# Maximizing Non-Linear Concave Functions in Fixed Dimension[1]

Sivan Toledo
*Laboratory for Computer Science*
*Massachusetts Institute of Technology*
*Cambridge, Massachusetts 02139*

**Abstract**

Consider a convex set $\mathcal{P}$ in $\mathbb{R}^d$ and a piecewise polynomial concave function $F: \mathcal{P} \to \mathbb{R}$. Let $\mathcal{A}$ be an algorithm that given a point $x \in \mathbb{R}^d$ computes $F(x)$ if $x \in \mathcal{P}$, or returns a concave polynomial $p$ such that $p(x) < 0$ but for any $y \in \mathcal{P}$, $p(y) \geq 0$. We assume that $d$ is fixed and that all comparisons in $\mathcal{A}$ depend on the sign of polynomial functions of the input point. We show that under these conditions, one can find $\max_{\mathcal{P}} F$ in time which is polynomial in the number of arithmetic operations of $\mathcal{A}$. Using our method we give the first strongly polynomial algorithms for many *non-linear* parametric problems in fixed dimension, such as the parametric max flow problem, the parametric minimum $s$-$t$ distance, the parametric spanning tree problem and other problems. We also present an efficient algorithm for a very general convex programming problem in fixed dimension.

**Keywords**: Convex programming, parametric searching, parametric optimization, network flow.

# 1 Introduction

Consider a convex set $\mathcal{P}$ in $\mathbb{R}^d$ and a piecewise polynomial concave function $F: \mathcal{P} \to \mathbb{R}$. Let $\mathcal{A}$ be an algorithm that given a point $x \in \mathbb{R}^d$ computes $F(x)$ if $x \in \mathcal{P}$, or returns a *separation polynomial*, a concave polynomial $p$ such that $p(x) < 0$ but for any $y \in \mathcal{P}$, $p(y) \geq 0$. We assume that $d$ is fixed and that all comparisons in $\mathcal{A}$ depend on the sign of polynomial functions of the input point which are called *comparison polynomials*. Our main result is that under these conditions, one can find $\max_{\mathcal{P}} F$ and a maximizer in time which is polynomial in the number of arithmetic operations of $\mathcal{A}$.

[1] An extended abstract of this paper appeared in [15].

The algorithm is based on the ingenious parametric searching technique invented by Megiddo [12]. This technique can be directly applied to one dimensional concave maximization problems, that is, when the domain of $F$ is an interval on the real line. The technique, which is described in detail in Section 2, can be summarized as follows. We simulate the execution of $\mathcal{A}$ on a maximizer $x^\star$ even tough we do not know the location of $x^\star$. Therefore we handle it in the algorithm as a symbolic constant. When we need to determine the sign of some polynomial $p$ at $x^\star$, we compute the roots of $p$. Since $F$ is concave, it is fairly easy to determine the location of the maximum of $F$ with respect to each root. By determining between which two roots $x^\star$ lies, we can determine the sign of $p(x^\star)$. In other words, we compute a decomposition of $\mathbb{R}$ such that in every cell the sign of $p$ does not change, and determine which cell of the decomposition contains a maximizer of $F$.

Cohen and Megiddo [4, 5] and independently Norton, Plotkin and Tardos [14] generalized the technique to handle the multidimensional case, but only when $F$ is piecewise linear and the separation and comparison polynomials are all affine. When a comparison polynomial $p$ is affine, a space decomposition which is invariant for the sign of $p$ is a hyperplane $H$ and two open half spaces. Such a hyperplane is called a *critical* hyperplane. The restriction of $F$ to $H$ is a concave function in one dimension lower. By induction, the maximum of $F$ on $H$ can be found. The assumption that the comparison polynomials are affine also makes it relatively easy to explore the neighborhood of $H$, and to determine on which side of it $F$ is increasing, thereby resolving the comparison. Since it is assumed that $F$ is piecewise linear, there is a maximizer which is a vertex of its graph. Such a maximizer can be found using linear programming.

In the non-linear case the problem of resolving comparisons becomes much harder. The comparison polynomials are not necessarily concave, and it is hard to compute a sign invariant space decomposition and to locate the cell in the decomposition which contain the maximizer. Our algorithm computes the decomposition and locates the cell, but we prefer to describe it in a slightly different form which is more amenable to recursive breakdown. We use a searching technique which is based on the weighted Euclidean 1-center algorithm of Megiddo [13]. In this technique, the $d$ dimensional algorithm works by simulating the $d-1$ dimensional one on a hyperplane that contains a maximizer of $F$. Three new tools are required in order to apply this searching technique to general concave maximization problems. The first is a sign invariant space decomposition which will enable us to simulate the algorithm in one dimension lower. The decomposition technique we use is Cylindrical Algebraic Decomposition [3]. We use *critical hyperplanes* to locate a maximizer, so we need a way to decide on which side of a hyperplane there is a maximizer. In Section 2.1 we describe a very general algorithm for doing so, which is based on Fibonacci search. Finally, we use Lagrange multipliers to find a maximizer.

Using our new technique we obtain the first strongly polynomial algorithms to a wide variety of non-linear parametric problems in fixed dimension. For example,

given a graph in which the edges have concave polynomial weights, we can maximize the max-flow in the graph, the minimum spanning tree, the minimum $s$-$t$ distance and so on. As is the case with other applications of Megiddo's parametric search technique, when there is a fast parallel algorithm for evaluating $F$, a fast parallel algorithm for maximizing $F$ can be obtained, and the efficiency of the sequential maximization algorithm can be greatly improved. Using this improvement, we obtain an efficient algorithm for optimizing a linear function under convex polynomial constraints in fixed dimension. Our running time analysis is in the Random Access Machine model [2].

Megiddo's parametric searching technique is a *lifting* transformation. An algorithm for evaluating a function $F$ is simulated, and the point in which the function is evaluated is handled as a symbolic constant $\lambda$. Megiddo's lifting technique applies to a very specific class of algorithms. Variables that are functions of $\lambda$ are assumed to be polynomial functions of $\lambda$, and conditional statements involving $\lambda$ are assumed to depend on the sign of such variables. Unfortunately, parametric searching algorithms do not belong to this class, since they also find roots of polynomials. This poses a difficulty in trying to lift such an algorithm, in order to generate a two dimensional searching algorithm. Our algorithm can be viewed as an extension of the Megiddo's technique to algorithms that are also allowed to find roots of polynomials. Interestingly, Megiddo's technique has been applied to many algorithms that perform polynomial root finding, especially in geometric optimization (see for example [1]). The implicit assumption in such applications is that the algorithm, which includes root finding, is implemented using symbolic algebra tools that allow computation in algebraic extensions and eliminate the need to handle algebraic numbers explicitly. If so implemented, the algorithm does not perform root finding and can be lifted using Megiddo's technique. In this paper we show explicitly how to lift an algorithm that performs root finding.

