Solving geometric optimization problems using parametric search

Michiel Smid*

November 25, 2002

1 Introduction

We now turn to the most difficult topic of this course[1]: the parametric search technique, invented by Megiddo in 1983. This is a very powerful and general (but somewhat abstract and mysterious) technique that can be used to solve a large variety of geometric optimization problems. Applications of this technique include

1. computing the k-th leftmost vertex in an arrangement of n lines in the plane,
2. computing the diameter of a set of n points in $\mathbb{R}^3$,
3. computing the width of a set of n points in $\mathbb{R}^3$, which is defined as the width of the narrowest slab that contains all points,
4. computing the width of the smallest ring (bounded by two concentric circles) that contains a given set of n points in the plane, and
5. computing a ray starting at the origin that is as far away as possible from a given set of n points in $\mathbb{R}^3$. (We saw already how to solve the planar version of this problem.)

*School of Computer Science, Carleton University, Ottawa, Ontario, Canada K1S 5B6. E-mail: michiel@scs.carleton.ca.

[1]Endlich mal was kompliziertes!

We will use the following problem to illustrate the technique.

Problem 1 *Let S be a set of n points in the plane. Each point of S moves with a constant velocity along a straight line. Hence, the position of any point p of S at time t can be written as*

$$p(t) = (a_p + v_p t, b_p + w_p t).$$

Let $D(t)$ denote the diameter of S at time t, i.e.,

$$D(t) = \max\{d(p(t), q(t)) : p, q \in S\}.$$

Our task is to compute the time t^* at which the diameter has its minimum value, i.e.,

$$D(t^*) = \min\{D(t) : t \in \mathbb{R}\}.$$

We first give an easy but inefficient solution to this problem. Then, we describe the parametric search technique and show how it can be applied to obtain a faster algorithm for solving Problem 1. In order to avoid a case analysis, we make the following assumption.

Assumption 1 *For all $p, q \in S$, $p \neq q$, we have $(v_p, w_p) \neq (v_q, w_q)$. That is, the distance function $d(p(t), q(t))$ is a non-constant function of time.*

2 An easy solution to Problem 1

For any two distinct points p and q of S, we define

$$f_{pq}(t) := d^2(p(t), q(t)) = (a_p - a_q + (v_p - v_q)t)^2 + (b_p - b_q + (w_p - w_q)t)^2.$$

Then,

$$D^2(t) = \max\{f_{pq}(t) : p, q \in S, p \neq q\}.$$

Clearly, instead of minimizing the function $D(t)$, we can as well minimize $D^2(t)$. The function $f_{pq}(t)$ is a polynomial in t of degree two. The function $D^2(t)$ is the *upper envelope* of all these polynomials. (There are $\binom{n}{2}$ of these.) Hence, t^* is the x-coordinate of the lowest point on this upper envelope.

Theorem 1 *Problem 1 can be solved in $O(n^2 \log n)$ time, using $O(n^2)$ space.*

Exercise 1 Prove this theorem using the results of the chapter on lower envelopes.

How can we improve this result? As usual, we should try to discover some properties of the solution to our problem. In our case, the upper envelope has a nice structure:

Lemma 1 *The upper envelope of the functions f_{pq}, where $p, q \in S$, $p \neq q$, is a convex function. It consists of a decreasing part followed by an increasing part.*

Proof. The claim holds for each f_{pq} separately. But then it also holds for the piecewise maximum of these functions. This piecewise maximum is the upper envelope. ∎

How does this lemma help us? If we know the upper envelope, then we can find its lowest point, and hence t^*, by a binary search in the sorted sequence of vertices of the upper envelope, in $O(\log n)$ time. But, the upper envelope has size $\Theta(n^2)$, so computing it explicitly takes $\Omega(n^2)$ time. Therefore, in order to obtain an $o(n^2)$-time algorithm, we must be able to search on the upper envelope *without* completely computing it. The parametric search technique enables us to do this. The following lemmas are crucial.

