TEL-AVIV UNIVERSITY
RAYMOND AND BEVERLY SACKLER
FACULTY OF EXACT SCIENCES
SCHOOL OF MATHEMATICAL SCIENCES

Extremal Polygon Containment Problems and Other Issues in Parametric Searching

Thesis submitted in partial fulfillment of the requirements
for the M.Sc. degree of Tel-Aviv University

by

Sivan Toledo

The research work for this thesis has been carried out at
Tel-Aviv University
under the direction of Professor Micha Sharir

April 1991

I deeply thank Professor Micha Sharir for his generous help in the preparation of this work. Thanks are also due to the Special Interdisciplinary Program at Tel-Aviv University for supporting my studies.

Contents

Abstract

This thesis studies several issues concerning applications of parametric searching in geometry.

In parametric searching problems, we are typically given a problem whose solution depends on some input and a real parameter. We are only given the input, not the value of the parameter, and we are asked to find a value of the unspecified parameter in which the answer to the problem has some property (it is zero, or maximal etc.).

The first chapter introduces the main techniques of parametric searching, most notably Megiddo's ingenious technique [Me]. We also present other techniques, most of them less general, and other extensions. While most of the material in the Introduction is an exposition of the known techniques (mainly those techniques needed later in the thesis, but not exclusively), it does contain some original material. Specifically, in section 1.3.1 we show how to relax the requirements set by Megiddo when the problem at hand is computing the minimum of a convex function (convex with respect to the parameter). In section 1.6.1 we present a simple argument that shows that in some cases the complex techniques of Megiddo and others can be replaced by a simple randomized procedure with the same (expected) efficiency. In section 1.8.1 we discuss the relation of linear programming to parametric searching.

The main part of this thesis is contained in Chapter 2. Most of the results in this chapter also appear in [To]. In that chapter, several polygon containment problems are solved. Given a convex polygonal object P and an environment Q consisting of polygonal obstacles, we seek a placement for the largest copy of P that does not intersect any of the obstacles, allowing translation, rotation and scaling. We employ the parametric search technique of Megiddo [Me] presented in the introduction. In order to solve this general problem, we also solve the fixed size problem, in which we are only required to decide whether a fixed-size copy of P can be placed in Q. We use an algorithm due to Leven and Sharir [LS, LS1] that finds a superset of placements, not all of which are valid. Our solution runs in time $O(k^2 n \lambda_4(kn) \log^3(kn) \log\log(kn))$ where k is the complexity of P, n is the complexity of Q, and $\lambda_q(r)$ is the maximum length of an (r, q)-Davenport-Schinzel sequence (which is almost linear in r for any fixed q) [ASS, HS].

In addition, we solve some restricted variants of the general case. One

variant in which Q is the interior of a convex polygon and P is not allowed to rotate, is solved in linear time (linear in $n + k$, the sum of complexities of P and Q). Another variant, in which we have to place (disjoint copies of) two convex polygons P_1 and P_2 inside a third convex polygon Q (of complexities k_1, k_2 and n, respectively) and P_1 and P_2 are not allowed to rotate, is solved in time $O((n+k_1+k_2)\log^2(n+k_1+k_2))$. The last variant we consider is placing the largest copy of a triangle under translation, rotation and scaling inside a convex polygon Q, and it is solved in time $O(n^2 \log^2 n)$. These algorithms rely on the algorithms of Chazelle [Ch] and Avnaim and Boissonnat [AB] that solve the corresponding fixed size polygon containment problems.

The last chapter describes some practical experimentation that we have carried out on parametric searching problems. We have coded algorithms that solve two geometric problems using the parametric search technique of Megiddo. Although these programs do not implement the best (and optimal) solutions of these problems, they run fast. Their speed is greatly increased by the use of a very simple heuristic. This heuristic allows even very inefficient versions of the algorithms (almost an order of magnitude slower) to run very fast. It must be added that these conclusions are experimental only and only valid for the input distributions that we have used. In addition to studying the behavior of the algorithms, practice in coding these (rather complex) algorithms was also gained; this is also briefly documented in Chapter 3. This is especially valuable because many parametric searching algorithms tend to be complex, but quite similar to one another.

Chapter 1

Introduction

1.1 What is Parametric Searching ?

This introduction plays two roles in this work. One is to explain the general methods we apply later to some specific problems. The other role is to show the width and depth of this subject. Later, we will use only a small portion of the theory exposed here, but we thought it appropriate to offer a better coverage in the introduction. This is not meant however to be a comprehensive survey. Rather we try to expose the reader to the various topics, and give pointers to the literature where a complete discussion can be found.

While most of the material in this chapter is a survey of published works, some is original material. This includes the material in sections 1.3.1, 1.6.1. Section 1.8.1 contains a new exposition of known results.

The term *parametric searching* is not a technical term, but rather a collective name to a number of techniques. These techniques can be applied to problems involving both combinatorial and numerical features, such as computational geometry problems, combinatorial optimization problems, etc. The output of these problems is always a real number or a vector of reals.

1.2 A Preliminary Example

The first technique we describe, and by far the most important and general one, is the ingenious technique developed by N. Megiddo [Me]. As in Megiddo's paper, we introduce it through an example.

The problem. Let $Y_i(\delta) = a_i\delta + b_i$ be n distinct linear functions such that all the a_i's are positive (thus all are increasing functions). For every real δ let $F(\delta)$ be the median of the set $\{Y_1(\delta), \ldots, Y_n(\delta)\}$. F is a piecewise linear, monotone increasing function with $O(n^2)$ breakpoints. Our problem is to find $\delta^\star$, the root of the equation $F(\delta) = 0$.

One (trivial) way to solve the problem is to find all the roots of the n functions Y_i; their median is $\delta^\star$. But this method will not serve our expository needs.

A first solution. We compute the median of the set $\{Y_1(\delta^\star), \ldots, Y_n(\delta^\star)\}$. There is an obvious obstacle, that we do not know $\delta^\star$ and therefore we do not know the members of the set. The median finding algorithm that we will employ however, will access the members of the set only through comparisons between pairs of elements. Comparing a pair of linear functions only amounts to finding in what side of their intersection we are. If we are to the left of the intersection, the line with smaller slope is above the other; to the right of the intersection, the line with higher slope is above.

Thus we can resolve a comparison between $Y_i(\delta^\star)$ and $Y_j(\delta^\star)$ in the following manner (see Figure 1.1): without loss of generality assume that $a_i > a_j$. We compute the intersection point $Y_i(\delta_{i,j}) = Y_j(\delta_{i,j})$. We evaluate $F(\delta_{i,j})$, the median of the set $\{Y_1(\delta_{i,j}), \ldots, Y_n(\delta_{i,j})\}$ whose elements can easily be computed. Because F is monotone increasing, if $F(\delta_{i,j}) < 0$ than $\delta_{i,j} < \delta^\star$ and thus $Y_j(\delta^\star) < Y_i(\delta^\star)$. The opposite case $F(\delta_{j,j}) > 0$ is symmetric, and if $F(\delta_{i,j}) = 0$ we can stop the algorithm right there, because we have found $\delta^\star = \delta_{i,j}$.

Once we know the median line at $\delta^\star$ (and this is what the algorithm computes!) we can easily find its root, and this is the sought $\delta^\star$.

The complexity of this algorithm is easy to analyze. We use the median finding algorithm of Blum et al [BFPRT] which performs $O(n)$ comparisons, each of which is resolved by performing another median finding procedure, this time on a concrete set of numbers. Thus every comparison takes $O(n)$ time and the whole algorithm runs in $O(n^2)$ time.

A better solution. We can improve this, by noticing that every single comparison made by the main median finding algorithm is very costly. The trick is to use a parallel median finding algorithm. We use for this purpose

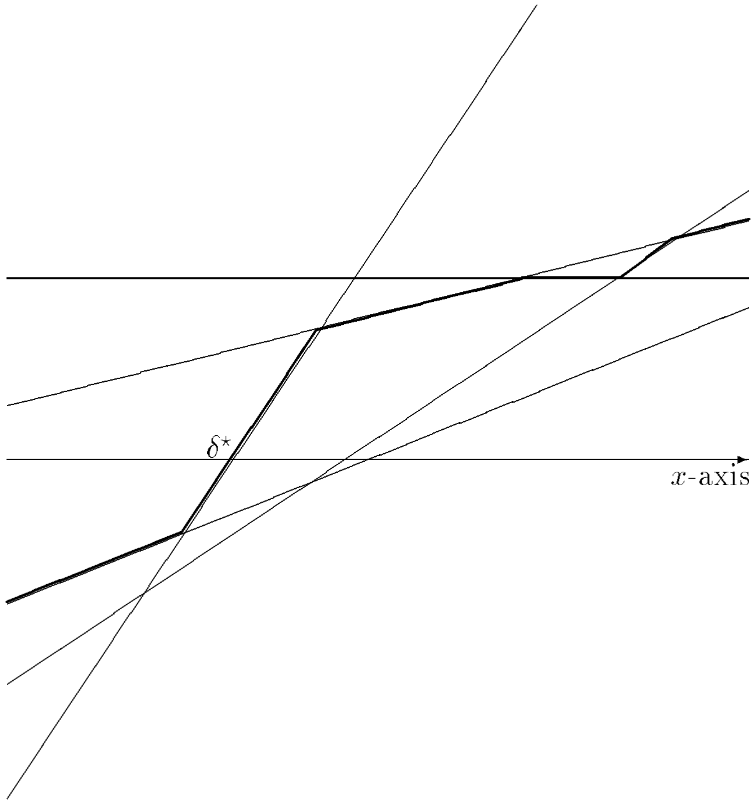

Figure 1.1: The median of a set of lines

a parallel sorting algorithm (that certainly finds the median), which uses n processors and $O(\log n)$ parallel time (see below for a more detailed discussion of the model of parallel computation that we assume). We simulate the first parallel step *sequentially*, by simulating the n processors one by one. Each one performs a comparison. We do not resolve them immediately as in our first version, but rather collect the comparisons from all the processors. When all n processors generate their comparisons, we calculate the n intersection points involved in the comparisons. We sort the points, $\delta_1 < \delta_2 < \cdots < \delta_n$. We now perform a binary search to locate δ^* in this sorted list. This is done using $\log n$ "costly" comparisons, each using $O(n)$ time.