The next section describes in detail the application of Megiddo's technique to concave maximization problems. The general case is presented in Section 4. The two dimensional case is presented separately in Section 3 in order to provide a full description of an easy to visulize case. We conclude the paper in Section 5 with several applications of our results.

# 2 Maximizing One Dimensional Concave Functions

We begin with a brief review of Megiddo's parametric search technique [12] and how to use it to solve parametric maximization problems. We first define the class of algorithms that can be used as evaluators of $F$.

**Definition** Let $\mathcal{A}$ be an algorithm that gets as a part of its input a point $x \in \mathbb{R}^d$, and returns a real number depending on $x$, denoted $F(x)$. We say that $\mathcal{A}$ is *polynomial in $x$ with degree $\delta$* if the only dependencies on $x$ are:

1. $\mathcal{A}$ is allowed to evaluate the polynomials $p_1(x), \ldots, p_k(x)$ of degree at most $\delta$, where $\delta$ does not depend on the input.

2. The only operations on variables in $\mathcal{A}$ that depend on $x$ are addition of such variables, addition of constants, and multiplication by constants.

3. The conditional branches in $\mathcal{A}$ that depend on $x$ depend only on signs of variables that depend on $x$.

**Definition** A point $x_0 \in \mathcal{P}$ is called a *non-singular point of* $F$ if there is an $\epsilon > 0$ such that the restriction of $F$ to $\mathcal{P} \cap \{x : |x - x_0| < \epsilon\}$ is a polynomial function. If $x_0$ is a non-singular point of $F$, this restriction of $F$ is called the *piece* of $F$ at $x_0$.

**Corollary 2.1** *If $\mathcal{A}$ is polynomial in $x$ with degree $\delta$, then all the variables in $\mathcal{A}$ that depend on $x$ contain polynomials of degree of at most $\delta$, and $F(x)$ is piecewise polynomial whose pieces are polynomials of degree at most $\delta$.*

Assume that we have an efficient algorithm $\mathcal{A}$ for evaluating $F(x)$, which is polynomial in $x$ with some fixed degree $\delta$, and that $F$ is a concave function. Megiddo's main idea is to simulate $\mathcal{A}$ at a maximizer of $F$, denoted $x^\star$. As long as no comparisons are made, that is, no conditional branches that depend on the input point are to be executed, it is easy to simulate the algorithm, by treating the variables as polynomials and performing polynomial arithmetic. How do we resolve a conditional branch that depend on the sign of a variable? We find the roots of the polynomial stored in that variable, and locate $x^\star$ among them as follows. We evaluate $F$ at each of the roots, and determine the location of a maximizer with respect to each of the roots. (For now we assume that if we can evaluate $F$ at a point we can also decide the direction to $x^\star$, and in section 2.1 we justify this assumption.) In other words, for every root, we test whether it is $x^\star$, or else whether $x^\star$ is to its left or to its right. Given this information, we can easily decide which way should the branch take, since the sign of a polynomial is constant between its roots. We thus obtain a smaller and smaller interval that is known to contain $x^\star$, and finally the algorithm terminates. In section 2.1 we show how to obtain at this stage a maximizer of $F$ and the two pieces of $F$ to its left and right. These pieces allow us to generalize the algorithm to higher dimensions, and they provide a certificate of optimality for the maximizer.

In more abstract terms, given a comparison polynomial $p$, we decompose the space (here $\mathbb{R}$) into cells which are invariant for the sign of $p$. In the one dimensional case, the cells are points, which are the roots of $p$, and open intervals. Given this decomposition, we decide in which cell there is a maximizer of $F$, and thus resolve the comparison.

**Running time analysis.** Assuming that the algorithm $\mathcal{A}$ runs in $T_0$ time, the one dimensional maximization algorithm runs in time $T_1 = O(T_0^2)$, since whenever the algorithm makes a comparison, we evaluate the function at each of the roots. Megiddo [12] noticed that if we also have a parallel algorithm that evaluates the function, we can exploit the parallelism to obtain faster maximization algorithm. Assume that the parallel algorithm uses $P$ processors and runs in $T_p$ parallel time. We simulate the algorithm sequentially. In each parallel step there are at most $P$ independent comparisons. Instead of evaluating the function at each of the roots of all the associated polynomials, we perform a binary search over the set of $O(P)$ roots to locate $x^\star$ among them. This results in $O(\log P)$ evaluations of $F$, and $O(P)$ overhead for performing the binary search by repeatedly finding the median of the set of unresolved roots. Having done this, we can determine the sign of each of the $O(P)$ roots at $x^\star$ and proceed to the next parallel step. The total cost of this procedure is $T_1 = O(PT_p + T_0 T_p \log P)$. Since we only require that comparisons will be made in parallel, we can use Valiant's weak model of parallel computation [16].

## 2.1 Where is $F$ Maximized?

Given a point $x_1$, we need to determine the location of $x^\star$ relative to $x_1$. The techniques for doing so in one dimension and the techniques that were used by [4, 5, 14] do not seem to generalize to non-linear comparison polynomials and higher dimensions. This section describes a new technique which is easy to generalize. We evaluate $F(x_1)$. If we have previously encountered a point $x_0$ such that $F(x_0) \geq F(x_1)$, we can safely assume that there is a maximizer in the direction of $x_0$. Otherwise, we do not resolve the comparison. We duplicate the state of the simulated algorithm, and in one copy resolve the comparison as if there is a maximizer to the left of $x_1$, and in the other copy as if there is a maximizer to the right of $x_1$. We run those two copies in parallel (by interleaving their execution on a sequential machine). For each root of a comparison polynomial we obtain (from either copies; we do not know which one of them is correct), we evaluate $F$ at that point. As long as we do not encounter a value of $F$ larger than $F(x_1)$, we can determine which side of a given root contains a maximizer. If we run into a point $x_2$ where the value of $F$ is larger than $F(x_1)$, we again will not be able to resolve the comparison. But in this case, the maximizer is on the same side of $x_1$ as $x_2$, so now we can resolve the comparison that involved $x_1$. In particular, we can decide which copy of the algorithm was given the correct answer and discard the other. Of course, we now must run two copies of the algorithm in which we resolve the comparison involving $x_2$ in different ways. There are always two copies of the algorithm executing. Eventually, both of our copies will terminate. Each one of them returns $F(x)$ as a polynomial. One of them corresponds to the piece of $F$ to the right of the point $x_k$ with the highest $F$ value encountered, and the other corresponds to the piece of $F$ to the left of this point. We maximize these polynomials over the corresponding intervals. If one of them attains a maximum higher than the

other inside its interval, this is the optimum, and this polynomial is the piece of $F$ on both sides of the maximum. Otherwise, they both attain the same maximum, and in that case the point $x_k$ is a maximizer, and these two polynomials are the pieces of $F$ on its two sides. Since in most cases the cost of evaluating $F$ dominates the cost of duplicating the state of the algorithm we ignore this cost in the running time analysis.