Lemma 2 *For $p, q \in S$, $p \neq q$, let t^*_{pq} denote the time at which p and q are closest, i.e.,*

$$f_{pq}(t^*_{pq}) = \min\{f_{pq}(t) : t \in \mathbb{R}\}.$$

Let $t \in \mathbb{R}$. Then $t > t^*$ if and only if $t > t^*_{pq}$ for all $p, q \in S$, $p \neq q$, such that $f_{pq}(t) = D^2(t)$.

Proof. Recall that each function $f_{pq}(t)$ is a parabola. Suppose that $t > t^*_{pq}$ for all $p, q \in S$, $p \neq q$, such that $f_{pq}(t) = D^2(t)$. Then the upper envelope is increasing in a small neighborhood of t. Hence, the (unique) lowest point of the upper envelope lies to the left of t. That is, $t > t^*$. The converse can be proved by a similar argument. ∎

Lemma 3 *Given the points of S, together with the equations describing their positions over time, and given an arbitrary $t \in \mathbb{R}$, we can in $O(n \log n)$ time decide whether $t > t^*$, $t < t^*$ or $t = t^*$.*

Proof. The following algorithm does the job: Compute the positions of all points of S at time t. Then, compute all diametral pairs at time t. For each such pair $p(t), q(t)$, check whether $t > t^*_{pq}$. If this is true for all diametral pairs, then $t > t^*$. Otherwise, we have $t \leq t^*$. Using a symmetric argument, we can decide whether $t < t^*$ or $t \geq t^*$. The bound on the running time follows from the results in the chapter on the diameter problem. ∎

3 The general framework

In this section, we define the class of problems that can be solved by the parametric search technique.

Let S be a set of n objects, and let $\mathcal{P}(t)$ be a decision problem whose value depends on S and a real parameter t.[2] That is, given $t \in \mathbb{R}$, $\mathcal{P}(t)$ is either *true* or *false*. We assume that $\mathcal{P}$ is *monotone*, which means that if $\mathcal{P}(t_0)$ is *false* for some real number t_0, then $\mathcal{P}(t)$ is also *false* for all $t < t_0$. We also assume that there is a maximum value $t^* \in \mathbb{R} \cup \{-\infty, \infty\}$ for which $\mathcal{P}$ is *false*. That is, t^* is given by

$$t^* = \max\{t \in \mathbb{R} \cup \{-\infty, \infty\} : \mathcal{P}(t) = \mathit{false}\}.$$

Note that $t^* = \infty$ in case $\mathcal{P}(t)$ is *false* for all $t \in \mathbb{R}$. Similarly, $t^* = -\infty$ in case $\mathcal{P}(t)$ is *true* for all $t \in \mathbb{R}$.

Our optimization problem is that of computing t^*. The parametric search technique gives an algorithm for solving this problem, if the following holds:

There is an algorithm A that, given the set S and any real number t as input, decides whether $t < t^*$, $t > t^*$ or $t = t^*$. In particular, A decides whether $\mathcal{P}(t)$ is *true* or *false*. This algorithm performs *computation* steps and *comparison* steps. We assume that each computation step only involves the basic operations $+, -, *, /$ and $\sqrt{\cdot}$. We also assume that each comparison step involves determining the sign—positive, negative or zero—of some polynomial at the value t. The coefficients of such a polynomial only depend on the set S. Also, the degree of such a polynomial should be bounded by a small constant. Finally, we assume that the value of each variable used by A can be written as a polynomial in t. Again, the coefficients of such a polynomial only depend on S, and its degree is bounded by a small constant.

[2] Actually, we should write $\mathcal{P}(S, t)$. Since S is assumed to be fixed, however, we just write $\mathcal{P}(t)$.