Once we have located $\delta^\star$ in the list, we can resolve all the comparisons in the first step of the parallel algorithm.

We repeat this for every parallel step, and as in the sequential case the termination of the algorithm provides us with a solution to the original problem of finding $\delta^\star$.

The time complexity of this version is much better. We perform $\log n$ comparisons per parallel step, and there are $O(\log n)$ such steps. Thus the cost of all comparisons is $O(n \log^2 n)$. We sort $O(\log n)$ sequences of test values, one in every parallel step, with a total cost of $O(n \log^2 n)$. In addition, the cost of simulating the parallel sort is $O(n \log n)$, and thus the overall running time is $O(n \log^2 n)$, almost an order of magnitude better than the sequential version of the algorithm.

An improvement. Before we conclude this section we would like show a minor improvement. This improvement is not enough to lower the asymptotic running time of the algorithm, but it does lower the running time and together with another improvement due to Cole [Co1], shown in section 1.8.2 will lead to lower asymptotic running time.

The observation is that we need not sort the set of test values $\delta_1 < \delta_2 < \cdots < \delta_n$ in order to resolve all the n comparisons. What we do is find the median δ_m of this set and compare it to $\delta^\star$. If $\delta_m < \delta^\star$ we resolve half the comparisons whose intersection is lower then δ_m. Otherwise the other half of comparisons is resolved. The cost of this step is one comparison and $O(n)$ other operations (finding the median δ_m and resolving half the comparisons).

We repeat this procedure with the values left until all the comparisons are resolved. There are $\log n$ such steps and comparisons, and the total number of other operations is $O(n)$ (because the sizes of the resulting subsequences shrinks geometrically). Thus there will still be $\log n$ comparisons per parallel step, but the number of other operations other than those involved in the comparisons is only $O(n \log n)$, an improvement of a logarithmic factor.

We note that another improvement could be achieved by employing a more efficient parallel median-finding algorithm such as in [CY].

1.3 A Formal Description of Megiddo's Technique

The previous example showed the main ideas behind Megiddo's technique, and it is now time to turn to more formal details, which reveal the full generality of the method.

Megiddo's technique. Suppose we have a problem $\mathcal{P}(\delta)$ that receives as input n data items and a real parameter δ. We want to find a value $\delta^\star$ at which the output of $\mathcal{P}(\delta)$ satisfies certain properties. Typical examples are when the output of $\mathcal{P}(\delta)$ is a real number and we want to find a value $\delta^\star$ in which the output is zero or extremal. Suppose we have an efficient sequential algorithm A_s that solves $\mathcal{P}(\delta)$ at any given δ. We also require that A_s can determine whether a given δ is equal to, less than, or greater than the sought value $\delta^\star$. Assume moreover that the flow of execution of A_s depends on comparisons, each of which involves testing the sign of a low-degree polynomial in the input data items and δ.

Megiddo's technique runs the algorithm A_s "generically", without specifying the value of δ, with the intention of simulating its execution at the unknown $\delta^\star$. Each time a comparison is to be made, we compute the few real roots of the polynomial associated with the comparison. We run the "concrete" version of A_s at each root, and thereby determining the location of $\delta^\star$ among the real roots. This determines of course the sign of the polynomial at $\delta^\star$, because the sign of a polynomial is constant between two consecutive roots. This in turn determines the outcome of the original comparison made by the "generic" algorithm A_s, and its execution can be resumed.

As we trace the execution of the "generic" A_s, each comparison resolved further constrains the range where $\delta^\star$ can lie. We obtain a sequence of smaller and smaller intervals containing $\delta^\star$ until we reach the end of A_s with a final interval I. The outcome of A_s will be the same combinatorially when we run it at any $\delta \in I$, including $\delta^\star$. If I is a singleton, we have found the sought $\delta^\star$. Otherwise it is usually straightforward to find $\delta^\star$, often as an endpoint of I. Note that if $\delta^\star$ is found to coincide with a real root of one of the comparisons, then the algorithm can be stopped "prematurely" right there.

Actually, if at the end of the algorithm one of I's endpoints is $\delta^\star$, then it was one of the real roots tested. If A_s can decide whether a value is $\delta^\star$, the "generic" algorithm will never terminate, but rather it will be halted as soon as $\delta^\star$ is found. This is the case in most of the applications of Megiddo's technique.

The cost of this search is usually dominated by C_sT_s, where C_s is the maximum number of comparisons made by A_s and T_s is the running time of A_s. This bound is generally too high, so Megiddo suggests to replace A_s with a parallel algorithm A_p that also solves the problem $\mathcal{P}(\delta)$. If A_p uses P processors, and runs in T_p parallel steps, then each such step involves at most P *independent* comparisons, that is, each can be carried out without knowing the outcome of the others. We then collect the roots of the P polynomials associated with the comparisons, and run a binary search to find the location of $\delta^\star$ among them, using the serial algorithm A_s at each binary step. This requires $O(P + T_s \log P)$ time per parallel step, for a total of $O(PT_p + T_sT_p \log P)$ time. This is often a saving of nearly an order of magnitude in running time over the generic sequential version.

As in our example, our overall algorithm is sequential. In practice, each step of the parallel algorithm is simulated sequentially. The only reason for using a parallel algorithm is to be able to generate only a small number (T_p) of "batches" of independent comparisons, which can be efficiently resolved. See below for a discussion of the model of computation that we use.

Note that we can maintain the sequence of smaller and smaller intervals known to contain $\delta^\star$ (that is we can maintain the tightest bounds on $\delta^\star$ found so far) so that we can trivially compare $\delta^\star$ with a value not lying in the current interval. Alternatively, we can behave in each parallel step as if we have no previous information on the location of $\delta^\star$. We did not make either assumption in the complexity analysis. It is certain however, that maintaining the interval will lower the number of costly comparisons. In a sense, this prevents the algorithm from asking questions the answer of which is already known. The effect of this heuristic will be examined experimentally in chapter 3.

Plugging the example into the formal framework. To render the general technique just demostrated somewhat more digestible, we show how to "plug in" the example of the previous section into the formal framework. The problem $\mathcal{P}(\delta)$ was computing the median $F(\delta)$. The value $\delta^\star$ was the root of this monotone increasing function F. The algorithm A_s can also determine whether a given δ is equal to, less than or greater than $\delta^\star$, because $F(\delta)$ is monotone increasing. The sequential algorithm A_s was a median finding algorithm run on the set of values $\{Y_1(\delta), \ldots, Y_n(\delta)\}$, and therefore T_s was $O(n)$ ([BFPRT]). The parallel algorithm A_p was a parallel sorting algorithm with $P = n$ processors and $T_p = O(\log n)$ running time. The comparisons in the sorting algorithm were resolved by testing the sign of the linear polynomial $(a_i\delta + b_i) - (a_j\delta + b_j)$.

1.3.1 Relaxing some requirements — convex functions.

In many cases in which $\mathcal{P}(\delta)$ is a monotone function and $\delta^\star$ is its root, the requirement that A_s determine whether a given δ is equal to, less than or greater than $\delta^\star$ is satisfied automatically, as in our example. Another class of problems involves a *convex* function $\mathcal{P}(\delta)$, and the solution $\delta^\star$ is the point where it achieves its minimum. Given a test value δ, the standard technique to determine the side of δ containing $\delta^\star$ is to compute the sign of the derivative of $\mathcal{P}(\delta)$). We show below how this computation can be avoided, and replaced by a simpler technique.

Consider a step of the parallel "generic" algorithm A_p. We compute $O(n)$ roots and we have to locate $\delta^\star$ among them. As before, we find their median δ_m and run A_s at this value. We also find the two root adjacent to δ_m, denote them $\delta_{m'} < \delta_m < \delta_{m''}$ and run A_s at these values too. If $\mathcal{P}(\delta_{m'}) < \mathcal{P}(\delta_m) < \mathcal{P}(\delta_{m''})$ we conclude that the convex function is decreasing at δ_m, therefore $\delta^\star > \delta_m$, and we resolve half the comparisons and continue as before. The case $\mathcal{P}(\delta_{m'}) > \mathcal{P}(\delta_m) > \mathcal{P}(\delta_{m''})$ is symmetric and dealt with in a similar manner. The difficult case is $\mathcal{P}(\delta_{m'}) > \mathcal{P}(\delta_m)$ but also $\mathcal{P}(\delta_{m''}) > \mathcal{P}(\delta_m)$. In this case we do not know in what side of the minimum δ_m lies. We can however resolve all the comparisons other than the one that generated δ_m, because we have located $\delta^\star$ between $\mathcal{P}(\delta_{m'})$ and $\mathcal{P}(\delta_{m''})$.

