Maximizing Non-Linear Concave Functions in Fixed Dimension

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.*

In addition we show that in one dimension, the same result holds even if we only know how to approximate *the value of F. Specifically, if we can obtain an α-approximation for $F(x)$ then we can α-approximate the value of $\max F$. We thus obtain the first polynomial approximation algorithms for many NP-hard problems such as the parametric Euclidean Traveling Salesman Problem.*

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 $\mathcal{A}$. Cohen and Megiddo [3, 4] and independently Norton, Plotkin and Tardos [9] obtained a similar result for the case where F is piecewise linear and the separation and comparison polynomials are all linear.

The technique of [3, 4, 9] extends the parametric search paradigm of Megiddo [7] whose main idea is to simulate $\mathcal{A}$ symbolically on a point $x^\star$ which is a maximizer of F. Since $x^\star$ is not known, a method for resolving comparisons in the simulated algorithm must be provided. Since every comparison is assumed to depend on the sign of some polynomial p in the input point, we could decompose $\mathbb{R}^d$ into cells in which the sign of p is constant. Determining the cell which contains $x^\star$ enables us to determine the sign of $p(x^\star)$ and to resolve the comparison. When a comparison polynomial is linear, the space decomposition is into a hyperplane H and two open half spaces. We call such a hyperplane 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 linear 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 of it which is a vertex of its graph which can be found using linear programming.

In this paper we show how to solve the problem without assuming that F is piecewise linear and that the comparison and separation polynomials are linear. Since in this case it is no longer true that the decomposition of the space into cells which are invariant for the sign of a comparison polynomial consists of a hyperplane and two open half spaces, we cannot use the parametric searching technique directly. Instead we use a searching technique which is based on the weighted Euclidean 1-center algorithm of Megiddo [8]. In this technique, the d dimensional algorithm

*This research was supported in part by the Defense Advanced Research Projects Agency under Grant N00014–91–J–1698. Author's e-mail address: `sivan@theory.lcs.mit.edu`.

works by *simulating* the $d-1$ dimensional one on a hyperplane that contains a maximizer of F. Unfortunately, the parametric searching technique as described by Megiddo and others [7, 3, 4, 9] can use only a certain class of algorithms as the non-parametric algorithm. It turns out that parametric searching algorithms do not belong to this class, since they compute roots of polynomials. Using Collins' cylindrical algebraic decomposition algorithm [1], we extend the class of algorithm which can serve as non-parametric algorithms. This "closes" this class under the operation of paramertization, and enables us to apply the parametric searching technique recursively.

Another consequence of the non-linearity of the comparison polynomials is that it becomes difficult to explore the neighborhood of a critical hyperplane by exploring the behavior of the comparison polynomials in the neighborhood of a point. We therefore develop a new technique to decide on which side of a critical hyperplane F is maximized. This technique is based on the observation that we can avoid resolving some comparisons and take both sides of the associated branch simultaneously. We show that at any time during the execution of our algorithm there can be only one unresolved comparison, which is the one corresponding to the hyperplane on which the highest value of F was computed. The last problem that must be solved is that of finding a maximizer of F when there is no maximizer which is a vertex of the graph of F. This is done by computing certificates of optimality during the execution of the algorithm in lower dimensions, and using Lagrange multipliers to find a maximizer.

Using our new method 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. The generality of our results does not come for free, and the cost is a double exponential dependency on the dimension in the running time of our algorithms, while in the linear case it is singly exponential.

The second result obtained in this paper is the application of Megiddo's technique to the approximation of the maximum of univariate piecewise polynomial concave functions which are NP-hard to evaluate, but for which there is an approximate evaluation algorithm. Specifically, if there is an algorithm $\mathcal{A}$ that on input $x \in [a,b]$ computes a value $F(x) \geq \mathcal{A}(x) \geq \alpha F(x)$ in time T, then we can compute in time $O(T^2)$ a value m that satisfies $\max_{[a,b]} F \geq m \geq \alpha \max_{[a,b]} F$. Since in most cases in which evaluating F is NP-hard finding $\max F$ is also NP-hard, we thus obtain the first polynomial approximation algorithm to many NP-hard problems, such as the parametric Euclidean Traveling Salesman Problem.