As an illustration, let us look at Problem 1. Here, S consists of the n points that are moving in the plane. Each point p of S is given as $p(t) = (a_p + v_p t, b_p + w_p t)$. Recall that the diameter of S at time t is denoted by $D(t)$. We define the decision problem $\mathcal{P}$ as follows. For any $t \in \mathbb{R}$,

$$\mathcal{P}(t) := \begin{cases} \mathit{true} & \text{if } t > t^*_{pq} \text{ for all } p, q \in S,\ p \neq q, \text{ such that } f_{pq}(t) = D^2(t), \\ \mathit{false} & \text{otherwise.} \end{cases}$$

Then it follows from Lemmas 1 and 2 that $\mathcal{P}$ is monotone. Moreover, the value of

$$t^* := \max\{t \in \mathbb{R} : \mathcal{P}(t) = \mathit{false}\}$$

is the time at which the diameter of the moving points of S has its minimum value. Hence, Problem 1 is an example of the general class of optimization problems defined above.

How does the algorithm A look like? This algorithm should take as input the points of S and a real number t, and decide whether $t > t^*$, $t < t^*$ or $t = t^*$. By Lemma 3, such an algorithm exists. In fact, we gave an algorithm in the proof of that lemma. Since this algorithm fits into the algebraic computation tree model, each of its computation steps only involves the basic operations $+, -, *, /$ and $\sqrt{\cdot}$.

What about the comparison steps? In order to answer this question, we have to look at the algorithm more closely. It first computes the convex hull of the points of S at time t. Then it computes all diametral pairs of this convex hull. From the results of the previous chapters, we know that only three types of comparisons occur in this algorithm:

Type 1: Given three distinct points $p(t)$, $q(t)$ and $r(t)$, decide whether $r(t)$ is on, to the left, or to the right of the directed line from $p(t)$ to $q(t)$.

This is equivalent to deciding whether the determinant

$$\begin{vmatrix} a_p + v_p t & b_p + w_p t & 1 \\ a_q + v_q t & b_q + w_q t & 1 \\ a_r + v_r t & b_r + w_r t & 1 \end{vmatrix}$$

is zero, positive or negative. Writing this out, however, shows that this is nothing but determining the sign of a polynomial of degree two at the given value t. This is exactly the requirement we made.

Type 2: Given four distinct points $p(t)$, $q(t)$, $r(t)$ and $s(t)$, such that $r(t)$ and $s(t)$ are to the left of the directed line ℓ from $p(t)$ to $q(t)$, decide whether $d(r(t), \ell) > d(s(t), \ell)$, $d(r(t), \ell) < d(s(t), \ell)$ or $d(r(t), \ell) = d(s(t), \ell)$.

This is equivalent to deciding which of the two determinants

$$\begin{vmatrix} a_p + v_p t & b_p + w_p t & 1 \\ a_q + v_q t & b_q + w_q t & 1 \\ a_r + v_r t & b_r + w_r t & 1 \end{vmatrix}$$

and

$$\begin{vmatrix} a_p + v_p t & b_p + w_p t & 1 \\ a_q + v_q t & b_q + w_q t & 1 \\ a_s + v_s t & b_s + w_s t & 1 \end{vmatrix}$$

is larger, or whether they are equal. This is again determining the sign of a polynomial of degree two at the value t.

Type 3: Given the four points $p(t)$, $q(t)$, $r(t)$ and $s(t)$, decide whether $d(p(t), q(t)) < d(r(t), s(t))$, $d(p(t), q(t)) > d(r(t), s(t))$ or $d(p(t), q(t)) = d(r(t), s(t))$.

By comparing the squares of the distances, we again want to determine the sign of a polynomial of degree two at the given value t.

Remark 1 Some convex hull algorithms first sort the points by their x-coordinates. Comparing the x-coordinates of two points at time t is of course nothing but determining the sign of a polynomial of degree one at the value t.

We have shown that the algorithm satisfies all requirements we made. As a result, the parametric search technique gives an algorithm for solving Problem 1.

Exercise 2 For each of the five optimization problems mentioned in Section 1, define the corresponding monotone decision problem $\mathcal{P}(t)$.

