to validate the applications of the technique.

The framework of van Oostrum and Veltkamp [8] simplifies considerably the implementation of parametric search algorithms. The framework hides the complexity of synchronizing the flow of control and lets the user focus on the high-level aspects of the problem. While the algorithms discussed here have an extra log factor, since they do not benefit from Cole's technique [1], it is likely that the impact on the actual execution time may not be significant, since there are no hidden constants and complexities that accompany more refined techniques. For example, the implementation of *Algorithm 3* essentially relies on two standard algorithms — the framework's *quick sort* and the modified *merge sort* for counting the number of inversions. Similarly, instead of *merge sort*, *Algorithms 1* and *2* use a procedure for intersecting a set of lines, which has a straightforward implementation.

In future work we plan to adapt the implementations to the framework proposed in [3] to remove the extra $O(\log n)$ factor in the current implementations. We are also working on an implementation of Matoušek's algorithm for computing the center of planar point sets, which uses multiple applications of parametric search including the algorithms discussed here.

References

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

[2] R. Cole, J. S. Salowe, W. L. Steiger, and E. Szemerédi. An optimal-time algorithm for slope selection. *SIAM J. Comput.*, 18(4):792–810, Aug. 1989.

[3] M. Goodrich and P. Pszona. Cole's parametric search technique made practical. In *Proc. 25th Canadian Conf. Comput. Geom.*, CCCG '13, pages 181–186, 2013.

[4] J. Matoušek. Computing the center of planar point sets. *Comput. Geom.: Papers from the DIMACS Special Year*, pages 221–230, 1991.

[5] N. Megiddo. Applying parallel computation algorithms in the design of serial algorithms. *J. ACM*, 30(4):852–865, Oct. 1983.

[6] J. Schwerdt, M. Smid, and S. Schirra. Computing the minimum diameter for moving points: an exact implementation using parametric search. In *Proc. 13th Annu. ACM Symp. Comput. Geom.*, SCG '97, pages 466–468, New York, NY, USA, 1997. ACM.

[7] S. Toledo. Extremal polygon containment problems and other issues in parametric searching. Master's thesis, Tel Aviv Univ., 1991.

[8] R. van Oostrum and R. Veltkamp. Parametric search made practical. *Comput. Geom. Theory Appl.*, 28(2-3):75–88, June 2004.