The model of computation we use in this paper is one in which all the solutions of a zero-dimensional polynomial ideal can be found in constant time, when the number of generators and variables is constant. It should be stressed that these assumptions are satisfied if one is using symbolic computation, and it can be verified that our algorithms are indeed polynomial both in terms of the number of arithmetic operations performed (which is our main result), and in terms of bitwise operations.

2 Maximizing One Dimensional Concave Functions

We begin with a brief review of Megiddo's parametric search technique [7] and how to use it to solve parametric maximization problems. We first define the class of algorithms we can use 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 δ* 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 δ, where δ 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 1 *If $\mathcal{A}$ is polynomial in x with degree δ, then all the variables in $\mathcal{A}$ that depend on x contain polynomials of degree of at most δ, and $F(x)$ is piecewise polynomial whose pieces are polynomials of degree at most δ.*

Assume that we have an efficient algorithm $\mathcal{A}$ for evaluating $F(x)$, which is polynomial in x with some fixed degree δ, 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 [7] 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 $T_1 = O(PT_p + T_0T_p \log P)$. Since we only require that comparisons will be made in parallel, we can use Valiant's weak model of parallel computation [10].

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 [3, 4, 9] 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 then $F(x_1)$, 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 then 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 then $F(x_1)$ 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 polynomial 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 [3, 4, 9], 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 Extending the Method to Two Dimensions

We now describe the parametric maximization algorithm in two dimensions, that is, when $F\colon \mathcal{P} \to \mathbb{R}$ 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 max-

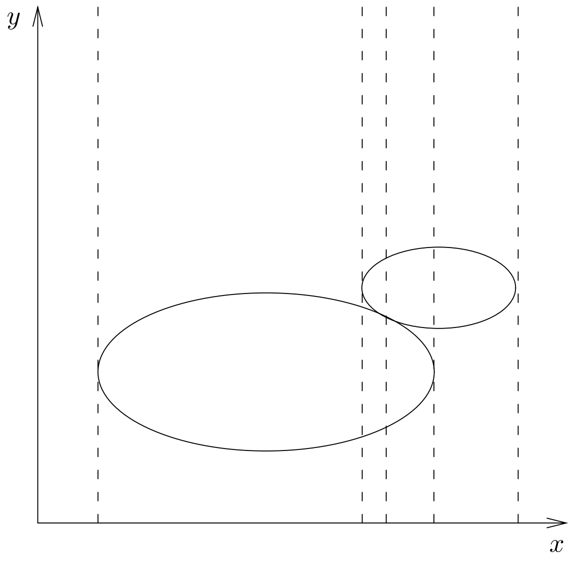

Figure 1: A cylindrical decomposition of $\mathbb{R}^2$ invariant for the sign of a polynomial. The solid curve is the root of the polynomial.

imize F on a line, we can also decide on which of its sides there is a maximizer).

The crucial point now 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 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 highest. 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

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

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.

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 $T_2 = O(T_0(T_p \log P)^3)$. Due to lack of space we omit the details of this improvement.

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 Higher Dimensions

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

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

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 2 (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 δ 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 1 *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 α. 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

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

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

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

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 2 *Using the same notation, let*

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

(that is the α's are the points in the one dimensional decomposition). Then the intersection of D_d with H depends continuously on α 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. ■

Now let us turn our attention to the d-dimensional algorithm. The algorithm has two phases. In the first phase, a series of d-dimensional cylindrical decompositions (CAD's for short) are constructed, each invariant for the signs of up to d polynomials. Once a CAD is constructed, the $(d-1)$ dimensional algorithm is executed on the hyperplanes $x_1 = \alpha_i$, where the α's are the points in D_1, which we call *critical values*. When this phase terminates, the algorithm obtains two values of F, that are the maximum of F on hyperplanes $x_1 = \alpha$; one of the values is valid for $\alpha_a < \alpha < \alpha_b$, and the other for $\alpha_b < \alpha < \alpha_c$. Each of those values, which are polynomials, is supplemented by a family of up to $d+1$ pieces of F which serve as a certificate of optimality. Let us denote those families P_L and P_R. The algorithm maintains the property that there is a maximizer of F, $(x_1^\star, \ldots, x_d^\star)$ with $\alpha_a < x_1^\star < \alpha_c$. During the second phase a global maximizer of F is found by solving