4 A slow version of parametric search

Recall that we are given a set S of n objects, a monotone decision problem $\mathcal{P}(t)$, and algorithm A. Our goal is to compute the value of t^* defined as

$$t^* = \max\{t \in \mathbb{R} \cup \{-\infty, \infty\} : \mathcal{P}(t) = \mathit{false}\}.$$

Given $t \in \mathbb{R}$, we can decide if $t < t^*$, $t > t^*$ or $t = t^*$, by running algorithm A on input S and t. Since S is fixed, we will simply say that we "run A on t".

How do we compute t^*? Here is the answer: We run algorithm A on t^*. But, we do not know t^*! Therefore, the precise answer is: We run A on a value t which we *pretend* to be t^*.

What do we mean by this? Suppose we know t^*. Then, running A on t^* gives a computation consisting of a sequence of computation steps and comparison steps. Now, in order to compute t^*, which in reality is not known to us, we run A on a symbolic variable t and "compute" exactly this sequence of computation and comparison steps. As we will see, during this computation, we find t^*.

The details are as follows. During the algorithm, we maintain an interval I for which we know that it contains t^*. Initially, we assign $I := [-\infty, \infty]$.

Now we start simulating algorithm A on the unknown value t^*. A computation step is easy to simulate: we just perform the basic operation, where the new value of the variable involved is written as a formula in t.

As an example, suppose we compute the convex hull of our moving points using Graham's Scan. One variant of this algorithm first chooses a point c—that does not necessarily belong to the point set—that is known to be inside the convex hull. Then it sorts the points in radial order around c. A good choice for c is the centroid of three non-collinear points. Hence, in order to obtain c, the following computation step is made: Take three distinct non-collinear points $p(t)$, $q(t)$ and $r(t)$, and compute

$$c := \left(\frac{1}{3}(a_p + a_q + a_r) + \frac{1}{3}(v_p + v_q + v_r)t\,,\, \frac{1}{3}(b_p + b_q + b_r) + \frac{1}{3}(w_p + w_q + w_r)t\right).$$

The variable c is expressed by two formulas involving t.

How do we simulate a comparison step? We know that a comparison step consists of determining the sign of a low-degree polynomial in t. Denote this polynomial by $f(t)$, and let it have degree d, where d is a small integer. Note that we know f explicitly. In order to simulate the comparison step correctly, we have to find out if $f(t^*)$ is positive, negative or zero. How can we decide this if we do not know t^*? Here is the trick: We compute the real roots $t_1 < t_2 < \ldots < t_k$ of the polynomial f. Since f has degree d, we have $0 \leq k \leq d$.

If $k = 0$, then $f(t)$ is either always positive or always negative. Since we know f explicitly, we can easily find out in which case we are. Hence, we know the sign of $f(t^*)$ and we can correctly simulate the current comparison step. (In fact, if $k = 0$, then this step is not really a comparison step, because the outcome is always the same.)

Assume now that $k > 0$. In order to simulate the current single comparison step, we run algorithm A on each of the roots $t_1, t_2, \ldots, t_k$. For each i, $1 \leq i \leq k$, the i-th run tells us whether $t_i < t^*$, $t_i > t^*$ or $t_i = t^*$. Of course, if $t_i = t^*$ for some i, we can stop the algorithm: our task was finding t^*, and we succeeded in doing this. So assume that $t_i \neq t^*$ for all i. Then at the end of the k runs of A, we know exactly which of the intervals

$$[-\infty, t_1), (t_1, t_2), (t_2, t_3), \ldots, (t_{k-1}, t_k), (t_k, \infty]$$

contains t^*. Since we know our polynomial f explicitly, we know the sign of f within each of the intervals. Define $t_0 := -\infty$ and $t_{k+1} := \infty$. Then, if $t^* \in (t_i, t_{i+1})$, we know the sign of the polynomial f at t^*, and we can correctly simulate our comparison step. We update I by assigning $I := I \cap (t_i, t_{i+1})$, and proceed with the next step of the simulation.

