Computing the Minimum Diameter for Moving Points: An Exact Implementation Using Parametric Search*

Jörg Schwerdt† Michiel Smid† Stefan Schirra‡

1 Introduction

Parametric Search, developed by Megiddo [8], is a powerful algorithmic technique that can be used to solve a large variety of geometric optimization problems, see e.g. [1]. Although this technique is ingenious, it is, in general, hard to implement. In this paper, we report on the implementation of (a practical variant of) an algorithm that is based on parametric search, and that is due to Gupta *et al.* [7]. As far as we know, this is the first implementation of a parametric search algorithm.

The algorithm of [7] solves the following problem. We are given a set of n points in the plane that are moving at constant but possibly different velocities. The diameter of the points at time t is the largest Euclidean distance among all pairs of points at time t. Our goal is to compute the time t^* at which the diameter is minimum.

This problem can be solved trivially as follows. For each pair of points, the square of their distance defines a quadratic function in t, i.e., a parabola. The minimum diameter is obtained by finding the lowest point on the upper envelope of these $\binom{n}{2}$ parabolas. This gives an algorithm with running time $O(n^2 \log n)$, and using $O(n^2)$ space. It was shown in [7], that the problem can be solved, using parametric search, in $O(n \log^3 n)$ time, using $O(n)$ space. The latter algorithm is complicated and not practical; it uses e.g. an optimal parallel sorting algorithm. Therefore, we have implemented a variant of it, which has running time $O(n \log^6 n)$.

We implemented this algorithm in C++ using LEDA [9, 10] and CGAL [5, 6]. In our first implementation we used the number type `double` in all numerical computations. For many problem instances the implementation could not find the minimum diameter. During parametric search an interval containing t^*, the time at which the diameter is minimum, is maintained and repeatedly made smaller until it is finally shrunken to t^*. For many problem instances the interval does not collapse to a single point due to rounding errors and a fairly small interval was reported in the end. We got rid of these precision problems by using the number type `real` [3, 4] from LEDA [9] in the numerical computations, cf. Section 4.

Our implementation of the algorithm of Gupta et al. [7] shows that parametric search is not at all a purely theoretical and totally impractical technique in algorithm design in general and computational geometry in particular.

2 Applying Parametric Search

We assume that the reader is familiar with the parametric search technique, see [8, 1, 7]. Recall that t^* denotes the time at which the diameter of the moving points is minimum. In order to apply parametric search, we need a sequential algorithm A and a parallel algorithm B, each of which decides if a given $t \in \mathbb{R}$ is smaller, larger, or equal to t^*.

Let $Z_{pq}(t)$ denote the square of the Euclidean distance between the moving points p and q. Note that $Z_{pq}(t)$ is a parabola. Let t^*_{pq} be the time at which p and q are closest. Finally, let $D(t)$ denote the diameter at time t.

Lemma 2.1 *We have $t < t^*$ iff $t < t^*_{pq}$ for all points p and q such that $Z_{pq}(t) = D^2(t)$. A similar claim can be used to decide if $t > t^*$.*

Based on this lemma, algorithms A and B do the following. They first compute the positions of the points at time t. Then they compute all diametral pairs, i.e., all pairs p, q such that $Z_{pq}(t) = D^2(t)$. Finally, for each such pair, they check if $t < t^*_{pq}$, $t > t^*_{pq}$, or $t = t^*_{pq}$. Algorithm A can be implemented in $T_A(n) = O(n \log n)$ time, whereas algorithm B can be implemented on a PRAM such that it takes $T_B(n) = O(\log n)$ time using $P(n) = n$ processors.

Both A and B perform computation and comparison steps. Each computation step only involves the basic operations $+, -, *,$ and $/$, whereas each comparison step involves determining the sign of a polynomial at the value t. The coefficients of such a polynomial depend on the moving points only, and, as we will see later, its degree is at most three.

The parametric search algorithm computes t^* by sequentially simulating the parallel algorithm B on the unknown value t^*. Comparisons are resolved by running algorithm A on the roots of the corresponding polynomials. As is shown in [8], the total time for finding t^* is bounded by $O(P(n)T_B(n) + T_A(n)T_B(n) \log P(n)) = O(n \log^3 n)$.

* This paper is based on the first author's Master's Thesis [12].

† Fakultät für Informatik, Otto-von-Guericke-Universität Magdeburg, Universitätsplatz 2, D-39106 Magdeburg, Germany. E-mail: {schwerdt,michiel}@cs.uni-magdeburg.de.

‡ Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany. E-mail: stschirr@mpi-sb.mpg.de. This work was (partially) supported by the ESPRIT IV LTR Project No. 21957 (CGAL).