We now duplicate the state of the parallel algorithm A_p to form two copies of it, $A_{p,<}$ and $A_{p,>}$. In $A_{p,<}$ we resolve the comparison as if $\delta^\star > \delta_m$ and in the other $\delta^\star < \delta_m$. We also record the value of δ_m. We now run two simulated parallel algorithms. In subsequent stages, as long as we do not run into a value δ' such that $\mathcal{P}(\delta') < \mathcal{P}(\delta_m)$ the initial assumptions remain valid and both $A_{p,<}$ and $A_{p,>}$ can continue their execution. If both reach termination in this state, we know that both assumptions were false and in fact $\delta^\star = \delta_m$, because the algorithms did not find any point lower than δ_m, and thus this is the minimum. If we run into a point δ' such that $\mathcal{P}(\delta') < \mathcal{P}(\delta_m)$, without loss of generality assume that $\delta' < \delta_m$ and therefore $\delta^\star > \delta_m$. In this case the assumption that was used to initiate $A_{p,>}$ was false and we "kill" $A_{p,>}$. But this state of affairs will lead to a new "lowest" point δ' whose relation to $\delta^\star$ is unknown. Thus we will duplicate the state of $A_{p,<}$ and resume the execution with two new algorithms as we did for δ_m.

It is clear that using this procedure does not impair the asymptotic complexity of the general method, because the nondeterminism is limited to at most two paths of execution at any time. There is an added complexity, a constant factor, that results from the fact that we perform three evaluations of $\mathcal{P}$ at any binary step instead of one.

One has to check however that the total cost of duplicating the state of the parallel algorithm A_p does not dominate the complexity. This will be the case in most problems, because we duplicate the state of the algorithm only once per parallel step, or T_p times.

Finally we note that in many applications the direction of the minimum can be found directly by computing the derivative $\mathcal{P}'(\delta)$ (see for example [Me3] for two different ways to compute the sign of these derivatives). The above procedure can facilitate however the application of Megiddo's technique to new problems, where deciding the direction of the minimum through the derivative may be difficult or costly.

1.4 Models of Computation

There are two issues concerning models of computation that need to be clarified. One is the model in which we require the parallel algorithm to be specified, and the second is the arithmetic model of computation in which the entire algorithms are specified.

1.4.1 Model of parallel computation

When we use a parallel algorithm in Megiddo's technique, its complexity has a two-fold influence on the overall parametric search algorithm. As we simulate it, we perform every step the parallel algorithm performs. Thus the overall number of operations of the algorithm, including pointer operations, number manipulations and so on, are counted in the overall complexity of the parametric searching algorithm. These operations have to be counted using some plausible model of computation such as a RAM or a Turing Machine. The initial motivation for using a parallel algorithm is to batch together (the costly) comparisons. So a separate count is needed for the number of processors used by the algorithm (in other words, how many comparisons can be batched together in a single step) and in how many steps will the algorithm terminate. This is in fact all that is required from the parallel algorithm. There is no need to concern ourselves about synchronization, inter-process communication, and other problems of concurrent programming, because the algorithm will be simulated on a sequential, rather than a parallel machine. The model of computation the parallel algorithm can be specified in is thus Valiant's comparisons model [Va] (in which only comparisons are counted in the complexity), with the added constraint that operations other than comparisons need to be counted as well, but need not be performed in parallel.

Since this is a rather weak model of parallel computation, it may be easier to design algorithms that run under it rather than under a more realistic model such as EREW-PRAM or a similar model.

1.4.2 The arithmetic model of computation

As in other geometric-related algorithms there are two issues that arise. One is the class of operations that can be performed in constant time. In addition to the standatd arithmetic operations, we also assume that real roots of polynomials of constant degree can be computed in constant time. This can actually be done, but it is rather complicated for high degree polynomials — see below.

Another, more subtle issue is related to the representation of real numbers. We assume that real numbers are represented *exactly* and that arithmetic operations on them can be done in constant time. This assumption lies at the very heart of parametric search techniques. Assume for example that we are working with a fixed point number representation with b bits. Instead of simulating a parallel algorithm we could simply perform a binary search over the interval of representable numbers and in b (which is constant) number of comparisons we could restrict the interval where the sought parameter lies to an interval containing no representable numbers. This procedure (which seems to be very practical by itself) shows that parametric search algorithms search for an *exact* value and are therefore not optimal as approximation algorithms.

1.5 Another Important Example

In this section we present another important example. This problem, called *slope selection* has several characteristics that highlight some issues in parametric searching.

Given a set of n lines in the plane and an integer $1 \leq k \leq \binom{n}{2}$, the problem is to select the k-th leftmost intersection point of the lines. The name slope selection comes from the geometric dual of this problem, where, given n points, we have to find the k-th largest slope of the $\binom{n}{2}$ lines determined by the points. We assume that no two lines are parallel and that the x-coordinates of the intersections are all distinct.

The problem was ingeniously solved in optimal $O(n \log n)$ time by Cole, Salowe, Steiger and Szemerédi [CSSS], and all the material in this section is from their paper.

By reduction from element uniqueness it follows that the lower bound for this problem is $\Omega(n \log n)$. If the search is *shallow*, that is $k = O(n)$, the problem can be solved in optimal $O(n \log n)$ time by a standard plane sweep technique. This observation will have a crucial role in the next section.

A parametric search algorithm. We now present a parametric search algorithm that solves the problem. We label the x-coordinates of the intersection points from left to right $t_1, \ldots, t_k, \ldots, t_{\binom{n}{2}}$. We also number the lines in decreasing order of their slopes, so that the lines are ordered upwards to the left of the first intersection point t_1 (see the numbering in Figure 1.2). We use a parallel sorting algorithm as the generic algorithm A_p, which attempts to sort the lines according to their y-coordinate immediately to the right of t_k, that is, at $t_k + \epsilon$, for sufficiently small $\epsilon > 0$. If we could perform this computation then the final interval I in which t_k is known to lie must be (t_k, t_{k+1}), because the permutation the sorting algorithm returns is valid in any point of I, but the permutation is certainly not valid outside (t_k, t_{k+1}). On the other hand, the interval I cannot be smaller than (t_k, t_{k+1}) because this interval does not contain any intersection points, so it can not be reduced further.

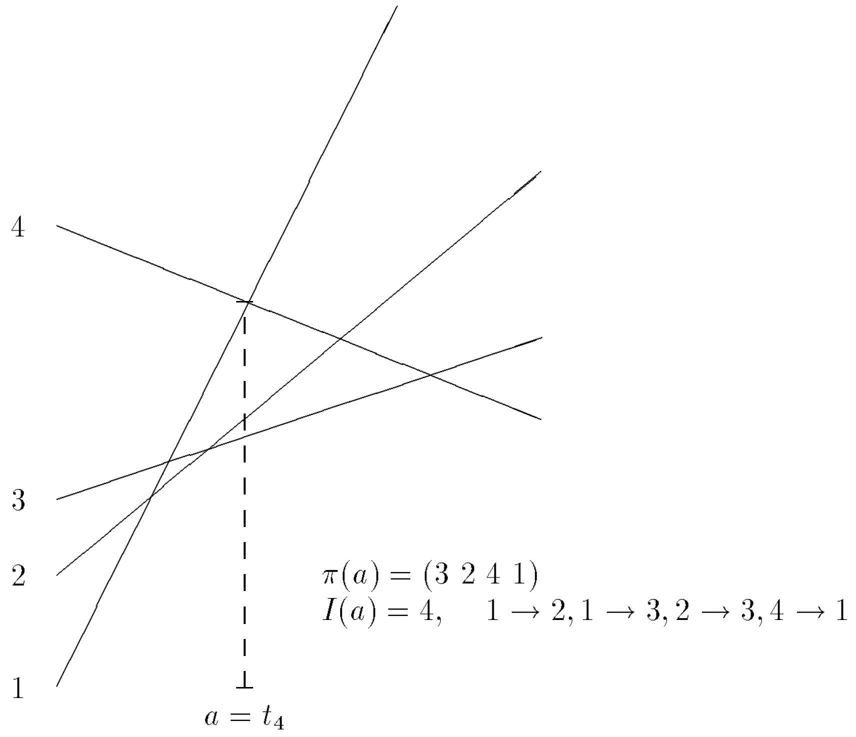

Figure 1.2: A permutation of lines and the number of inversions

It remains to show how to decide, given an x value (which will always be an intersection point t_i of two lines being compared), whether $x < t_k$, $x = t_k$ or $x > t_k$, or actually to decide whether $i < k$, $i = k$ or $i > k$. The solution is to compute the y coordinates of the lines at $x + \epsilon$, sort this list in decreasing order, and calculate the number of inversions $I(x)$ in this permutation which we label $\pi(x)$. This will give us the index i, because every intersection point to the left of (and including) t_i (and only these points) adds one to the number of inversions (see Figure 1.2). Computing the permutation amounts to sorting the lines according to their y values, which can be done in time $O(n \log n)$. Calculating the number of inversions in a permutation can also be done in time $O(n \log n)$ by performing a merge sort, and in each merging step computing the number of inversions between the two halves of the list and adding the number of permutations inside each half. These numbers of course are computed in a recursive manner together with the sorting.

We conclude that if we use a parallel sorting algorithm with n processors and $O(\log n)$ parallel parallel time (for example [AKS]), the overall running time of the algorithm will be $O(n \log^3 n)$. Because the parallel procedure is a sorting network, we can make use of Cole's improvement [Co1] discussed below in section 1.8.2, which performs only $O(\log n)$ sequential tests instead of $O(\log^2 n)$, thereby reducing the running time to $O(n \log^2 n)$.