The same idea works in any dimension. Let $F\colon \mathbb{R}^d \to \mathbb{R}$ be a concave function. Suppose that we already know the value of $F$ at some points in $\mathbb{R}^d$, and that the highest value we computed is $F(x_0)$. Given a hyperplane $H$, let the maximum of $F$ on $H$ be $F(x_1)$. If $F(x_0) \geq F(x_1)$, we can safely assume that there is a maximizer in the direction of $x_0$. Otherwise there is another point $x$ on the other side of $H$ with $F(x) > F(x_1)$. Hence at the intersection of the line segment $\overline{x\,x_1}$ with $H$ the value of $F$ must be higher than $F(x_1)$ due to the concavity of $F$, a contradiction.

## 2.2 Finding a Feasible Point

In many cases the domain $\mathcal{P}$ is either all of $\mathbb{R}^d$, or easy to compute as the intersection of a small number of constraints $p_i(x) \geq 0$, where the $p_i$'s are concave polynomials, such as in the parametric max flow problem. But there are cases in which the domain of $F$ is defined by an exponential number of constraints, such as the parametric minimum $s$-$t$ distance in a connected graph. Using ideas from [4, 5, 14], we describe how to deal with this problem in the non-linear case.

We assume that there is an algorithm $\mathcal{A}_f$ for testing whether a point $x$ belongs to $\mathcal{P}$, which either declares that $x \in \mathcal{P}$, or declares that $x \notin \mathcal{P}$ and provides in addition a violated constraint $p(x) < 0$, where $p$ is a concave polynomial. We use this algorithm to either find a point $x_f$ in $\mathcal{P}$ or decide that $\mathcal{P}$ is empty. If $\mathcal{P}$ is empty we report this and halt. Otherwise, given a critical point $x_0$, we test whether $x_0 \in \mathcal{P}$, and if not, we know that there is a maximizer of $F$ in the direction of $x_f$.

We simulate the feasibility testing algorithm $\mathcal{A}_f$ on $x_f$. During the simulation, we maintain an interval $[a, b]$, which is known to contain $\mathcal{P}$. In addition, for each endpoint $z \in \{a, b\}$ of the interval, if it is finite, we also maintain a constraint $p_z$ that is violated if we pass this endpoint. (We begin with the interval $(-\infty, \infty)$.) When we must resolve a comparison, we find the roots of the comparison, and determine whether one of them is a feasible point, in which case we return this point and halt. If a given root being tested is not in $\mathcal{P}$, a violated constraint $p$ is returned. Since for each $x \in \mathcal{P}$, $p(x) \geq 0$, we know that $\mathcal{P}$ must lie in $[a, b]$ and also in the interval $\{x : p(x) \geq 0\}$. We therefore update $[a', b'] = [a, b] \cap \{x : p(x) \geq 0\}$. If this new interval is empty, we conclude that $\mathcal{P}$ is empty. Note that this event actually carries more information, since if we assume without loss of generality that $\{x : p(x) \geq 0\}$ is to the left of $[a, b]$, then the two constraints $p$ and $p_a$ provide a certificate that $\mathcal{P}$ is empty. If the new interval in not empty and not equal to $[a, b]$, we replace the polynomials associated with the updated endpoints with $p$. It is easy to see that if we

have a parallel feasibility testing algorithm, we can exploit the parallelism and obtain a faster algorithm using Megiddo's scheme.

If at no point of the simulation the feasible interval becomes empty, then our simulated algorithm terminates, and returns an answer. In addition, we have an interval $[\hat{a}, \hat{b}]$ where $\mathcal{P}$ must lie. If the algorithm returns "yes", it means that every point in $[\hat{a}, \hat{b}]$ is a feasible point. Otherwise, it returns "no" and a violated constraint $p$. It follows that this constraint is violated for all points of $[\hat{a}, \hat{b}]$, so this constraint together with either $p_{\hat{a}}$ or $p_{\hat{b}}$ provide proof of emptiness for $\mathcal{P}$.

# 3 A Two Dimensional Algorithm

We now describe the parametric maximization algorithm in two dimensions, that is, when $F: \mathcal{P} \to \mathbb{R}$ where $\mathcal{P} \subseteq \mathbb{R}^2$. Let $(x^\star, y^\star)$ be a maximizer of $F$. The main idea of the algorithm is to simulate the one dimensional algorithm on $F$ restricted to a line $x = x^\star$. Since a concave function restricted to a line (or a hyperplane in higher dimensions) is still concave, the problem of maximizing $F$ restricted to a line is a one dimensional problem. If we can simulate the one-dimensional algorithm on such a line, we can find a maximizer $y^\star$ on the line, which is also a global maximizer, and we are done. The problem of course is how to make decisions during the simulation. Let $p$ be a comparison polynomial in the simulated non-parametric algorithm. We compute a cylindrical decomposition of $\mathbb{R}^2$ which is invariant for the sign of $p$. This decomposition is constructed by computing the self intersections of the curve $p = 0$ and the points of vertical tangency of the curve. Those points are projected on the $x$-axis, and the plane is decomposed into vertical slabs between those points. A vertical slab (which is a generalized cylinder) may intersect the curve $p = 0$, but the roots of $p$ do not intersect each other inside the slab (see Figure 1).

We execute the one dimensional algorithm on the vertical lines that decompose the plane into slabs, and we decide in which slab there is a maximizer (recall that in Section 2.1 we have shown that if we can maximize $F$ on a line, we can also decide on which of its sides there is a maximizer).