$$\begin{aligned}&\max\ p_{L,1}\\ &p_{L,1} = \cdots = p_{L,k} \quad p_{L,i} \in P_L, k \leq d+1\end{aligned}$$

using Lagrange multipliers, and similarly for P_R. The global maximizer is the highest of the two. If one is higher than the other, the new family P that serves as a certificate for optimality is either P_L or P_R. If the two values are equal, then the new family is a subset of $P_L \cup P_R$. The $d+2$ required polynomials are found by using Helly's Theorem and examining all subsets of $P_L \cup P_R$ of size $d+2$.

The above description is true for the one dimensional algorithm and the two dimensional algorithm that we have already described, and it was already established that these are correct. It is obvious that given a $d+1$ dimensional problem, we can execute the d dimensional algorithm on any hyperplane $x_1 = \alpha$ in $\mathbb{R}^{d+1}$, and the $d-1$ dimensional algorithm on any hyperplane $\{x_1 = \alpha, x_2 = \beta\}$ and so on. Consider an execution of the d dimensional algorithm on $x_1 = \alpha$. If we carry the value α as a symbolic value, we obtain the location of the maximizer and the value of the maximum in terms of α (a value v is implicitly expressed as $p(v, \alpha) = 0$ for some polynomial p). Since the algorithm has only a finite number of execution paths, there must be discrete values of α in which the execution of the d dimensional algorithm changes. The $d+1$ dimensional algorithm simulates the d dimensional one over two adjacent open intervals, such that in each of them the execution path of the d dimensional algorithm does not change, and such that the convex hull of them contains a value $x_1^\star$ which is a projection of a maximizer of F, $(x_1^\star, \ldots, x_{d+1}^\star)$.

We use the fact that given a value α_0, it is possible to decide the relation of α_0 to $x_1^\star$ by executing the d dimensional algorithm on the hyperplane $x_1 = \alpha_0$. There is always one value with respect to which we do not know the location of $x_1^\star$. This is why we end up with two open intervals whose convex hull contains $x_1^\star$. We simulate the d dimensional algorithm. When it constructs a CAD, we construct the CAD in $\mathbb{R}^{d+1}$, and test which slab of $\mathbb{R}^{d+1}$ contains a maximizer. Within this slab, which is induced by the points in the one dimensional decomposition, the intersection of the hyperplane $x_1 = \alpha$ with the CAD is a d dimensional CAD that changes continuously, by the lemmas above. Now the d dimensional algorithm executes the $d-1$ dimensional one on the x_2 critical values of the CAD (since we associate $\mathbb{R}^d$ with the space $(\alpha, x_2, \ldots, x_{d+1})$). However, in the simulation these critical values depend on α, in the sense that they are specific roots of some polynomial $p(x_1, x_2)$. Therefore we simulate the $d-1$ dimensional algorithm, and whenever it constructs a CAD invariant for the signs of some polynomials, we construct a CAD invariant for their signs and for the sign of $p(x_1, x_2)$ in $\mathbb{R}^{d+1}$. We then test in which slab of $\mathbb{R}^{d+1}$ there is a maximizer, and in that slab the $d-1$ dimensional CAD on the root of p changes continuously.

Finally, when the two values of F are obtained, each with a family of polynomials that serve as a certificate for optimality, it is easy to find a global maximizer, by comparing the two values, finding the critical points of x_1 in which they are equal, and deciding in which slab there is a maximizer. In that slab, one of them is higher, so we use its value, and we obtain a real value by maximizing the answer (which is a function of x_1) over the interval of x_1 that is known to contain a maximizer.

Note that it is still true that this algorithm only constructs CAD's in $\mathbb{R}^{d+1}$ and executes the d dimensional algorithm on the critical values of the CAD's. This proves the correctness of the algorithm in any dimension by induction.