Further relaxation of the requirements. Recall that in the formal description of Megiddo's method it was required that the parallel algorithm A_p also solves the problem $\mathcal{P}(\delta)$. We are now in position to comment that this is far too stringent a requirement. As in this section, all we have to require is that the output of A_p changes combinatorially at $\delta^\star$, and this will ensure that $\delta^\star$ will be tested and found.

This observation is useful in many algorithms, including polygon containment problems to be discussed later, and elsewhere, [AS] for example. This observation was noted in Megiddo's original paper [Me] where he used sorting as the parallel procedure in a problem involving minimum spanning trees.

An optimal solution. Cole et al. [CSSS] were able to shave another logarithmic factor off the running time of the algorithm and bring it down to an optimal $O(n \log n)$. There are many technical difficulties in their solution, and therefore we do not describe it in full; the interested reader should consult their paper. We restrict out attention to the main ideas behind their improvement, since we believe similar techniques can improve other algorithms as well. The reader should bear in mind however, that these ideas cannot be implemented without solving some technical difficulties.

The first observation is that computing the exact number of inversions in the permutation $\pi(t_i)$ is expensive and not necessary. All we want to know is whether $i < k$, $i = k$ or $i > k$. Knowing i exactly is not required. Cole et al. suggest that instead of calculating the exact number of inversions one can calculate an *approximation* to this number. An error bound on this approximation is also needed. For example, if $k = 250$ and the approximation to the *rank* i of t_i is $i^\star = 200$ with an error bound of 25, we know that $i < k$ because $175 \le i \le 225 < 250 = k$. If the error bound is too large, we will refine the approximation and obtain a new approximation with a smaller error bound, and repeat this step until the relation between i and k can be decided.

The second observation is that the sequential tests the algorithm makes are not independent, and the resolving of a previous test can ease the resolving of the current one. Consider a sequential test made by the algorithm. We are given a point t_i and we would like to compute the above approximation. The point t_i lies in the interval $I = (x', x'')$ where t_k is known to lie (otherwise the comparison can be trivially resolved). We have computed an approximation to the rank of the interval's endpoint. If t_i is not too far away from one endpoint of the interval, say x', the permutations $\pi(x')$ and $\pi(t_i)$ are not much different, and therefore $\pi(t_i)$ and the number of inversions may be computed cheaply from $\pi(x')$ and its rank.

The third observation is that if we use the permutations themselves as suggested above, then the calculation of $\pi(t_i)$ from $\pi(x')$ is still too costly. Here the idea of approximation comes to the rescue once again, because if we keep only an approximation $\pi^\star$ to the permutation π, then "dragging" it from x' to t_i will not be too costly. Specifically, the cost of this "dragging" while keeping the same error bound can be made $O(n)$, and of course the total number of refinements during the whole algorithm is $\log n$ if every refinement halves the error bound. Thus the total cost of the sequential tests will be $O(n \log n)$ time, bringing down the total cost of the algorithm to the same complexity.

We will not go into further details here, but we would like to summarize the method. The main idea is not to compute the permutation π, but to maintain a data structure that represents an approximation to that permutation. The approximated rank is the number of inversions in this approximated permutation. The approximation is fine enough so that the relation of the rank to k can be decided, but at the same time coarse enough so that "dragging" the approximated permutation and maintaining the number of inversions is not too costly. As "dragging" is cheap, and the number of refinements is logarithmic, we can save a logarithmic factor in running time.

1.6 A Probabilistic Approach

In this section we show how randomization can be exploited in solving parametric search problems. The first part of this section describes a direct way of using random sampling, eliminating the need for using Megiddo's complex technique. The second part is a remark on the use of probabilistic algorithms within the framework of Megiddo's technique.

1.6.1 Direct use of random sampling

We now describe how to use random sampling to solve parametric search problems. The idea is simple; we have a large ordered (discrete) parameter space where the solution in known to exist. We do not want to construct it explicitly. Megiddo's technique uses a parallel algorithm to direct this search. But a simpler approach can be taken.

Let us simply select at random some elements from this parameter space, sort them in increasing order $\delta_1 < \delta_2 < \cdots < \delta_n$, and locate the desired solution among them, say $\delta_k < \delta^\star < \delta_{k+1}$. Intuitively, the number of elements which lie in the interval (δ_k, δ_{k+1}) should be small, and we now conduct the rest of the search in a much smaller space.

Let us see how this idea can be implemented for the slope selection problem. We have $\binom{n}{2} = O(n^2)$ intersection points between our n lines. The x-coordinate of each one of them can be the solution. We therefore select a random sample of size n from the set of intersection points, assigning equal probability to each point. As these points are referenced by a two-dimensional index (point $p_{i,j}$ is the intersection of lines i and j) we simply select n pairs of lines and compute their intersections (we are only interested in the x-coordinate). Next we sort the n numbers $\delta_1 < \delta_2 < \cdots < \delta_n$ and locate $\delta^\star$ among them, say $\delta_k < \delta^\star < \delta_{k+1}$, using $\log n$ costly comparisons. If we use the method of Cole et al [CSSS] to locate $\delta^\star$, the cost of this step will be only $O(n \log n)$. We now need an argument that shows that the expected number of intersections in the interval (δ_k, δ_{k+1}) is $O(n)$, so that we can perform a shallow search to locate $\delta^\star$, using a sweep procedure, for an overall expected running time $O(n \log n)$.

We need the following lemma:

Lemma 1 *Given a set of N points, $x_1, \ldots, x_N$, one of which, say x_j, is* *marked, and a random sample of K points $x_{i_1} \leq \cdots \leq x_{i_K}$ such that $x_{i_l} \leq x_j \leq x_{i_{l+1}}$, where the points in the random sample are assumed to be chosen independently and with equal probability, then the expectation of $i_{l+1} - i_l$ is $O(N/K)$.*

Proof. As the values of the points are not important, we can ignore them and work with the indices only. So we are given a random sample of K points in the range $1, \ldots, N$. Let us decide on the following sampling strategy: we select K i.i.d. random variables $X_1, \ldots, X_K$ in the real interval $(0, N]$ and take their ceilings. Sorting the random variables (which is equivalent to sorting the ceilings) we obtain their K order statistics $X_{(1)} \leq \cdots \leq X_{(K)}$. It is known that $E(X_{(i)}) = \frac{iN}{K+1}$ (see for examle [Fe, p. 23] for the density function; the expectation is easily derived). Therefore the differences $E(X_{(i)} - X_{(i-1)}) = \frac{N}{K+1}$ (we take $X_{(0)} = 0$ and $X_{K+1} = N$). But j lies in one of these intervals and the proof is complete.

Note that while the lemma we needed is of a general nature, the shallow search procedure is specific to the problem. Repeating the sampling procedure will not lead us quickly to the solution.

To summarize, we were able to solve the slope selection problem in optimal expected time, without resorting to parallel algorithms or to Cole's improvement. At the present time, this solution seems to be much more practical in terms of the constants hidden in the O-notation.

Jiří Matoušek [Ma] has obtained similar results for the same problem. His algorithm is quite similar to ours, but he proves its correctness in a different way. The only difference between the algorithms is that Matoušek uses a "quality control" to make sure the sample is good. If the sample is not good, that is the remaining interval contains too many intersection points, the sampling is repeated until a good sample is obtained. This is clearly better than our algorithm that will sweep an interval containing many intersections when the sample is not good.

Perhaps even more important is Matoušek's observation that even a small sample containing only one point, leads to a good algorithm. In this case the (slightly worse) expected running time will be $O(n \log^2 n)$. The importance of this technique (referred to as *randomized halving*) lies in the fact that it may be widely applicable to parametric searching problems.

1.6.2 Megiddo's probabilistic improvement

Cole [Co1] reports that Megiddo has suggested another use of randomness in parametric searching. Cole refers to this as Megiddo's probabilistic improvement, so we shall also use this name. Megiddo suggests that if one uses probabilistic parallel algorithms as the "generic" algorithm in this technique, the expected running time will be lower than the running time of the deterministic version. Cole [Co1] uses probabilistic parallel median and minimum finding algorithms that use $O(n)$ processors and constant expected time to improve some parametric search algorithms.

Agarwal, Aronov, Sharir and Suri [AASS] use this idea extensively. They devise randomized algorithms for both the parallel "generic" algorithm and the sequential algorithm to reduce considerably the complexity of the deterministic version of the algorithm.

1.7 A Matrix Search Approach

In a number of papers, Frederickson and Johnson [FJ1, FJ2, Fr] propose a completely different approach to parametric searching. The material in this section is taken from their papers. Their approach is both efficient and simple when applicable. It also demonstrates the nature of parametric searching. As usual, we do not go into all the details and generalizations possible, but rather outline the technique to help the reader appreciate it. Again we recommend that the interested reader consult the original papers for the missing details.

Consider the special case of the *max-min tree k-partitioning problem* where the tree is a single path. In the max-min tree k-partitioning problem we are given a tree with n nodes and a non-negative weight associated with each node, and we are required to delete k of the edges so as to maximize the weight of the lightest of all resulting subtrees. Let us define $F(\delta)$ to be the maximal number of edges that can be deleted so that the weight of every subtree is at least δ. We would like to find δ^*, the maximal δ such that $F(\delta) \geq k$.

As pointed out by Perl and Schach [PS], $F(\delta)$ can be evaluated in $O(n)$ time. Megiddo exploited this, and designed a parallel procedure for evaluating $F(\delta)$ that runs in $O(\log n)$ time with $O(n)$ processors, and obtained an $O(n \log^2 n)$ algorithm for the case where the tree is a path. Cole [Co1] later improved this and obtained an $O(n \log n)$ algorithm.