The crucial point is that in each slab there is a constant number of roots to the polynomial $p(x, y)$ as a one dimensional polynomial in $y$. The location of those roots depends on $x$, but the dependency is continuous. Hence we can simulate the one dimensional algorithm in a consistent manner. However, the one dimensional algorithm executes the non-parametric algorithm on the roots of $p$. We cannot perform this directly, since the location of the roots depend on $x$. However, we can simulate the non-parametric algorithm. When the non-parametric algorithm performs a comparison involving a polynomial $q$, we compute a cylindrical decomposition which is invariant for both $p$ and $q$. Again, we determine the slab in this decomposition that contains a maximizer. We examine the root of $p$ on which we simulated the non-parametric algorithm in this slab, and determine the sign of $q$ in that cell of the decomposition, which is possible since the decomposition is invariant for the sign of

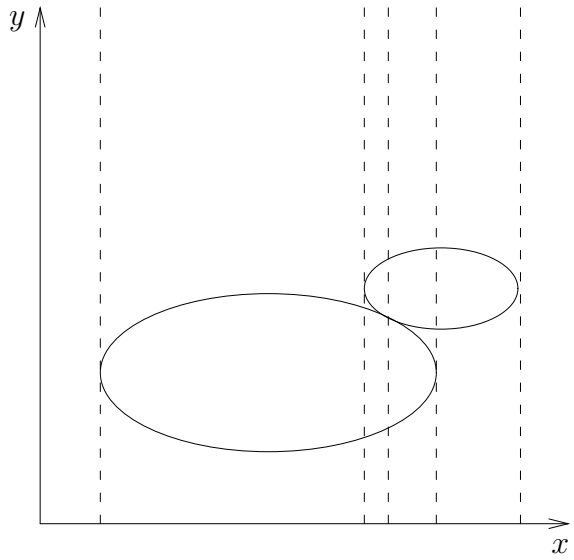


Figure 1: A cylindrical decomposition of $\mathbb{R}^2$ invariant for the sign of a polynomial. The solid curve is the root of the polynomial.

$q$.

Once all the executions of the non-parametric algorithm terminate, we must compare the returned values, which are polynomials, to each other in order to decide which one is higher. To compare a value $r_1(x, y)$ on a root $y_1(x)$ of $p$ with a value $r_2(x_y)$ on a root $y_2(x)$, we compute (again using cylindrical decomposition) the $x$-coordinates of the intersections of $\{r_1(x, y_1) = r_2(x, y_2), p(x, y_1) = p(x, y_2) = 0\}$, and decide in which slab there is a maximizer. Using this information, we can compare $r_1$ to $r_2$. Using a similar approach we can also find the maximum of a polynomial in a certain interval, which is required in the technique of Section 2.1. Finally the simulated one dimensional algorithm terminates. In fact, two copies of it terminate, one which is a simulation to the left of the vertical line $l$ on which the highest value of $F$ was found, and the other to its right. Each returns a curve $p_i(x, y) = 0$, $i \in \{L, R\}$ on which the maximum is obtained, and the two pieces of $F$ above and below this curve, $p_{i,A}$ and $p_{i,B}$. We find the maximum of $F$ to the left of the line $L$ by solving

$$\begin{aligned} &\max\ p_{L,A}(x, y) \\ &p_{L,A}(x, y) = p_{L,B}(x, y) \end{aligned}$$

using Lagrange multipliers. All the functions involved are polynomial, so this can be solved using cylindrical decomposition (since the problem reduces to finding all the solutions to a system of polynomial equations). This method establishes the global maximum, and in addition generates four pieces of $F$ that prove that the point found is indeed a maximizer. Using Helly's theorem, it is easy to show that we can reduce

the number of these pieces to three.

When there exist a parallel non parametric algorithm, we can obtain a parallel one dimensional algorithm. To use it, we also need to sort the list of roots that are obtained in each step of the one dimensional algorithm. We could do it by constructing the cylindrical decomposition invariant for the signs of *all* the comparison polynomials, but this would be too expensive. Instead, we simulate a sorting algorithm, and whenever it compares two roots, we construct the decompositions invariant for the signs of both polynomials, locate the slab that contains a maximizer, and test which of our two roots is higher in that slab. If we use Megiddo's technique and sort the roots using a simulation of a parallel sorting algorithm such as Cole's parallel merge sort [7], the number of calls to the one dimensional algorithm during the sorting algorithm will be only $O(\log^2 P)$, where $P$ is the number of roots we have to sort.

**Running time analysis.** We first note that the cost of constructing the cylindrical algebraic decomposition of a constant number of polynomials of bounded degree in fixed dimension is only a constant (in fact, the cost is polynomial in the number of polynomials, the degree, and the binary encoding of the coefficients, but not in the dimension). Denoting the running time of the non-parametric algorithm $\mathcal{A}$ by $T_0$ and of the $d$-dimensional algorithm by $T_d$, the running time is $T_2 = O(T_0(T_1 + T_0T_1)) = O(T_0^4)$. If there exists a parallel non-parametric algorithm that runs in $T_p$ parallel time and uses $P$ processors, we can again improve the running time. The total running time is $O(PT_p + T_p(\log PT_1 + \log^2 PT_1 + \log PT_p \log PT_1))$ and since $T_1 = O(PT_p + T_p \log PT_0)$, the running time is $T_2 = O(T_0(T_p \log P)^3)$. The breakdown of the running time into terms is as follows. In each parallel step of the non-parametric algorithm $P$ comparison polynomials are generated and a decomposition invariant for each one of them is constructed. Then the one dimensional algorithm is called $\log P$ times to locate the slab in which there is a maximizer in the combined list of critical values. As explained before, sorting all the $y$ critical values requires $\log^2$ more calls to the one dimensional algorithm. Finally a binary search is performed using a simulated non-parametric algorithm. It is simulated on $\log P$ $y$ roots. On each of them the algorithm proceeds in $T_p$ steps, and in each one $P$ comparison polynomials are generated, each requiring a constant number of calls to the one dimension algorithm.

## 3.1 Finding a Feasible Point

We now extend the technique of section 2.2 to two dimensions; the same technique works in any dimension. We use the technique we have just described for simulating the one dimensional feasibility testing algorithm on a vertical line that intersects $\mathcal{P}$. All we need to show is how to decide on what side of a given line $\mathcal{P}$ lies, since if $\mathcal{P}$ intersects the line, the one dimensional algorithm will detect this.

Suppose we are given a line in the plane, and must test on which of its sides $\mathcal{P}$ lies. Since the one dimensional algorithm determines that the line does not intersect

$\mathcal{P}$, it reports two constraints that are contradictory on that line, $p$ and $q$. Since we simulate the one dimensional algorithm and perform its arithmetic on polynomials, the violated constraints are polynomials in $x$ and $y$. Since for any $(x, y) \in \mathcal{P}$ both $p(x, y) \geq 0$ and $q(x, y) \geq 0$, we find a point in the intersection of $p(x, y) \geq 0$ and $q(x, y) \geq 0$. If there is such a point $x_0$ and $\mathcal{P}$ is not empty, then $\mathcal{P}$ lies on the same side of the line as $x_0$, and we maintain $p$ and $q$ as a certificate for this fact. If $p(x, y) \geq 0 \cap q(x, y) \geq 0 = \emptyset$, then $\mathcal{P}$ is empty.