Consider again our example problem. Suppose the comparison step is of Type 1. (See Section 3.) Let $p(t) = (1+t, 1-t)$, $q(t) = (3-2t, -1+3t)$ and $r(t) = (3-3t, 4t)$. Hence, we want to determine the sign of the determinant

$$\begin{vmatrix} 1+t & 1-t & 1 \\ 3-2t & -1+3t & 1 \\ 3-3t & 4t & 1 \end{vmatrix} = t^2 - 3t + 2 = (t-1)(t-2),$$

at the value t^*. In this case, we have $f(t) = (t-1)(t-2)$, $t_1 = 1$ and $t_2 = 2$. We run A on t_1. This tells us whether $t_1 < t^*$, $t_1 > t^*$ or $t_1 = t^*$. Similarly, running A on t_2 tells us whether $t_2 < t^*$, $t_2 > t^*$ or $t_2 = t^*$. If $t_1 = t^*$ or $t_2 = t^*$, then we can stop the algorithm. Otherwise, if $t^* \in [-\infty, t_1)$, then we know that $f(t^*) > 0$; if $t^* \in (t_1, t_2)$, then $f(t^*) < 0$; and if $t^* \in (t_2, \infty]$, then $f(t^*) > 0$. Hence, we (i) have simulated the comparison step correctly and (ii) can update the interval I that contains t^*.

In this way, our algorithm simulates the computation of A on the input t^*. There are two possibilities for our algorithm to stop. First, if t^* is equal to one of the roots of the polynomial that arises during a comparison step, our algorithm stops. Otherwise, our algorithm stops because it has simulated the entire computation of A on the input t^*. In the rest of this section, when we talk about *our simulation*, then we refer to the above algorithm that simulates A on t^*.

In order to argue more formally about our simulation, we will consider the *computation tree* of algorithm A. This tree represents all possible computations of A over all inputs $t \in \mathbb{R}$.[3] There are three types of nodes in the tree.

[3]The computation tree is of course just an analysis tool. We only use it to argue about

A *computation node* is labeled with a computation step. Such a node has only one child. A *comparison node* is labeled with a comparison of the form $f(t)\,?\,0$. This node has three outgoing edges, leading to its three children. The leftmost (resp. middle, rightmost) outgoing edge is labeled with $<$ (resp. $=$, $>$). Finally, a *leaf* is a node that does not have any children. A leaf is labeled with one of the labels "$t < t^*$", "$t = t^*$" and "$t > t^*$". Additionally, we give the leaf the label *true* if $t > t^*$, and *false* if $t < t^*$ or $t = t^*$.

The algorithm A defines a computation tree in the natural way. Given $t \in \mathbb{R}$, running A on t corresponds to following a path in this tree from the root to a leaf. The label stored with this leaf tells us whether $t < t^*$, $t = t^*$ or $t > t^*$. In particular, it tells us whether $\mathcal{P}(t)$ is *true* or *false*.

Our simulation also defines a path in the computation tree of A. This path starts in the root. Each time a computation step or comparison step is simulated, this path is extended in the obvious way.[4] This path does not necessarily end in a leaf of the tree: If during a comparison step, we find out that t^* is the root of the corresponding polynomial, then the path ends.

The following lemma states that our simulation indeed correctly simulates algorithm A on the input t^*.

Lemma 4 *The path in the computation tree of A that is defined by our simulation is a prefix of the root-to-leaf path defined by running A on t^*.*

Proof. The proof follows from the description of the algorithm and the discussion above. ∎

We talked about two possibilities for our algorithm to stop. The next lemma states that if t^* is finite, then there is just one possibility.

Lemma 5 *Assume that t^* is finite. Then for one of the simulated comparison steps, the corresponding polynomial has t^* among its roots.*

Proof. Suppose the claim is not true. Then our simulation simulates the entire computation of A on t^*. Consider the interval I at the end of the algorithm. We know that I contains t^*.

---

algorithm A.

[4] During the simulation of a comparison step, we run A on the roots of the corresponding polynomial. We do not consider the corresponding paths in the computation tree. These computations are only needed for determining the outcome of the current comparison.

First, we claim that for each $t \in I$, running A on t defines the same path in the computation tree as running A on t^*. Why? The way we construct the interval I implies that for each comparison step—with corresponding polynomial f—that is made when running A on t and on t^*, $f(t)$ and $f(t^*)$ have the same sign.