We now present the matrix search approach. Let the nodes be $v_1, v_2, \ldots, v_n$ and let the weights be $w_1, w_2, \ldots, w_n$. We define the distances $d_{i,j} = \sum_{\nu=j}^{i} w_\nu$ for $1 \le j \le i \le n$, and for technical reasons $d_{i,j} = 0$ for $1 \le i < j \le n$. It is clear that the solution $\delta^\star$ equals to $d_{i,j}$ for some i and j. Thus have identified a set of size $\frac{1}{2}\binom{n}{2} = O(n^2)$ that contains the solution. We could compute the set explicitly and perform a binary search over it to find $\delta^\star$. This would cost $O(n^2)$ time just to compute the set, so we have to abandon this idea, although the binary search would not examine more then $O(\log n)$ elements of the set.

The important observation is that elements in the set can be referenced implicitly as $d_{i,j}$ and that there is a partial order on the set. If the order was a total order, we could perform the binary search easily, without precomputing the whole set. The idea is to exploit the partial order that is present in the set to direct the binary search. The partial order follow from the fact that $d_{i,j} \ge d_{i,j+1}$ and $d_{i,j} \le d_{i+1,j}$. Frederickson refers to this partial order as a *sorted matrix* because the elements can be arranged in a square matrix

$$A_{n\times n} = \begin{pmatrix} d_{1,1} & 0 & 0 & \ldots & 0 \\ d_{2,1} & d_{2,2} & 0 & & 0 \\ d_{3,1} & d_{3,2} & d_{3,3} & & 0 \\ \vdots & & & & \vdots \\ d_{n,1} & d_{n,2} & d_{n,3} & \ldots & d_{n,n} \end{pmatrix}$$

where the rows and the columns are sorted. The matrix need not be constructed explicitly of course, since we can compute an element in constant time if we keep only the linear-space array $w_1, w_2, \ldots, w_n$ (more precisely, if we maintain the prefix sums in this array).

Frederickson and Johnson show how to search for a value $\delta^\star$ in such a matrix, assuming a procedure that can compare $\delta^\star$ with any givan value (as usual, $\delta^\star$ is not known explicitly). Their technique requires $O(\log n)$ such comparisons and $O(n)$ other constant-time operations. As the comparisons cost $O(n)$ each, the total cost of their max-min path k-partitioning algorithm is $O(n \log n)$. Frederickson [Fr] shows how this can be extended to trees, and designs better comparison algorithms so that he finally obtains a linear time optimal algorithm for the max-min tree k-partitioning problem.

We now describe the search procedure on this matrix. We assume that $n = 2^k$ for some integer k. The procedure has two phases. In the first phase, we have a collection of sorted matrices, which are all sub-matrices of the initial matrix A. In every step of the first phase we divide each of the matrices into four square submatrices and discard some of these matrices. After $k = \log n$ steps we are left with only singleton submatrices, and the second phase consists of a regular binary search, as the one described in section 1.2.

The first phase starts with A as the only matrix and divides it into four square submatrices. In every step, after dividing the matrices we form two sets of elements: S is the set of the smallest element of each matrix (the upper right element) and L is the set of largest elements (lower left elements). We compute x_s, the median element in S, and x_l, the median element in L. We determine the relation of x_l and x_s to the sought value $x^\star$ by using two binary tests. If one of them is $x^\star$ we are done. Otherwise there are four possible outcomes to these comparisons:

i. If $x_s < x^\star$ we update the lower bound on $x^\star$ and discard every matrix whose largest element $\leq x_s$.

ii. If $x_s \geq x^\star$ we update the upper bound on $x^\star$ and discard almost every matrix whose smallest element $\geq x_s$. We leave one matrix whose smallest element is x_s so that this element is still in the searched range. This results in discarding at least half the matrices minus one.

iii. If $x_l > x^\star$ we update the upper bound on $x^\star$ and discard every matrix whose smallest element $\geq x_l$.

iv. If $x_l \leq x^\star$ we update the lower bound on $x^\star$ and discard almost every matrix whose largest element $\leq x_l$. We leave one matrix whose largest element is x_l so that this element is still in the searched range. This results in discarding at least half the matrices minus one.

We shall now show that the number of matrices in each step at most roughly doubles. As we start with one matrix and there are $\log n$ steps, the total number of matrices over all steps is linear (it is bounded by a geometric series whose largest element is n). Thus the number of "non-comparison" operations (median finding and discarding matrices) is $O(n)$ as stated.

We use induction to prove that the number of matrices after step i is smaller than or equal to $B_i = 2^{i+2} - 1$. This is obvious for step 0, because in the beginning of step 1 we have one matrix, and $B_0 = 2^2 - 1 = 3$. If after step i *either* condition (ii) or (iv) is satisfied then at least half the (but one) matrices are discarded and we are left with at most

$$\frac{1}{2}(4B_{i-1}) + 1 = 2B_{i-1} - 2 + 1 = B_i.$$

Otherwise *both* conditions (i) and (iii) occur, that is $x_s \leq x^\star \leq x_l$. Now consider the matrices we have at hand as sub-matrices of the original, that is we consider the original matrix as a block matrix of the appropriate size. Because the original matrix is sorted, if two such blocks are one above and to the right of another along a secondary diagonal of the block matrix, then every element of the first block is smaller than or equal to every element of the second block. Thus there can be at most one block along each secondary diagonal such that its largest element is larger than x_l and its smaller element is smaller than x_s. The number of diagonals is $d = 2^{i+1} - 1$. Let h be the number of matrices we have at hand. At least $\frac{h}{2}$ of them have an element smaller than x_s (by the definition of x_s). At least $\frac{h}{2} - d$ of those have their largest element smaller than x_s, and can therefore be discarded. Similarly at least $\frac{h}{2} - d$ of the present matrices have their smallest element greater than x_l, and can also be discarded. Thus the number of remaining matrices is at most

$$2d = 2(2^{i+1} - 1) < 2^{i+2} - 1 = B_i$$

which concludes the proof.

We summarize the technique. The problem is such that a set of candidate parameters for the solution can be specified implicitly. The set of candidates has a special structure such that a partial order can be imposed on them. A binary search is carried over this set using the notion of sorted matrices, using $O(\log n)$ comparisons and $O(n)$ other operations. Note that weaker partial orders, such as a "sorted cube" with a similar structure, enable the use of a similar technique, but unfortunately (for the case of a cube) with an $O(n^2)$ overhead of other operations. This is because in the case of a cube there are $O(n^2)$ "diagonals", so we can arbitrarily embed n^2 elements, upon which no order information can be imposed in a sorted cube of n^3 elements.

1.8 Some More Extensions

1.8.1 The Connection Between Linear Programming and Parametric Searching

In this section we show the connection between parametric searching and linear programming. We have two goals in mind. One is to show the potential practical value of Megiddo's technique. The other goal of this section is to show the connection between Megiddo's parametric search technique and Megiddo's linear programming algorithm [Me1, Me2]. The connection between the two algorithms suggest a general way to improve the efficiency of parametric search algorithms.

Parametric search solution to linear programming. Consider linear programming with only 2 variables. For the sake of simplicity assume that we have a set of n half planes $y \geq a_i x + b_i$, $i = 1, \ldots, n$ (the simplification is that all the inequalities have the same sign). We would like to find the lowest point that satisfies all the n constraints.

If we knew $x^\star$, the x coordinate of this point $(x^\star, y^\star)$, we could compute also $y^\star$ which is simply $\max\{a_1 x^\star + b_1, \ldots, a_n x^\star + b_n\}$. So as usual in Megiddo's technique we use a parallel maximum finding algorithm to compute $\max\{a_1 x^\star + b_1, \ldots, a_n x^\star + b_n\}$ without actually knowing $x^\star$. We still have to show how to determine for a given $x_{i,j}$, the abscissa of the intersection of two lines, its relation to $x^\star$, in order to resolve comparisons made by the parallel algorithm. To answer this question we compute $Y = \max\{a_{i,j} x_{i,j} + b_{i,j}, \ldots, a_{i,j} x_{i,j} + b_{i,j}\}$. We then compute the set $\mathcal{I}$ of lines achieving this maximum, i.e. $Y = a_k x_{i,j} + b_k$ for $k \in \mathcal{I}$. We now look at the slopes a_k for $k \in \mathcal{I}$. If there are both non-positive and non-negative slopes, then $x^\star = x_{i,j}$. If there are only negative slopes then $x^\star > x_{i,j}$ and only negative slopes indicate that $x^\star < x_{i,j}$ (see Figure 1.3).

As for the maximum finding algorithm we can use Valiant's algorithm [Va], which uses n processors and takes $O(\log\log n)$ time.

As each comparison costs $O(n)$ time, both for maximum finding and for computing the set $\mathcal{I}$, the overall running time of the algorithm is $O(n \log n \log\log n)$.