If the simulated algorithm terminates without either finding a point in $\mathcal{P}$, or deciding that $\mathcal{P}$ is empty, we have two cases. The answer it gives is valid for any line $x = x_0$ we might run it on, as long as $x_0$ is in the interval $[x_a, x_b]$ which is known to contain $\mathcal{P}$. When the simulated algorithm terminates, if it returns a point in $\mathcal{P}$, we are done. Otherwise it declares that the intersection of $\mathcal{P}$ with any vertical line $x = x_0$ in the vertical slab which is known to contain $\mathcal{P}$ is empty, and supplies a pair of constraints $p$ and $q$ as a certificate. We compute a point in $p(x, y) \geq 0 \cap q(x, y) \geq$. If there is no such point, then $p$ and $q$ are a proof that $\mathcal{P}$ is empty. Otherwise, if they intersect above to the left of the slab for example (i.e. to the left of $x_a$), then $p$ and $q$ together with the two constraints that assert that $\mathcal{P}$ is to the right of $x_a$, provide a proof that $\mathcal{P}$ is empty. Those four contradictory constraints allow us to generalize the algorithm to higher dimension, in the same way we used the one dimensional certificates of emptiness for constructing the two dimensional algorithm in this section. The number of constraints is a certificate of emptiness can be brought down to at most $d + 1$ in dimension $d$, by Helly's Theorem.

# 4 The General Algorithm

Before we describe the algorithm and prove its correctness, we need some definitions and lemmas.

**Definition** A *semi-algebraic cell* of $\mathbb{R}^d$ is a set of points satisfying a finite set of polynomial equalities and inequalities.

**Definition** A *semi-algebraic variety* is either a semi-algebraic cell, or one of the sets $A \cap B$, $A \cup B$ and $A \setminus B$, where $A$ and $B$ are two semi-algebraic varieties.

**Definition** A *decomposition* of $\mathbb{R}^d$ is the representation of $\mathbb{R}^d$ as the union of a finite number of disjoint and connected semi-algebraic varieties.

**Definition** A decomposition of $\mathbb{R}^d$ is *invariant for the signs* of a family of polynomials if, over each cell of the decomposition, each polynomial is always positive, always negative, or always zero.

**Definition** A decomposition $D_d$ of $\mathbb{R}^d$, that is $\mathbb{R}^d = E_1 \cup \cdots \cup E_N$ is *cylindrical* if $n = 0$ (the trivial case) or if $n > 0$ and:

1. $\mathbb{R}^{d-1}$ has a cylindrical decomposition $D_{d-1}$ which can be written $\mathbb{R}^{d-1} = F_1 \cup \cdots \cup F_M$, and

2. For each cell $E_i$ of $D_d$ there is a cell $F_j$ of $D_{d-1}$ such that $E_i$ can be written in one of the following forms

$$\begin{array}{lll} \{(\boldsymbol{x}, y) : \boldsymbol{x} \in F_j \ \wedge \ y < f_k(\boldsymbol{x})\} & (a\ segment) \\ \{(\boldsymbol{x}, y) : \boldsymbol{x} \in F_j \ \wedge \ y = f_k(\boldsymbol{x})\} & (a\ section) \\ \{(\boldsymbol{x}, y) : \boldsymbol{x} \in F_j \ \wedge \ f_k(\boldsymbol{x}) < & \\ \qquad\qquad y < f_l(\boldsymbol{x})\} & (a\ segment) \\ \{(\boldsymbol{x}, y) : \boldsymbol{x} \in F_j \ \wedge \ y > f_k(\boldsymbol{x})\} & (a\ segment) \end{array}$$

   where the $f_k$'s are the solutions of polynomial equations ($\boldsymbol{x}$ denotes $x_1, \ldots, x_{d-1}$ and $y$ denotes $x_d$).

**Theorem 4.1 (Collins)** *There exists an algorithm that computes a cylindrical decomposition of $\mathbb{R}^d$ invariant for the sign of a family of $n$ polynomials. If the polynomials are all of degree $\delta$ or less, and the length of the binary encoding of their coefficients is bounded by $H$, the running time of this algorithm is bounded by*

$$(2\delta)^{2^{2d+8}} n^{2^{d+6}} H^3.$$

**Lemma 4.2** *Let $D_d$ be a cylindrical decomposition of $\mathbb{R}^d$ invariant under a family $P$ of polynomials, and let $H$ be a hyperplane in $\mathbb{R}^d$ specified by $x_1 = \alpha$ for some real $\alpha$. Then the intersection of $D_d$ with $H$ is a cylindrical decomposition of $\mathbb{R}^{d-1}$ (with the natural mapping of $\mathbb{R}^{d-1}$ onto $H$) invariant under the restriction of the polynomials in $P$ to $H$.*

**Proof:** The intersection of $D_d$ with $H$ is obviously a decomposition of $\mathbb{R}^{d-1}$ and invariant under the signs of the family of polynomials. We prove that it is also cylindrical. The proof uses induction on the recursive structure of the cylindrical decomposition. We assume that the intersection of $D_{d-1}$ with $H$ is cylindrical, and we prove that the intersection of $D_d$ with $H$ is cylindrical. The claim is obviously for $d = 1$, because $D_1$ is a decomposition of the $x_1$ axis which is invariant for the sign of some family of polynomials $P_1$. The intersection of $H$ with the $x_1$-axis is only a point, and the decomposition of a point is always cylindrical and invariant for the signs of $P_1$.

We now assume that the claim is true for $D_{d-1}$. Let $C'$ be a cell of the intersection of $D_d$ with $H$, which is the intersection of the cell $C$ with $H$. Let us assume that $C$ is of the form

$$\begin{array}{ll} \{(x_1, x_2, \ldots, x_d) : & (x_1, x_2, \ldots, x_{d-1}) \in F \ \wedge \\ & x_d > f_k(x_1, x_2, \ldots, x_{d-1})\} \end{array}$$

where $F \in D_{d-1}$. Let $F'$ be the intersection of $F$ with $H$. Then $C'$ can be written as

$$\{(\alpha, x_2, \ldots, x_d) : \quad (\alpha, x_2, \ldots, x_{d-1}) \in F' \wedge \\ x_d > f_k(\alpha, x_2, \ldots, x_{d-1})\}.$$

The other cases are similar. We note that the crucial point in the proof is that $H$ is parallel to all the projection axes. ■

**Lemma 4.3** *Using the same notation, let*

$$D_1 = \{-\infty = \alpha_0, \alpha_1, \ldots, \alpha_k, \alpha_{k+1} = \infty\}$$

*(that is the $\alpha$'s are the points in the one dimensional decomposition). Then the intersection of $D_d$ with $H$ depends continuously on $\alpha$ as long as $\alpha_i < \alpha < \alpha_{i+1}$.*

**Proof:** It is obvious that the intersection of $H$ with $D_1$ changes continuously. Let us assume that the intersection of $H$ with $D_d$ changes continuously but that the intersection with $D_{d+1}$ does not. This can only happen if for some $\alpha_i < \alpha < \alpha_{i+1}$ two sections of $D_{d+1}$ intersect, which contradicts the previous lemma. ■

**The algorithm[2].** We construct the algorithm inductively. The induction hypothesis describes the structure and correctness of the $d-1$ dimensional algorithm. We assume that the $d$ dimensional algorithm work by constructing a sequence of cylindrical decompositions (CADs for short) in $\mathbb{R}$ through $\mathbb{R}^{2d}$ of up to $2d+1$ polynomials, and tests the sign of one of the polynomials in various cells of the decomposition. The algorithm returns a maximizer of $F$. The location of the maximizer is returned as a specific zero dimensional cell in a $d$ dimensional CAD of up to $d$ polynomials. The value of the maximizer is returned as a polynomial.