Hence, the path in the computation tree corresponding to any $t \in I$ ends in the same leaf ℓ. Since $\mathcal{P}(t^*) = \mathit{false}$, and $t^* \in I$, we know that ℓ is labeled *false*. We distinguish two cases.

Case 1: I has ∞ as its right endpoint, i.e., it has the form $(a, \infty]$ or $[-\infty, \infty]$.

Let $t > t^*$. Then $t \in I$ and running A on t leads to leaf ℓ in the computation tree. Since ℓ has label *false*, it follows that $\mathcal{P}(t) = \mathit{false}$. This is a contradiction, because t^* is the largest value at which $\mathcal{P}$ is *false*.

Case 2: The right endpoint of I is finite, i.e., it has the form (a, b) or $[-\infty, b)$.

Note that it follows from the description of our simulation that I is indeed *open* at its right end. Let $t \in I$ such that $t^* < t < b$. Then the same argument as in Case 1 leads to a contradiction. ∎

Lemma 5 proves that our simulation is correct if t^* is finite: It finds t^* during one of the simulated comparison steps. The next lemmas tell us what happens when t^* is not finite.

Lemma 6 *If $t^* = \infty$, then our simulation completes the entire simulation of A on t^*. We have $t^* = \infty$ if and only if at the end of our simulation the interval I has ∞ as its right endpoint.*

Proof. The proof follows from the facts that (i) ∞ cannot be a root of any polynomial, (ii) $t^* \in I$, (iii) $\mathcal{P}(t^*) = \mathit{false}$ and (iv) $\mathcal{P}(t) = \mathcal{P}(t^*)$ for all $t \in I$. ∎

Lemma 7 *If $t^* = -\infty$, then our simulation completes the entire simulation of A on t^*. We have $t^* = -\infty$ if and only if at the end of our simulation the interval I has the form $[-\infty, b)$ for some (finite) $b \in \mathbb{R}$.*

Proof. The proof is similar to that of Lemma 6. Note that I is open at its right end. ∎

At this moment, we know that our simulation correctly solves our optimization problem: It always finds t^*. What is the running time? Let $T(n)$

denote the running time of A. That is, running A on any $t \in \mathbb{R}$ takes at most $T(n)$ computation and comparison steps.

Our simulation needs only $O(1)$ time for simulating a computation step. A comparison step is simulated by running A on at most d roots.[5] Since d is a constant, the comparison step is simulated in $O(T(n))$ time. As a result, our entire simulation takes $O(T^2(n))$ time. We have proved the following result.

Theorem 2 *Let $\mathcal{P}(t)$ be a monotone decision problem whose value depends on a set S of n objects and a real parameter t. Let*

$$t^* := \max\{t \in \mathbb{R} \cup \{-\infty, \infty\} : \mathcal{P}(t) = \mathit{false}\}.$$

Let A be an algorithm that, given as input the set S and any real number t, decides whether $t < t^*$, $t > t^*$ or $t = t^*$, and that satisfies the conditions given in Section 3. Let $T(n)$ denote the running time of A. Then t^* can be computed in $O(T^2(n))$ time.

Let us apply this theorem to our example problem. In this case, algorithm A is given by Lemma 3, and we have $T(n) = O(n \log n)$. Hence, we can compute the time t^* at which the diameter of the moving points is minimum, in $O(n^2 \log^2 n)$ time.

But, in Section 2 we saw a *much simpler* algorithm that solves the same problem in only $O(n^2 \log n)$ time! Did you, dear student, waste your time while suffering through the previous pages? The answer is, of course, "no".

What is the reason that the running time of our simulation is so high? Well, for *every* comparison step that we simulate, we run algorithm A on at most d inputs. We do this because of the following reason: We only know the *next* comparison step after we have determined the outcome of the *current* comparison. To be more precise, in order to resolve the current comparison, we first run A on at most d inputs. Knowing the outcome, we follow the corresponding edge in the computation tree. This tells us which comparison step we have to simulate next.