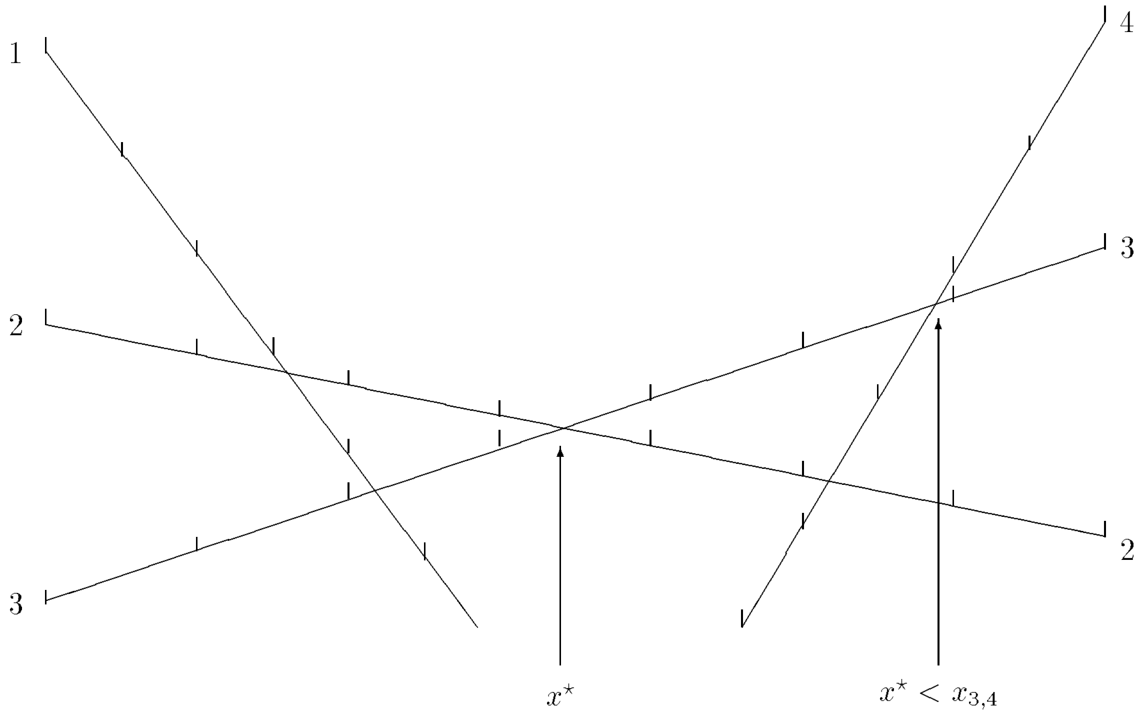

Figure 1.3: A linear programming problem; half-plane 4 is redundant

Megiddo's linear programming algorithm. We can improve upon the algorithm of the last paragraph, using the following observation: when we resolve a comparison and conclude that $a_i x^\star + b_i < a_j x^\star + b_j$, we know that discarding the half plane i from the set of n half planes does not change the solution to the problem (see Figure 1.3).

We examine carefully the state of the algorithm after the first sequential comparison was made. We shall assume that Valiant's parallel *sorting* algorithm [Va] is used for maximum finding. The algorithm begins by performing $n/2$ comparisons, between disjoint pairs of lines. That is, the algorithm arbitrarily groups the lines into pairs and compares the lines in each pair (this is equivalent to the bottom step in a recursive merge sort). The parametric search technique now finds the median of the test values (intersection points) and sequentially compares the median to the unknown $x^\star$. This results in resolving half the comparisons.

Instead of letting the parametric search algorithm continue, we discard a quarter of the half-planes and restart the algorithm with the remaining half-planes. The discarded half-planes are one out of each pair whose comparison is resolved. The total cost of the algorithm up to now is $O(n)$, for grouping the lines into pairs and computing $n/2$ intersection points, for finding their median, for performing the sequential test and for discarding a quarter of the half-planes.

We restart the algorithm with only $3n/4$ lines, and since these problem sizes form a geometric series, the total cost until reaching a trivial case involving only two or three half-planes is an optimal $O(n)$.

As we restart the parallel algorithm before it reaches its second step, we do not actually need it at all. Instead we arbitrarily group the lines into pairs.

Discussion. The preceding paragraphs shows both the ability of parametric search techniques to solve important problems with close to optimal complexity, and that close inspection of what the algorithm does can lead to improvements in its complexity. In this case the improved complexity is optimal.

This method of eliminating a constant fraction of the input at each step, referred to as *prune and search* by Edelsbrunner [Ed], applies to many other problems, such as the planar 1-center problem, which is also solved in linear time by Megiddo [Me].

Finally we note that Megiddo's linear programming algorithm, and hence our first parametric search algorithm can handle all types of inequalities. The algorithm can also be generalized to any fixed dimension and still run in linear time (but the complexity grows more than exponentially with the dimension).

1.8.2 An Improvement Due to Cole

Cole [Co1] improves upon Megiddo's original technique in certain cases, in which the parallel algorithm A_p is either sorting (or can be solved by sorting, like minimum finding) or performing several independent binary searches.

Using the notations of section 1.3, recall that the running time of parametric search algorithms is usually dominated by $O(T_s T_p \log P)$. The $O(\log P)$ factor is caused by the fact that in every simulated parallel step $\log P$ comparisons are resolved, each of which cost $O(T_s)$. Cole shows how to shave this logarithmic factor, by performing only one comparison at each parallel step. This increases the number of steps, but the additional number of steps is $O(\log P)$ which is usually not more than T_p.

We assume for simplicity that one root is associated with every comparison (as is the case in the preliminary example and in the slope selection). Assume that we have a sorting network with width $n/2$ and depth $O(\log n)$, such as the Ajtai-Komlós-Szemerédi [AKS] sorting network. We regard the sorting network as a parallel sorting algorithm which uses $n/2$ processors and $O(\log n)$ parallel steps. Instead of simulating each parallel step (that is each level in the network) in turn, and resolving all the comparisons in that step, Cole proposes to resolve only some comparisons by performing only one costly sequential binary test (running T_s).

Specifically, Cole divides the comparators in the network into two classes: *active* ones whose two inputs are known but their order is not known yet (so the comparison can be made) and *inactive* ones that are not active (either the comparison cannot be made because the inputs are not known or the comparison resolved). Initially, the $n/2$ comparators in level 0 are active and all the other are inactive.

Megiddo's original technique would resolve all the active comparisons at once and thus make the comparisons in the next level the active ones. Cole uses a more refined method. He assigns a *weight* to every active comparison. An active comparison in level j has weight 4^{-j}. At each step, instead of finding the median of the roots of the active comparisons, we find the *weighted median* of the roots in $O(n)$ time [Re]. On this test value we run A_s, the serial binary test. Thus the accumulated weight of the comparisons weighted is at least half the weight of all active comparisons.

It can be shown that under the above weighting scheme, any method of resolving comparisons with half the active weight results in no more than $O(\log n)$ steps until the sort is complete (the bound is $O(f(n) + \log n)$ for a sorting network with depth $f(n)$).

Since we run A_s once in each step, the total running time will be $O(T_s \log n)$ instead of $O(T_s \log^2 n)$ for Megiddo's original method (using a parallel sorting algorithm for A_p).

This usually improves the running time of parametric search algorithms only if one uses a sorting network with width $O(n)$ and depth $O(\log n)$. The only such sorting network known is the AKS network, which at present time involves huge constants. This means that the improvement is not practical. The situation will be changed of course if faster sorting networks are found. Cole's improvement is useful however from a theoretical viewpoint, as we saw in section 1.5 where it was used to construct an optimal algorithm to the slope selection problem.

1.8.3 Multidimensional Parameters

Up until now we have only presented problems and algorithms that compute a single real number. It is interesting to note that Megiddo's parametric search technique can compute multidimensional parameters, for example, a point in the plane (linear programming in higher dimensions is another example). There are two approaches that can be taken, which are applicable to different problems.

The simpler method is to use Megiddo's technique or some other parametric search technique to compute a single real number, and then, given this number to compute the desired solution, such as a point in the plane. This is the approach taken up later in this work to solve polygon containment problems. For example, we would like to find a placement of the largest similar copy of a given triangle in a given convex polygon. The solution thus is a four-dimensional vector, the expansion ratio (size) of the triangle, the placement in the plane of a reference point on the triangle, and the angle in which the triangle is placed. We tackle the problem in stages. We first compute the extremal value of the expansion ratio, using parametric search, that is we find what is the size of the largest triangle similar to the given one that can be placed in the given polygon. Once this value is computed it is straightforward to compute a placement of the largest triangle (which has a fixed size) in the polygon.

A more complicated approach is taken up by Megiddo [Me3]. His problem is formulated as minimization of a real convex function $f(x,y) : R^2 \rightarrow R$. Given a real y_0 the function $h_{y_0}(x) = f(x, y_0)$ is also convex. The function $g(y) = \min_x f(x,y) = \min_x h_y(x)$ is also a convex one. Megiddo uses his parametric search technique to evaluate $g(y_0)$ at a specific y_0, that is, to minimize $h_{y_0}(x)$. He then proceeds to design a similar parallel algorithm that minimizes

$h_{y_0}(x)$. Now, having both a parallel and a sequential algorithms to evaluate $g(y)$, one can apply the parametric search technique again to minimize $g(y)$, using e.g. the technique of Section 1.3.1. The solution is of course the minimum of $f(x,y)$, because $\min_{x,y} f(x,y) = \min_y \min_x f(x,y) = \min_y g(y)$. This can be repeated again and again to work in any fixed dimension.

There is a fundamental difference between the two approaches. In the first case, the optimization problem was one-dimensional, and once solved the other dimensions of the problems can be computed. In the second case, the optimization problem itself was two-dimensional, and this leads to the increased complexity.

Chapter 2

Extremal Polygon Containment Problems

2.1 The Problem

Let P be a convex polygon having k vertices and edges, and let Q be a closed two dimensional space bounded by a collection of polygonal obstacles (the "environment") having altogether n corners. The main problem solved in this chapter is to compute the largest possible placement of a similar copy of P that can be placed inside Q, that is, a placement in which the copy of P does not intersect any of the obstacles. We also give efficient algorithms that solve similar extremal polygon containment problems under more restrictive conditions, and an algorithm that computes largest disjoint placements of two polygons in a third.