Let us prove that the induction hypothesis holds for the one dimensional parametric searching algorithm. The one dimensional algorithm finds the roots of polynomials. Finding the roots of a polynomial is equivalent to computing a CAD invariant for the sign of it. Then the parametric searching algorithm evaluates the sign of other polynomials on the roots. Finding the sign of a polynomial $q$ at a root of a polynomial $p$ can certainly be done by consructing a CAD invariant for the sign of $p$ and $q$ and evaluating the sign of $q$ at the root of $p$, which is a cell of the CAD. The one dimensional algorithm also compares values of $F$ at various roots. The values of $F$ are all polynomials. Suppose we need to compare the value of $q'$ at a root $\alpha'$ of $p'$ with the value of $q''$ at a root $\alpha''$ of $p''$. In other words, we need to test the sign of the polynomial $q'(\alpha') - q''(\alpha'')$ at a point $(\alpha', \alpha'')$ in which $p'(\alpha') = 0$ and $p''(\alpha'') = 0$. We could certainly do this by constructing the CAD invariant for the sign of $q'(\alpha') - q''(\alpha'')$, $p'(\alpha')$ and $p''(\alpha'')$, and testing the sign of $q'(\alpha') - q''(\alpha'')$ in

[2]The notation $p'$ in this section means some arbitrary polynomial and not the derivative of a function $p$.

some particular cell. This is a CAD of 3 polynomials in $\mathbb{R}^2$. Finally, we need to maximize $F$ over two open intervals in which $F$ does not have a breakpoint. This is done by maximizing two concave polynomials over the intervals, which can be done be finding the roots of their derivatives, which again amounts to computing CADs. We now compare the two maxima using the method just described. The higher is the global maximum. The maximum is returned as a polynomial $p$ at a root $\alpha$ of another polynomial $q$.

Let us describe the $d$ dimensional algorithm. The algorithm works by simulating the $d-1$ dimensional algorithm on a hyperplane $x_1 = x_1^\star$, where $x_1^\star$ is a projection of a maximizer $x^\star = (x_1^\star, \ldots, x_d^\star)$. Suppose the simulated algorithm constructs a CAD $D$ of $n$ polynomials in $\mathbb{R}^m$ and tests the sign of one of the polynomial in some cell of the CAD. The polynomials are of dimension $m+1$. We therefore constructs the CAD $D'$ in $\mathbb{R}^{m+1}$ (we consider the additional variable to be $x_1$). Let the critical values of the decomposition be $\alpha_1, \ldots, \alpha_k$, the roots of some one dimensional polynomial $p(x_1)$. We locate a slab containing a maximizer $\alpha_j < x_1^\star < \alpha_{j+1}$ by performing a binary search (or a Fibonacci search, which would result in a slightly better constant in the running time). To determine whether the slab is to the left or to the right of some critical value $\alpha_i$, we call the $d-1$ dimensional algorithm on the hyperplane $x_1 = \alpha_i$. In the called $d-1$ dimensional algorithm we add to the CADs constructed the polynomial $p(x_1)$. The number of polynomials and the dimension of each CAD are increased by one. We compare the values returned from different calls to the $d-1$ dimensional algorithm, in order to find a slab containing a maximizer. Suppose we need to compare a value $q'(\alpha', x_2, \ldots, x_d)$ on a zero dimensional cell in the CAD of

$$p'_1(\alpha', x_2, \ldots, x_d), \ldots, p'_{d-1}(\alpha', x_2, \ldots, x_d)$$

where $\alpha'$ is some root of $p'(x_1)$, with a value $q''(\alpha'', x_2, \ldots, x_d)$ on a zero dimensional cell in the CAD of

$$p''_1(\alpha'', x_2, \ldots, x_d), \ldots, p''_{d-1}(\alpha'', x_2, \ldots, x_d)$$

where $\alpha''$ is some root of $p''(x_1)$. We construct a CAD invariant for the signs of

$$\begin{array}{l} q'(\alpha', x'_2, \ldots, x'_d) - q''(\alpha'', x''_2, \ldots, x''_d) \\ p'_1(\alpha', x'_2, \ldots, x'_d) \\ \vdots \\ p'_{d-1}(\alpha', x'_2, \ldots, x'_d) \\ p'(x'_1) \\ p''_1(\alpha'', x''_2, \ldots, x''_d) \\ \vdots \\ p''_{d-1}(\alpha'', x''_2, \ldots, x''_d) \\ p''(x''_1). \end{array}$$

This is a CAD in $\mathbb{R}^{2d}$ of $2(d-1)+3 = 2d+1$ polynomials. The space $\mathbb{R}^{2d}$ here is basically the cartesian product of two $d$ dimensional spaces, so we resolve the comparison by testing the sign of

$$q'(\alpha', x'_1, \ldots, x'_d) - q''(\alpha'', x''_1, \ldots, x''_d)$$

on the zero dimensional cell which is the cartesian product of the two cells in $\mathbb{R}^d$ returned by the $d-1$ dimensional algorithm. Now we have a slab in the original CAD which is known to contain a maximizer of $F$. In this slab the intersection of any hyperplane $x_1 = \alpha$ with the CAD $D'$ changes continuously with $\alpha$, so we can determine the sign of any of the polynomials in any of the cells. Since the sign is constant for any $x_1$, the sign equals the sign for $x_1 = x_1^\star$ which is what we need to determine in order to continue the simulation of the $d-1$ dimensional algorithm. When the simulation of the $d-1$ dimension algorithm terminates we end up with two values, one valid in a slab $\alpha_a < x_1 < \alpha_b$ and the other valid in a slab $\alpha_b < x_1 < \alpha_c$. Let us describe how we find the maximum of $F$ in the slab $\alpha_a < x_1 < \alpha_b$. The maximum of $q(x_1, \ldots, x_d)$ on some one dimensional cell $c$ of the CAD of