3 The Implementation

The implementation of the sequential algorithm A does the following. First, it calls a LEDA routine to sort the points—at a given time t—by their x-coordinates. Then the convex hull of the points is constructed by running Andrew's modification of Graham's Scan (see [11]) on the sorted sequence. Next, the rotating calipers algorithm is used to compute all diametral pairs of the convex hull. Finally, for each diametral pair p, q, it is decided if $t < t^*_{pq}$, $t > t^*_{pq}$, or $t = t^*_{pq}$. The entire algorithm has running time $O(n \log n)$.

Since the optimal parallel algorithm B is complicated, we decided to implement a simpler, and slower, variant. This algorithm does the following.

Using parallel merge-sort in a bottom-up fashion, the points—at time t—are sorted by their x-coordinates. This algorithm consists of $O(\log n)$ stages. At the beginning of stage i, we have $n/2^i$ sorted sequences, each of length 2^i. For each element x, there is one processor that uses binary search to find the rank of x in the neighboring sequence. Given these ranks, we can merge pairs of neighboring sequences. The entire sort algorithm takes $O(\log^2 n)$ time, using n processors. The polynomials that occur during comparison steps have degree one.

The convex hull of the sorted sequence is computed by parallel bottom-up merging. Again, this algorithm consists of $O(\log n)$ stages. At the beginning of stage i, we have the convex hulls of the leftmost 2^i points, the next 2^i points, and so on. For each pair of neighboring convex hulls, there is one processor that computes their upper and lower tangents using binary search. For resolving a comparison, nine cases have to be distinguished (see [11, Figure 3.21]), eight of which lead to a polynomial in t of degree two. Unfortunately, one case (concave–concave in [11, Figure 3.21]) leads to a polynomial of degree three. In this case, we do not proceed as in [11], but change the tangent finding algorithm, thereby slowing it down (see [12] for details). As a result, one stage may take $O(\log^3 n)$ parallel time. The entire parallel convex hull algorithm takes $O(\log^4 n)$ time using n processors. All polynomials that occur have degree two.

Given the convex hull, all diametral pairs are obtained as follows. For each hull edge $e = (p, q)$, there is one processor that uses binary search to find the at most two points a and b that are at maximum distance from the line through e. (The distances of the hull vertices to this line form a uni-modal sequence.) Then the four pairs (p, a), (p, b), (q, a), and (q, b) are inserted into appropriate positions of an array. All this takes $O(\log n)$ time using n processors. The polynomials that occur have degree two.

At this point, we know that the array contains all diametral pairs. By playing a tournament in parallel, level by level, these pairs are found in another $O(\log n)$ time using n processors. Again, the polynomials that are involved have degree two. Finally, given all diametral pairs, it can be decided in $O(1)$ time using n processors, whether $t < t^*$, $t > t^*$, or $t = t^*$. The comparisons in this final step do not depend on t^* and, hence, no polynomials are involved.

The entire algorithm B takes $O(\log^4 n)$ time and uses n processors. Moreover, the polynomials that arise at comparison steps all have degree at most two.

The sequential simulation of B on the value t^* is straightforward, and the total time for finding t^* is bounded by $O(n \log^6 n)$.

4 Computing Exact Results

Due to precision problems our implementation with `double`s often could not find t^*, but reported an interval only. Fig. 1 gives an example. The reported interval might be a sufficiently good approximation to t^* and in our examples it usually was. However, when we got the result of the implementation with `double` arithmetic we had no idea how good an approximation we had got. Even worse, we did not really know whether the fact that we got an interval and not a single time was caused by precision problems or a bug in the implementation.

exact:	$-\frac{507777}{587485} + \frac{1}{1174970}\sqrt{1239667408036}$
`double`:	(0.0832666379676957336... , 0.0832783171914589365...)
`real`:	0.0832783171914590203...

Figure 1: Results for t^*.

Replacing double arithmetic by exact rational arithmetic is not sufficient, because quadratic irrationals, real numbers of the form $A + B\sqrt{C}$, where A, B, and C are rational, arise in the computations (as roots of the polynomials). Fortunately the number type `real` [3, 4] in LEDA [9] provides exact computation for arithmetic expressions involving $+, -, \cdot, /$ and $\sqrt{\ }$. To be more precise, it provides exact comparisons of `real`s and hence the use of `real`s ensures correct control flow in the implementation: All decisions made in branching steps are correct, i.e., as if the computation would have been done with infinite precision.

In C++-code `real`s are very easy to use, they can be used exactly like `doubles` [3]. The switch from `double` to `real` in our code was simplified by the use of a preliminary version of the CGAL kernel [6], which provides geometric objects, especially points, parametrized by number types.