Some papers study the *fixed-size* polygon containment problem, in which (the convex) P is only allowed to translate and rotate and we wish to determine whether there is any placement of a copy of P inside Q [Ch, AB1].

Chazelle [Ch] studies the problem for the case where P and Q are arbitrary simple polygons and presents a naive algorithm that takes time $O(k^3n^3(k+n)\log(k+n))$. A more restricted case of the problem, in which both P and Q are convex is also studied by Chazelle [Ch], who solves this case in time $O(kn^2)$. Chazelle gives a simple solution to an even more restricted version in which P is a triangle; this version runs in time $O(n^2)$. Avnaim and Boissonnat [AB1] present an algorithm for the case where both P and Q are non-convex, possibly non-connected polygons, which runs in time $O(k^3n^3\log(kn))$. In another paper Avnaim and Boissonnat [AB] investigate the problem of simultaneous placement of two or three not necessarily convex polygons in a closed polygonal environment. For this problem they allow translations only.

Extremal polygon containment problems were also previously studied. Fortune [Fo], and Leven and Sharir [LS1] consider the following problem: find the largest homothetic copy of P inside Q. In other words, translation and scaling of P are allowed, but rotation is not. When P is convex and Q is an arbitrary polygonal environment, this problem is solved in time $O(kn\log(kn))$ by constructing a generalized Voronoi diagram of Q under a convex distance function induced by P.

Chew and Kedem [CK] follow a related approach to solve a more difficult variant of the problem, in which P is also allowed to rotate, which is also the main problem studied in this chapter. Instead of a Voronoi diagram, they compute the Delaunay triangulation of Q under the convex distance function induced by P at some arbitrary fixed orientation. By using a clever incremental technique for constructing all the topologically different triangulations obtained as the orientation of P varies, they solve the problem in time $O(k^4n\lambda_3(kn)\log n)$, where $\lambda_q(r)$ is the maximum length of an (r,q)-Davenport-Schinzel sequence (which is almost linear in r for any fixed q) [ASS, HS].

In this chapter we follow a different approach that applies the parametric search technique introduced by Megiddo [Me]. By exploiting efficient sequential and parallel algorithms for the fixed size containment problem, we solve the extremal problem in time $O(k^2n\lambda_4(kn)\log^3(kn)\log\log(kn))$.

There are two advantages of our technique over the technique of [CK]. First, our solution is considerably faster than theirs when k is large — roughly two orders of magnitude faster. Second, the application of Megiddo's technique to largest placement problems is so natural that it is surprising that no one has observed this connection before. Roughly speaking, a solution for the fixed-size problem allows us to determine whether any specified expansion ratio is too large or too small. This, plus an efficient parallel version of the fixed size containment algorithm, is all that is required for Megiddo's technique to apply (see below for more details). We demonstrate the generality of our approach by considering several other extremal containment problems, and show that Megiddo's technique applies to all of them. Specifically we consider the extremal versions of the following problems: placing a convex polygon in another convex polygon under translation, placing two convex polygons in a third convex polygon under translation, placing a triangle in a convex polygon under translation and rotation, and finally the general case of placing a convex polygon in a polygonal environment under translation and rotation. Except for the general case, these problems were never solved before. Some additional possible extensions of the technique are discussed at the end of the chapter.

The chapter is organized as follows. In section 2.2 we investigate some simple versions of the extremal polygon containment problem, involving one and two polygons, and allowing the polygons only to translate. Section 2.3 presents a reduction of some polygon containment problems to linear programming problems. Section 2.4 is devoted to a simple version of the general case, in which we also allow rotation; we study the placement of a triangle in a convex polygon. This can be regarded as a warm-up exercise that sheds some light on the general and more complex algorithm. In section 2.5 we state some neccesary definitions and results from [LS], and note that the combinatorial bound derived in that paper can be somewhat improved. In section 2.6 we describe a variant of the fixed size containment algorithm from [KS], which we use as a decision procedure, to decide whether a copy of P having some fixed expansion ratio can be placed in Q. In section 2.7 we give a parallel version of the fixed size containment algorithm. In section 2.8 we show how to combine the algorithms of section 2.6 and 2.7 to produce an algorithm for the largest placement problem. We conclude with a discussion of our results and some open problems.

2.2 Placement of Polygons Under Translation

In this section we investigate problems of placement of the largest homothetic copies of polygons inside another polygon (i.e. allowing translations only).

Definitions: We denote the set of translations of P that place it inside Q by $\mathcal{C}(P,Q)$, and the set of translations of P that make it intersect Q by $\mathcal{O}(P,Q)$

We will assume that the polygons P and Q are given as an array of their vertices in counter clockwise direction, $P = (p_1, \ldots, p_k)$ and $Q = (q_1, \ldots, q_n)$. We will consider Q as fixed and P as movable, and we will use the vertex p_1 as a reference point for P.

Proposition 1 ([Ch]) *If P and Q are convex, then $\mathcal{C}(P,Q)$ is a convex polygon with at most n edges.*

Proposition 2 ([GRS]) *If P and Q are convex, then $\mathcal{O}(P,Q)$ is a convex polygon with at most $n+k$ edges.*

2.2.1 Computation of $\mathcal{C}(P,Q)$

We assume that both P and Q are convex. The procedure given below for the computation of $\mathcal{C}(P,Q)$ is taken from Chazelle [Ch].

1. For each edge q_iq_{i+1} of Q, we find the vertex p_i of P that is nearest to it, when P lies completely on the same side of the line q_iq_{i+1} as Q (there may be two such vertices, in which case we choose one of them arbitrarily).

   This may be done in time $O(n+k)$ by merging the normal diagrams of P and Q, i.e., merging the edges of P and Q to a single list sorted by slope, and finding for each edge of Q between which edges of P it lies in the merged list.

2. For every $i = 1, 2, \ldots, n$ we place P so that p_i lies on the edge q_iq_{i+1}, and P and Q lie on the same side of the line q_iq_{i+1}. Now we draw a line t_i parallel to q_iq_{i+1} and passing through the reference point p_1 of P.

   Let ht_i denote is the half plane that lies below t_i if Q lies below the line q_iq_{i+1} and above t_i otherwise. The computation of all the ht_i takes $O(n)$ time, a constant time for each half plane.

3. As shown in [Ch], $\mathcal{C}(P,Q) = \bigcap_i ht_i$, so what remains to do is to compute the intersection of the n half planes. We note that the half planes are given sorted by their slope. We compute the intersection by solving the dual convex hull problem, and the sorting of half planes by slope gives us a convex hull problem of n points sorted by their x-coordinate. This problem can be solved in $O(n)$ time, using the beneath-beyond algorithm.

We conclude that the computation of $\mathcal{C}(P,Q)$ can be done in $O(n+k)$ time.

In order to apply the parametric search technique of Megiddo, we need a parallel version of this algorithm. Step 1, the merging of the normal diagrams, could be performed in parallel in $O(\log\log(\min(n,k)))$ parallel time using $\sqrt{nk}$ processors, using Valiant's algorithm [Va]. However, the normal diagrams of P and Q are independent of the expansion ratio, so no comparison that this merge generates depends on $\delta^\star$. We can thus implement this step sequentially, "outside" the generic scheme of Megiddo. Step 2 involves no comparisons, so it too can be performed sequentially. The coefficients of the t_i's will be however functions of the expansion ratio of P. Step 3 is performed in parallel using the parallel algorithm for computing the convex hull of a plane point set, by Aggarwal et al. [ACGOY], that works in $O(\log n)$ time and uses $O(n)$ processors.

We now combine the sequential and parallel algorithms to obtain an algorithm that computes the largest homothetic copy of P that can be placed in Q. Note that the problem at hand satisfies the requirements of Megiddo's technique, that is, when the expansion ratio δ is smaller than some (unknown) value $\delta^\star$, there is a placement of P inside Q, and when $\delta > \delta^\star$ there is no such placement. We run the generic parallel algorithm, without specifying δ. We resolve comparisons needed by the algorithm by computing the set of real roots of the characteristic polynomials associated with the comparisons, and locating $\delta^\star$ in this (ordered) set by binary search. The decisions made during the binary search are based on the outcome of the fixed-size algorithm, applied to a copy of P with expansion ratio equal to the root δ being compared. Note that the decision step only tells us whether $\delta \geq \delta^\star$ or $\delta < \delta^\star$. In order not to get stuck, we interpret $\delta \geq \delta^\star$ as $\delta > \delta^\star$ and continue in this manner. When the entire algorithm terminates, it will have produced an interval I so that $\delta^\star$ is either its left endpoint or an interior point. However, the second case is impossible, because the output of the generic algorithm is the same for all $\delta \in \mathrm{int}(I)$, but the output must change at $\delta^\star$, by definition. Hence $\delta^\star$ is the left endpoint of I.

The running time of the algorithm is $O(n+k)$, for the initial step 1 performed just once, plus the cost of the parametric search itself, which is $O(n\log^2 n)$. We thus obtain:

Theorem 1 *Given a convex polygon P with k vertices and a convex polygon Q with n vertices, we can compute a placement of the largest homothetic copy of P inside Q in $O(k + n\log^2 n)$ time.*

Remark. As noted by Chazelle [Ch], this will work even if P is not convex, because in this case we simply apply our algorithm to $\mathrm{conv}(P)$ instead of P.