$$p_1(x_1, x_2, \ldots, x_d), \ldots, p_{d-1}(x_1, x_2, \ldots, x_d)$$

is found using Lagrange multipliers. We need to solve the equations

$$\nabla q - \sum_{i=1}^{d-1} \lambda_i \nabla p_i = 0$$

and

$$p_1(x_1, x_2, \ldots, x_d) = 0, \ldots, p_{d-1}(x_1, x_2, \ldots, x_d) = 0$$

for both $x_1, \ldots, x_d$ and $\lambda_1, \ldots, \lambda_{d-1}$. We can do so by constructing a CAD in $\mathbb{R}^{2d-1}$ with $2d-1$ polynomials. The variables are ordered $x_1, \ldots, x_d, \lambda_1, \ldots, \lambda_{d-1}$. The space $\mathbb{R}^{2d-1}$ is a cartesian product of two spaces, the $x_1, \ldots, x_d$ space and the $\lambda_1, \ldots, \lambda_{d-1}$ space. Our one dimensional cell $c$ is mapped into a $d$ dimensional manifold $c_d$. If there is a point on this manifold in which the sign of the polynomials

$$\nabla q - \sum_{i=1}^{d-1} \lambda_i \nabla p_i$$

is zero, then this is an extremum of $q$ in $c$. If $q$ is constant on $c$ we are done. Otherwise there is an extremum point, and points which are not extrema. Suppose there is a point $x'_1$ which is an extremum, that is, for small enough $\epsilon \neq 0$, $q(x'_1 + \epsilon) < q(x'_1)$ (the notation $q(x)$ here denotes the value of $q$ on a point $x_1 = x$ in $c$). We claim that in this case, $x'_1$ is one of the critical values in the one dimensional decomposition which is the base of the CAD. Assume for contradition that it is not. Consider the intersection of a hyperplane $x_1 = x'_1 + \epsilon$ with the CAD for small $\epsilon$. The intersection

is a CAD which changes continuously. But we know that at $\epsilon = 0$ there is a set of Lagrange multipliers, or a zero for the polynomials in the CAD, where as for any $\epsilon \neq 0$, there isn't, a contradiction. Assuming that $\alpha_a < x_1' < \alpha_b$ is the root of the polynomial $p(x_1)$, the location of a maximizer of $F$ can be defined by some particular zero dimensional cell in the CAD of

$$p(x_1), p_1(x_1, x_2, \ldots, x_d), \ldots, p_{d-1}(x_1, x_2, \ldots, x_d).$$

This concludes our inductive proof.

**The running time.** Let us count the number of CADs constructed by the algorithm. We denote the number of CADs by $C_d$. If the evaluation algorithm $\mathcal{A}$ runs in time $T_0$, it performs no more then $T_0$ comparisons, so $C_0 \leq T_0$. For each CAD performed by the $d-1$ dimensional algorithm, the $d$ dimensional algorithm constructs the same CAD in dimension one higher. If this CAD has $N$ critical values, the $d$ dimensional algorithm calls the $d-1$ dimensional one $\log N$ times during the binary search. From Theorem 4.1 and from the fact that all CADs constructed contain at most $2d+1$ polynomials, we conclude that $N$ is a function of $d$, $N = N(d)$. Each call to the $d-1$ dimensional algorithm constructs $C_{d-1}$ CADs. To find a slab containing a maximizer, $\log N$ comparisons between returned values need to be performed. Every comparison is resolved by constructing a CAD. Finally, two sets of Lagrange multipliers need to be found, which results in the construction of two more CADs. Therefore we have

$$\begin{aligned} C_d &\leq C_{d-1}(C_{d-1} \log N(d) + \log N(d)) + 2 \\ &\leq 2C_{d-1}^2 \log N(d) \ . \end{aligned}$$

Solving the recurrence we obtain

$$C_d \leq (2 \log N(d))^{2^d - 1} C_0^{2^d} = K(d) C_0^{2^d}$$

where $K(d)$ is some constant depending on $d$. Therefore the running time of the $d$ dimensional algorithm is $O(C_0^{2^d}) = O(T_0^{2^d})$.

Let us examine the use of a parallel evaluation algorithm in the construction of a more efficient optimization algorithm. Let as assume that the $d_1$ dimensional algorithm computes $C_{d-1}$ batches of at most $P_{d-1}$ CADs each. If there is an evaluation algorithm that runs in $T_p$ parallel time and uses $P$ processors, we set $C_0 = T_p$ and $P_0 = P$. The $d$ dimensional algorithm simulates the construction of a batch of CADs in the following way. $P_{d-1}$ CADs are constructed, and the combined list of critical values of all the CADs is sorted. Sorting is done using a parallel sorting algorithm, which works in $O(\log N P_{d-1})$ parallel steps. In each step $O(N P_{d-1})$ pairs of roots are compared. A comparison between a root of $p'(x_1)$ and a root of $p''(x_1)$ is resolved by constructing the CAD invariant for the signs of $p'$, $p''$ which totally orders all their roots. Then a binary search is performed over the sorted list. If each CAD

generates $N$ critical values, the $d$ dimensional algorithm calls the $d-1$ dimensional algorithm $\log(NP_{d-1})$ times. Otherwise the algorithm is similar to the case of one CAD per batch (a sequential algorithm). The number of CADs constructed per batch is therefore unchanged, $P_d = P_{d-1}$. The number of batches is

$$C_d \leq C_{d-1}\left(C_{d-1}\log(NP_{d-1}) + \log(NP_{d-1}) + O(\log(NP_{d-1}))\right) + 2.$$

Again, $N = N(d)$, so solving the recurrence we obtain

$$\begin{aligned} C_d &\leq K(d)\left(C_0 \log P_0\right)^{2^d-1} \\ &= K(d)\left(T_p \log P\right)^{2^d-1} \\ &= O\left(\left(T_p \log P\right)^{2^d-1}\right). \end{aligned}$$

This concludes the proof of our main result.

**Theorem 4.4** *Let $\mathcal{A}$ be a polynomial algorithm in $x$ with degree $\delta$, where $x \in \mathbb{R}^d$. Let $F\colon \mathcal{P} \to \mathbb{R}$ be a concave function, and let $\mathcal{P} \subseteq \mathbb{R}^d$ be a convex set. Assume that for any $x \in \mathcal{P}$, $\mathcal{A}(x) = F(x)$, and for any $x \notin \mathcal{P}$, $\mathcal{A}(x) = p_x$ where $p_x$ is a concave polynomial such that $p_x(x) < 0$, but for any $y \in \mathcal{P}$, $p_x(y) \geq 0$. Assume that $\mathcal{A}$ runs in time $T_0$, and that there is an equivalent parallel (in Valiant's model of parallel computation [16]) algorithm $\mathcal{A}_p$ that runs in time $T_p$ and uses $P$ processors. Then there is an algorithm $\mathcal{A}_d$ that runs in time $O(T_0(T_p \log P)^{2^d-1})$ and either decides that $\mathcal{P} = \emptyset$, or decides that $F$ is unbounded on $\mathcal{P}$, or finds the maximum of $F$ on $\mathcal{P}$.*

# 5 Applications

**Convex Programming.** Consider the convex programming problem

$$\begin{array}{l} \text{Minimize } y \\ \text{subject to} \\ y \geq p_1(x_1, \ldots, x_d) \\ \vdots \\ y \geq p_n(x_1, \ldots, x_d) \\ p_1, \ldots, p_n \text{ are convex.} \end{array}$$

The function

$$F(x_1, \ldots, x_d) = \max_i \{p_i(x_1, \ldots, x_d)\}$$

can be evaluated at a point in parallel time $O(\log \log n)$ using $n$ processors [16]. Using Theorem 4.4 we conclude that the above convex programming problem can be solved in time $O(n(logn \log \log n)^{2^d-1})$. Dyer [10, 11] showed how to solve some special cases in $O(n)$ time, but no efficient algorithm was known for the general case.