The running time. When the $d+1$ algorithm executes, it simulates the d dimensional one, then the $d-1$ dimensional one and so on. For each CAD constructed by one of the simulated algorithm, a CAD in $\mathbb{R}^{d+1}$ is constructed and the d dimensional algorithm is called a constant number of times. Therefore the number of CAD's that are constructed is $C_{d+1} = C_d C_{d-1} \cdots C_1$, and the running time is is $T_{d+1} = T_d C_1^{2^d - 1}$. The solution of this recurrence is $C_d = C_0^{2^{d-1}}$ and $T_d = T_0 C_0^{2^d - 1}$. Again, the running time can be improved when there is a parallel algorithm for evaluating F. We thus obtain the following theorem.

Theorem 3 *Let $\mathcal{A}$ be a polynomial algorithm in x with degree δ, 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, and that there is an equivalent parallel (in Valiant's model of parallel computation [10]) 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 Approximately Maximizing NP-Hard functions

Roughly speaking, up to now we have shown that given a piecewise polynomial concave function, if we can evaluate it then we can also maximize it. However, there are many concave functions of this type that we do not know how to evaluate efficiently, but that we can approximate efficiently. For example, consider the problem of maximizing the Euclidean TSP function over an interval $[a, b]$ in which the distances in the graph satisfy the triangle inequality. The value of Δ-TSP at a point is induced by the lengths of the edges in the graph, $l_e\colon [a, b] \to \mathbb{R}$ for all $e \in E$. In every point of this interval, it is possible to approximately evaluate the length of the minimum TSP tour. In $O(V^3)$ time we can find a tour with a length of at most 3/2 times the length of the minimum tour, using Christofides' algorithm [2]. Since this algorithm is strongly polynomial, the natural question that arises is therefore "can one construct an algorithm for approximating the maximum of Δ-TSP?" (It is obviously NP-Hard to find the exact maximum.) The following theorem implies that the answer is yes.

Theorem 4 *Let $F\colon [a, b] \to \mathbb{R}$ be the minimum of some concave polynomial functions $\{p_i\}_{i \in I}$ of some fixed degree, let $M = \max_{[a,b]} F(x)$, and let $\mathcal{A}$ be an α-approximation algorithm of F. That is, $\mathcal{A}$ is an algorithm which is polynomial in x, that on input $x \in [a, b]$ returns in time T a polynomial p_j, $j \in I$, such that $F(x) \leq p_j(x) \leq \alpha F(x)$. Then a point $x^\star \in [a, b]$ can be found in time $O(T^2)$ such that $M \leq \mathcal{A}(x^\star) \leq \alpha M$.*

Proof: We employ Megiddo's technique, simulating $\mathcal{A}$, performing the arithmetic on polynomials. When we encounter a comparison in $\mathcal{A}$ involving the sign of some polynomial p, we find its roots and run $\mathcal{A}$ on the roots that fall inside $[a, b]$, and on a and b. While running $\mathcal{A}$ on the roots, we keep track of the variables as polynomials, so that we get the results as polynomials p_j. Unfortunately, there is not enough information in these polynomials to tell us between which two roots of p we can find a maximizer of F. However, given a root x_j and a concave polynomial p_j that was returned by $\mathcal{A}$ when executed on x_j, we behave as if the maximizer of F is on the side of x_j where p_j is increasing. If p_j is not increasing on either side, we halt and return $p_j(x_j)$. We keep track of the interval where we assume the maximizer is (replacing perhaps a or b with x_j), and continue to the next root. When the simulated algorithm terminates (in case we do not halt earlier), we report the highest point we have encountered, or the maximum of the polynomial that was returned by the simulated algorithm over the interval that is assumed to contain a maximizer of F, whichever is higher.

The correctness of this procedure follows from the following case analysis. If the final interval *does not* contain a maximizer of F, then we made an error somewhere. Without loss of generality assume that we decided that the maximizer is to the left of x_j, when it is on the right (see Figure 2). Since p_j is non-increasing to the right, $F(x) \leq p_j(x_j) = \mathcal{A}(x_j)$ for any x to the right of x_j. Since the maximizer is to the right, it is one of these x's, so $M \leq \mathcal{A}(x_j)$ and therefore $\mathcal{A}(x_j)$ is the required approximation for M. Otherwise, the final interval contains a maximizer of F, denoted x'. But in this case, $M = F(x') \leq \mathcal{A}(x')$, so we obtain a value which is at least M.