In the next section, we will give an improved version of parametric search that simulates *many* comparison steps "simultaneously". As we will see, in this way we can decide the outcomes of many comparisons by running A on only a relatively small number of inputs.

[5]Here, d denotes the maximum degree of the polynomial that corresponds to any comparison step. Recall that we assume d to be a small constant.

5 An improved version of parametric search

The discussion above implies that the main question to be answered is: How can we "collect" a large number of comparison steps and resolve them simultaneously? Here is the answer: We replace the *sequential* algorithm A by a *parallel* algorithm.

The ingredients of the improved algorithm are as follows. As before, S is a set of n objects, and $\mathcal{P}(t)$ is a monotone decision problem whose value depends on S and the real parameter t. We want to compute

$$t^* = \max\{t \in \mathbb{R} \cup \{-\infty, \infty\} : \mathcal{P}(t) = \mathit{false}\}.$$

We assume that we have a sequential algorithm A that, given as input S and any $t \in \mathbb{R}$, decides whether $t < t^*$, $t > t^*$ or $t = t^*$. This algorithm satisfies the conditions given in Section 3. The worst-case running time of A on any input t is denoted by $T_A(n)$.

Additionally, we assume that we have a parallel algorithm B that, again on input S and any $t \in \mathbb{R}$, decides whether $t < t^*$, $t > t^*$ or $t = t^*$. This algorithm can be implemented on a PRAM, and its running time (resp. number of processors) is denoted by $T_B(n)$ (resp. $P(n)$). That is, for any $t \in \mathbb{R}$, algorithm B takes at most $T_B(n)$ parallel steps while using $P(n)$ processors. Each processor performs computation steps and comparison steps that satisfy the conditions of Section 3.

Now we can give the improved algorithm for computing t^*. The main idea is to sequentially simulate algorithm B on the unknown value t^*.

The details are as follows. As in the previous section, we maintain an interval I that contains t^*. Initially, we assign $I := [-\infty, \infty]$.

Now the simulation starts. Consider a parallel step of algorithm B. During this step, each of the $P(n)$ processors performs one computation step or one comparison step. Let X (resp. Y) denote the total number of computation (resp. comparison) steps that are performed during this parallel step.

It is clear that the X computation steps can be simulated in $O(X)$ sequential time. How do we simulate the Y comparison steps? Let $f_1(t), f_2(t), \ldots, f_Y(t)$ denote the polynomials that correspond to these comparisons. Note that each polynomial has degree at most d, which is assumed to be a small constant. For $m = 1, 2, \ldots, Y$, we have to determine the sign of $f_m(t^*)$. In order to determine all these signs, we do the following:

For $m = 1, 2, \ldots, Y$, we compute the real roots $t_{m1} < t_{m2} < \ldots < t_{mk_m}$ of the polynomial f_m. This gives us a sequence of at most dY real numbers, which we sort in non-decreasing order. Now we perform a binary search in this sorted list, and determine between which roots t^* lies. Deciding if $t^* < t_{ij}$, $t^* > t_{ij}$ or $t^* = t_{ij}$ is done by running the sequential algorithm A on input t_{ij}.

If t^* is equal to one of the roots of these Y polynomials, then the algorithm will find this root during the binary search. In this case, the algorithm stops because it has found t^*. Otherwise, the binary search gives us two consecutive roots $t_{ij} < t_{k\ell}$ that contain t^* between them. (Note that there are no roots between t_{ij} and $t_{k\ell}$.) Now for each $m = 1, 2, \ldots, Y$, we can in $O(1)$ time decide which of the intervals

$$[-\infty, t_{m1}), (t_{m1}, t_{m2}), (t_{m2}, t_{m3}), \ldots, (t_{m,k_m-1}, t_{mk_m}), (t_{mk_m}, \infty]$$

contains t^* and, hence, determine the sign of $f_m(t^*)$. Finally, we update the interval I, by assigning $I := I \cap (t_{ij}, t_{k\ell})$. In this way, we have determined the outcomes of the Y comparisons, in sequential time