**Functions defined in terms of graphs.** Let $G = (V, E)$ be a graph, let $s$ and $t$ be two vertices of $G$, and let $W$ be a function mapping edges of $G$ to real numbers. We use the notation $W_e$ to denote $W(e)$. Let $S \subset 2^E$ (for example, the set of all spanning trees, all minimum $s$-$t$ cuts etc.). A *minimization problem* on $G$ is the problem of finding $\min_{s \in S} \sum_{e \in s} W_e$, and usually finding a minimizer is also desirable.

Now assume that $W$ maps edges to concave polynomial functions over some convex set $\mathcal{P} \subseteq \mathbb{R}^d$, instead of to numbers. For every point $x \in \mathcal{P}$, we get an induced minimization problem obtained by mapping every element $e \in E$ to a real number $W_e(x)$. We define a function $F: \mathcal{P} \to \mathbb{R}$ by

$$F(x) = \min_{s \in S} \sum_{e \in s} W_e(x).$$

**Lemma 5.1** *The function $F$ is a concave function.*

**Proof:** Since for all $e \in E$, the function $W_e$ is concave, so is the function $\sum_{e \in s} W_e$ for any subset $s$ of $E$, and therefore the minimum of such functions is also concave. ■

**Lemma 5.2** *If the edge weights $W$ are all polynomial of degree at most $\delta$, then $F$ is a piecewise polynomial function, and its pieces are of degree at most $\delta$.*

**Proof:** Obvious. ■

When $S$ is the collection of all the edge-cuts separating $s$ from $t$ in $G$ and the weights $W_e$ are interpreted as capacities, then the associated minimization problem is the max flow problem in $G$, by the Max-Flow Min-Cut Theorem. The parametric max flow function is hence a concave function on $\mathcal{P} = \cap_{e \in E} \{x : W_e(x) \geq 0\}$. The definition of $\mathcal{P}$ ensures that all the edge capacities are non-negative. Since each $W_e$ is concave, the regions $\{x : W_e(x) \geq 0\}$ are convex, and therefore their intersection $\mathcal{P}$ is convex.

When $S$ is the collection of all paths between $s$ and $t$, the minimization problem is the problem of finding the minimum $s$-$t$ distance. In this case the domain of the parametric function is the convex region $\mathcal{P} = \cap_C \{x : \sum_{e \in C} W_e(x) \geq 0\}$ where the intersection is over all the simple cycles in $G$. The combinatorial complexity of $\mathcal{P}$ may be super-polynomial, but fortunately there is a *separation algorithm* for $\mathcal{P}$. The Bellman-Ford algorithm can be modified so that it either decides that the graph does not contain a negative cycle and finds the shortest path, or finds a negative cycle $C$ (see [8]). Summing the weights of the edges of the cycle as polynomials, we find a concave violated constraint $p(x) = \sum_{e \in C} W_e(x) < 0$ which is *not* violated for any $y \in \mathcal{P}$. Therefore the conditions of Theorem 4.4 are satisfied, and we can find the maxi-min $s$-$t$ distance in strongly polynomial time.

Cohen and Megiddo [4, 6] showed how to solve such problems when the edge weights are affine functions. Again, an algorithm for the general concave polynomial case was not known until now.

# Acknowledgments

Thanks to Pankaj K. Agarwal for reading and commenting on a preliminary version of this paper. Thanks to Esther Jesurum, Mauricio Karchmer, Nimrod Megiddo, Boaz Patt-Shamir and Serge Plotkin for helpful discussions. My research was supported in part by the Defense Advanced Research Projects Agency under Grant N00014–91–J–1698.

# References

[1] P.K. Agarwal, M. Sharir and S. Toledo, Applications of parametric searching in geometric optimization, *Proc. 3nd ACM-SIAM Symp. on Discrete Algorithms*, 1992, 72–82.

[2] A.V. Aho, J.E. Hopcroft and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[3] D.S Arnon, G.E. Collins, S. McCallum, Cylindrical algebraic decomposition I: the basic algorithm, *SIAM J. Comput.* **13** (1984), 865–877.

[4] E. Cohen and N. Megiddo, Strongly polynomial time and NC algorithms for detecting cycles in dynamic graphs, *Proc. 21st ACM Symp. on Theory of Computing*, 1989, 523–534.

[5] E. Cohen and N. Megiddo, Maximizing concave functions in fixed dimension, Research Report RJ 7656 (71103), IBM Almaden Research Center, San Jose, 1990. (Also in this volume.)

[6] E. Cohen and N. Megiddo, Algorithms and complexity analysis for some flow problems, *Proc. 2nd ACM-SIAM Symp. on Discrete Algorithms*, 1991, 120–130.

[7] R. Cole, Parallel merge sort, *SIAM J. Comput.* **17** (1988), 770–785.

[8] T.H. Cormen, C.E. Leiserson and R.L. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, MA, 1990.

[9] J.H. Davenport, Y. Siret and E. Tournier, *Computer Algebra* Academic Press, 1988.

[10] M. Dyer, On a multidimensional search procedure and its application to the Euclidean one-centre problem, *SIAM J. Comput.* **13** (1984), 31–45.

[11] M. Dyer, A class of convex programs with applications to computational geometry, *Proc. 8th ACM Symp. on Computational Geometry*, 1992, 9–15.)

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

[13] N. Megiddo, The weighted Euclidean 1-center problem, *Math. of Operations Research*, **8** (1983), 498–504.

[14] C.H. Norton, S.A. Plotkin and É. Tardos, Using separation algorithms in fixed dimension, *J. of Algorithms*, **13** (1992), 79–98.

[15] S. Toledo, Maximizing non-linear concave functions in fixed dimension, *Proc. 33rd Annual Symp. on Foundations of Computer Science*, 1992, 676–685.

[16] L. Valiant, Parallelism in comparison problems, *SIAM J. Comput.* **4** (1975), 345-348.