Finally, it is obvious that we cannot find any value $\mathcal{A}(x)$ higher than the desired approximation, otherwise $\alpha M < \mathcal{A}(x) \leq \alpha F(x) \leq \alpha M$, a contradiction. ■

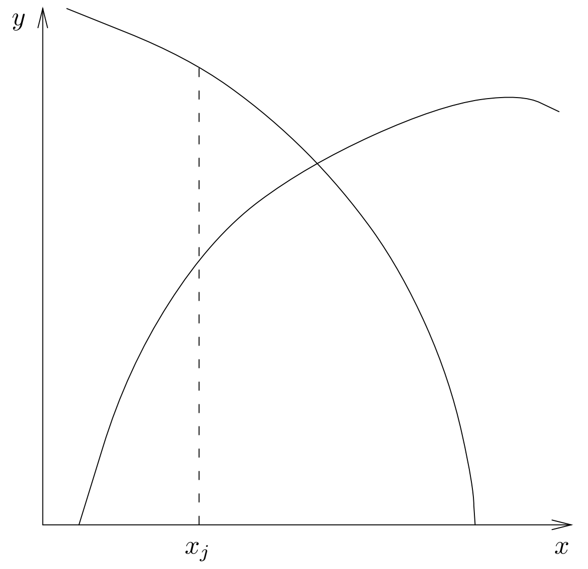

Figure 2: Making a wrong decision and searching to the left of x_j does no harm since if F is maximized to the right of x_j, then we already have an α-approximation of $\max F$.

An important point that should not be overlooked in this scheme is the problem of consistency. While in Megiddo's general framework it is not necessary to maintain the interval in which the optimum is known to be located, it is absolutely necessary to do so in our framework. The reason is that when it is possible to evaluate F exactly, we are always guaranteed to make consistent decisions with respect to where we are simulating the algorithm. But here we might make inconsistent decisions if we do not maintain this interval.

Remark. As usual, when there is also a parallel approximation algorithm, the running time can be improved. For example, using a 2-approximation algorithm for Δ-TSP which is based on minimum spanning trees, it is possible to obtain a 2-approximation for the parametric Δ-TSP in time $O(E \log V + V \log^2 V)$.

6 Applications

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 $F\colon \mathcal{P} \to \mathbb{R}$ by

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

Lemma 3 *The function F is a concave function.*

Lemma 4 *If the edge weights W are all polynomial of degree at most δ, then F is a piecewise polynomial functions, and its pieces are of degree at most δ.*

Some examples. 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 [6]). 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 3 are satisfied, and we can find the maxi-min s-t distance in strongly polynomial time.

Many minimization problems on graphs are NP-hard. For example, when S is the set of all Hamiltonian tours in the graph, the corresponding minimization problem is the Traveling Salesman Problem. If $\mathcal{P}$ is a convex subset of $\mathbb{R}^d$ in which the weights always satisfy the triangle inequality, it is possible to approximate F up to a factor of 3/2 using Christofides' algorithm [2]. Again, it is important to observe that this approximation algorithm is strongly polynomial, and the only operations on weights in it are additions and comparisons between sums of weights. Since it always returns a tour whose length is a polynomial p_j, we can use Theorem 4 to approximate the maximum of the one dimensional parametric problem in time $O(V^6)$.

7 Conclusions and Open Problems

It is interesting to note that it is not the non-linearity of F that makes the problem hard, but the non-linearity of comparison and separation polynomials. We also note that our method of deciding on which side of a hyperplane there is a maximizer of F uses its concavity in a very strong way. We conclude the paper with some open problems that arise from our research.

- While our algorithms are indeed strongly polynomial in any fixed dimension, it is desirable to improve the running times, especially the dependency on the dimension.
- Can one obtain an approximation scheme similar to the one presented in section 5 in higher dimensions?

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.

References

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

[2] N. Christofides, Worst-case analysis of a new heuristic for the Traveling Salesman Problem, Technical Report, GSIA, Carnegie-Mellon University, 1976.

[3] 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.

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

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

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

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

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

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

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