In addition, we wrote a number type which maintains quadratic irrationals symbolically in the form $A + B\sqrt{C}$ using LEDA `rational`s to represent A, B, and C. In the implementation all arithmetic operations are over elements from the same real quadratic field. In addition to arithmetic operations in a real quadratic field we had to provide a comparsion operation for quadratic irrationals from different extension fields. This comparison is made by repeated squaring.

5 Running Times

In our implementation we used the newest version of the LEDA number type `real` which incorporates recent results on separation bounds [2]. The old version was much too slow. For the old version it would have been necessary to provide reasonable separation bounds calculated by hand. The new version automatically finds such good bounds.

Fig. 2 shows ranges of running times in seconds of our `double` and `real` version for "random" point sets of size $n \in \{500, \ldots, 3000\}$ on a SUN SPARCstation 4. In the examples a moving point p was created by randomly choosing four integers x, y, v_x, v_y in the range $[-500, 500]$. The position of point p at time t is then $p(t) = (x, y) + t\,(v_x, v_y)$. For each value of n we created 10 different example point sets. It turned out that the exact `real` version is about 6 times

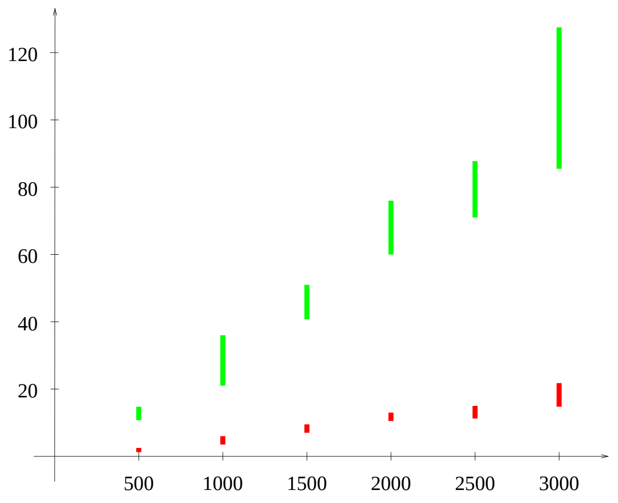

Figure 2: Running times in seconds for `real` version 🟩 and `double` version 🟥 .

slower than the inherently inexact `double` version, which returned time t^* in less than 20% of our test examples. The implementation with symbolic quadratic irrationals is about 40 times slower than the double version.

References

[1] P.K. Agarwal, M. Sharir and S. Toledo. *Applications of parametric searching in geometric optimization.* J. Algorithms **17** (1994), pp. 292–318.

[2] C. Burnikel, R. Fleischer, K. Mehlhorn, and S. Schirra. A strong and easily computable separation bound for arithmetic expressions involving square roots. In *Proceedings of the 8th ACM-SIAM Symposium on Discrete Algorithms*, 1997, pp. 702–709.

[3] C. Burnikel, J. Könemann, K. Mehlhorn, S. Näher, S. Schirra, and C. Uhrig. Exact geometric computation in LEDA. In *Proceedings of the 11th ACM Symposium on Computational Geometry*, 1995, pp. C18–C19.

[4] C. Burnikel, K. Mehlhorn, and S. Schirra. *The LEDA class real number.* Technical Report MPI-I-96-1-001, Max-Planck-Institut für Informatik, Saarbrücken, January 1996.

[5] CGAL Project. cf. `http://www.cs.ruu.nl/CGAL/`

[6] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. The CGAL kernel: A basis for geometric computation. In M.C. Lin and D. Manocha, editors, *Applied Computational Geometry: Towards Geometric Engineering*, Proceedings First ACM Workshop on Applied Computational Geometry, 1996, Springer LNCS 1148, pp. 191–202.

[7] P. Gupta, R. Janardan and M. Smid. *Fast algorithms for collision and proximity problems involving moving geometric objects.* Computational Geometry: Theory and Applications **6** (1996), pp. 371-391.

[8] N. Megiddo. *Applying parallel computation algorithms in the design of serial algorithms.* Journal of the ACM **30** (1983), pp. 852–865.

[9] K. Mehlhorn, S. Näher, and C. Uhrig. *The LEDA User Manual*;
cf. `http://www.mpi-sb.mpg.de/LEDA/leda.html`

[10] K. Mehlhorn and S. Näher. *LEDA: A platform for combinatorial and geometric computing.* Communications of the ACM **38** (1995), pp. 96–102.

[11] F.P. Preparata and M.I. Shamos. *Computational Geometry, an Introduction.* Springer–Verlag, 1985.

[12] J. Schwerdt. *Das Diameterproblem einer bewegten Punktemenge: eine Implementierung mit Hilfe von Parametric Search.* Master's Thesis (Diplomarbeit), Universität des Saarlandes, Saarbrücken, 1996.