$$O(Y \log Y + T_A(n) \log Y).$$

Hence, we have correctly simulated a single parallel step of algorithm B on the input t^*. Since $X \leq P(n)$ and $Y \leq P(n)$, the total time for simulating this parallel step is

$$O(P(n) \log P(n) + T_A(n) \log P(n)).$$

In exactly the same way as in Section 4, it can be shown that our entire simulation correctly computes the value of t^*. Since B makes at most $T_B(n)$ parallel steps, it follows that our complete algorithm takes time

$$O(T_B(n)P(n) \log P(n) + T_B(n)T_A(n) \log P(n)).$$

This proves our main result.

Theorem 3 *Let $\mathcal{P}(t)$ be a monotone decision problem whose value depends on a set S of n objects and a real parameter t. Let*

$$t^* := \max\{t \in \mathbb{R} \cup \{-\infty, \infty\} : \mathcal{P}(t) = \mathit{false}\}.$$

Let A (resp. B) be a sequential (resp. parallel) algorithm that, given as input the set S and any real number t, decides whether $t < t^*$, $t > t^*$ or $t = t^*$, and that satisfies the conditions given in Section 3. Let $T_A(n)$ denote the running time of A, and let $T_B(n)$ (resp. $P(n)$) denote the parallel time of (resp. number of processors used by) algorithm B. Then t^* can be computed in time

$$O(T_B(n)P(n)\log P(n) + T_B(n)T_A(n)\log P(n)).$$

Corollary 1 *Problem 1 can be solved in $O(n\log^3 n)$ time and using $O(n)$ space.*

Proof. Algorithm A is given by Lemma 3. We have $T_A(n) = O(n\log n)$. Combining the proof of Lemma 3 with the results of the previous chapter of this course shows that there is a CREW PRAM algorithm B that decides whether $t < t^*$, $t > t^*$ or $t = t^*$, in parallel time $T_B(n) = O(\log n)$, using $P(n) = O(n)$ processors. Then the time bound for solving Problem 1 immediately follows from Theorem 3. The space bound can be verified by carefully checking the details of the algorithm. ∎

Hence, the improved version of parametric search reduces the running time for solving Problem 1 from $O(n^2\log^2 n)$ to $O(n\log^3 n)$. This is an improvement of almost a linear factor.

6 Concluding remarks

It will be clear that the general technique is very difficult to implement. Also, the constant factors that are hidden in the Big-Oh are probably quite large. When I wrote a first version of these notes in 1995, I had the impression that implementing the algorithm for solving Problem 1—by not trying to be general, but by adapting all details to this special case—may actually give a reasonably efficient program. Here are some reasons why I thought this should be true: First, the sequential algorithm A is very simple. Second, the parallel algorithm is quite easy, and it is not difficult at all to simulate it sequentially. Finally, as we already saw in Section 3, the comparison steps that have to be made all lead to polynomials of degree one or two[6]. For such

[6]This is not quite true: When computing upper tangents during the parallel convex hull algorithm, there is one case, in which the degree is three. This case, however, can be avoided.

polynomials, we can immediately write down the roots explicitly.

Jörg Schwerdt implemented a version of the parametric search algorithm for solving Problem 1. And indeed, it is fast in practice: about 100 seconds on inputs of 3,000 moving points (this was in 1996). For further details, see

- J. Schwerdt. *Das Diameterproblem einer bewegten Punktemenge: eine Implementierung mit Hilfe von parametric search.* Master's Thesis, University of the Saarland, 1996. See:

  `http://isgwww.cs.uni-magdeburg.de/~schwerdt/Welcome.html`

- J. Schwerdt, M. Smid and S. Schirra. *Computing the minimum diameter for moving points: an exact implementation using parametric search.* Proc. 13th Annual ACM Symp. Comput. Geom, 1997, pp. 466–468.

- You can find the program at the following web page:

  `http://www.mpi-sb.mpg.de/LEDA/friends/paramsearch.html`