Remark. In section 2.3 we show how to improve the running time to linear.

2.2.2 Computation of $\mathcal{O}(P, Q)$

As shown by Guibas et al. [GRS], the calculation of $\mathcal{O}(P, Q)$ in the case of two convex polygons P and Q amounts to the merging of the lists of their edges sorted by slope.

This takes time $O(n+k)$ using a serial algorithm, or $O(\log\log(\min(n, k)))$ parallel time using $\sqrt{nk}$ processors, using Valiant's algorithm [Va].

2.2.3 Finding Largest Homothetic Placements of Two Convex Polygons Inside a Third

We now consider the following problem. Given two convex polygons P_1 and P_2 having k_1 and k_2 vertices respectively, and a third convex polygon Q having n vertices, find the largest expansion ratio α such that αP_1 and αP_2 can be translated into Q without overlapping each other.

For the fixed-size containment problem we use the procedure given by Avnaim and Boissonnat [AB] and Guibas et al. [GRS]. The procedure computes the set U of all the valid translations T_r of P_1 relative to P_2, for which there exists a translation that position both P_1 and P_2 in Q in their valid relative position without overlapping. This is done by several consecutive applications of the primitive operations $\mathcal{C}$ and $\mathcal{O}$:

1. Compute $C_1 = \mathcal{C}(P_1, Q)$.
2. Compute $C_2 = \mathcal{C}(P_2, Q)$.
3. If C_1 or C_2 is empty then return $\emptyset$.
4. Compute $I = \mathcal{O}(P_2, P_1)$.
5. Compute $S = \mathcal{O}(C_2, C_1)$.
6. Compute $S^\star$, the polygon symmetric to S with respect to the origin.

7. Return $U = S^\star \setminus I$.

The correctness of this algorithm is proved in [AB], and we repeat it here for the sake of completeness.

Lemma 2 ([AB]) *The set of all the valid relative positions is given by* $U = S^\star \setminus I$.

Proof. Assume that u is a valid relative translation of P_2. We have to show that $u \in U$. If u is valid, then P_1 and P_2 do not collide, so certainly $u \notin I$. However as u is a valid relative translation, it is the difference between two valid absolute translations $u = u_1 - u_2$, where u_1 is a translation of P_1 and u_2 a translation of P_2. Since these are valid absolute translations, $u_1 \in \mathcal{C}_1$ and $u_2 \in \mathcal{C}_2$. In addition, if we translate P_2 by $-u$, we should be able to translate both polygons into Q using the translation u_1, and this means that $\mathcal{C}_2$ translated by $-u$ should intersect $\mathcal{C}_1$, or $-u \in S = \mathcal{O}(\mathcal{C}_2, \mathcal{C}_1)$. Thus $u \in S^\star$ but $u \notin I$. This argument shows the other direction as well and the proof is complete.

From the propositions above and the descriptions of algorithms for the computations of $\mathcal{C}$ and $\mathcal{O}$, it follows that the running time of this algorithm is $O(n + k_1 + k_2)$.

The parallel version of the algorithms for computing $\mathcal{C}$ and $\mathcal{O}$ can be used for performing steps 1–5 of the algorithm above. Step 6 does not involve comparisons, so we need not perform it in parallel. Step 7 is more difficult to handle, but we exploit the fact that we are only interested in the existence of a translation in U, not in its full structure. So instead of computing U, we will only decide in step 7 whether $U = S^\star \setminus I$ is empty or not. As both $S^\star$ and I are convex polygons, the difference is not empty if and only if the convex hull of $S^\star \cup I$ is not simply I. This is so because $S^\star \setminus I$ is empty if and only if $S^\star$ is contained in I, and this is true if and only if the convex hull of their union is I. So computing the convex hull of $S^\star \cup I$ is sufficient to decide on the non-emptiness of U, and this computation can be performed in $O(\log(n + k_1 + k_2))$ parallel time using $O(n + k_1 + k_2)$ processors.

Applying the parametric search paradigm, we obtain

Theorem 2 *Given two convex polygons P_1 and P_2 with k_1 and k_2 vertices respectively, and a convex polygon Q with n vertices we can compute disjoint placements of the largest homothetic copies of P_1 and P_2 inside Q (with* *the same expansion ratio), without intersecting each other in* $O((n + k_1 + k_2)\log^2(n + k_1 + k_2))$ *time.*

The assumption that the expansion of P_1 and P_2 is the same is not necessary; we only have to assume that the expansion ratios of the two polygons are expressed by two monotone increasing functions of the same parameter, $f_1(\alpha)P_1$, $f_2(\alpha)P_2$.

2.3 Placing Polygons Using Linear Programming

In cases where we have to place a translated copy of one polygon in another convex polygon a much more efficient approach can be taken. In these cases we can reduce the polygon placement problems to linear programming problems. We are indebted to Nimrod Megiddo and independently to Alon Efrat for pointing this out.

We are given two polygons, a convex polygon $Q = (q_1, \ldots, q_n)$ and a simple polygon $P = (p_1, \ldots, p_k)$. Without loss of generality assume that P is also convex (otherwise replace P with $\text{conv}(P)$). We have to place a copy of P inside Q. The transformations allowed (translation, translation and scaling etc.) will be specified later.

Assume that we seek a placement of the largest homothetic copy of P in Q. The copy of P is thus $\alpha P + b$ where $\alpha > 0$ is a real expansion ratio and b is a translation vector. As in section 2.2.1 we only need to ensure that P is on the "right" side of each edge of Q, and this holds if the vertex of P nearest to that edge is on the "right" side of the edge (the edge-vertex pairing can be computed in time $O(n + k)$). That is, we will have n inequalities that ensure that the copy of P is inside Q,

$$a_i^T(\alpha p_i + b) \leq 1$$

for $i = 1, \ldots, n$, with the linear goal function $\max \alpha$. The a_i's and p_i's are constants. The variables are the vector b and the expantion-ratio α.

This linear problem can be solved in $O(n)$ time using Megiddo's linear programing algorithm [Me1, Me2] and the overall running time is thus $O(n + k)$.

The technique can also be applied to 3-dimensional convex polyhedra. As before, we find the nearest vertex of P for each face of Q, and solve the linear programming problem

$$a_i(\alpha p_i + b) \leq 1$$

for $i = 1, \ldots, n$, with the linear goal function $\max \alpha$.

Finding the nearest vertices of P to faces of Q can be done by computing the normal diagram of P (which is actually a planar map on a sphere), preparing it for fast point location, and locating the normal direction of each face of Q in this map. Computing the normal diagram can be done in linear time, given a reasonable representation of the polyhedra (e.g. a quad-edge structure), preprocessing can be done in $O(k)$ time using the technique of [EGS], and the point location queries can be all done in $O(n \log k)$ time. Thus the overall running time of the algorithm is $O(k + n \log k)$.

It is obvious that the technique can be applied to fixed-size queries as well. We run the same procedure as above. If the resulting expansion-ratio is smaller than 1 then a copy of P cannot be placed in Q, otherwise it can.

We thus obtain the following result:

Theorem 3 *Given a convex polygon P with k vertices and a convex polygon Q with n vertices, we can compute a placement of the largest homothetic copy of P inside Q in $O(k + n)$ time. In three dimensions, we can do the same for convex polytopes in time $O(k + n \log n)$.*

2.4 Placing a Triangle Under Translation and Rotation

Before we tackle the general problem of extremal containment of a convex polygon in a general polygonal environment, we consider a restricted version in which we compute the largest similar copy of a triangle $T = ABC$ in a convex polygon $Q = (q_1, \ldots, q_n)$.

This (fixed size) containment problem was studied by Chazelle [Ch]. He observed that there is a free placement of T in Q if and only if there is a placement of T in Q in which a vertex of T and a vertex of Q coincide. Thus in order to test if there exists a free placement of T in Q, we go over all the $3n$ pairs of a vertex of T and a vertex of Q and for each pair test if there is a free placement such that the relevant vertices coincide.

When the vertices of such a pair, say A and q_1, coincide, we use the angle θ of rotation around the fixed vertex A of T to describe the placement of T. The placement is free iff both edges AB and AC lie in the half-planes whose intersection is Q. As B and C can intersect the line defining a half-plane only twice when T rotates around A, we can generate an interval of placements (= angles) that are free relative to that half-plane. The intersection of all n intervals is the set of free placements. This intersection can be computed in time $O(n)$ per pair, or $O(n^2)$ overall.

The parallel version works by sorting all the endpoints of all free intervals for each vertex-vertex contact. Note that the parallel algorithm does not solve the same problem as the sequential one. However, the output of the sorting algorithm changes combinatorially at $\delta^\star$. As indicated in Section 1.5 this is all that is required from the "generic" algorithm. This takes $O(\log n)$ time and uses $O(n)$ processors per pair [Co], or $O(\log n)$ time and $O(n^2)$ processors overall. Thus we obtain:

Theorem 4 *Given a triangle T and a convex polygon Q with n vertices, we can compute a placement of the largest possible similar copy of T inside Q in time $O(n^2 \log^2 n)$.*

The discussion above is similar in nature to the solution of the general case given below. The increased complexity caused by allowing rotations prevents us from computing the set of all possible free placements as we did when translation alone was allowed. Instead we restrict our attention to a distinguished subset of "critical" free placements that necessarily exist if any free placement exists. There is also an analogy between computing the intersection of relatively free intervals to find a free placement, and the use of lower envelopes below. The details of the general case, however, are much more complex.

2.5 The General Case — Finding Free Critical Orientations

We now begin the description of our solution to the general case. In this section we give a short exposition of the definitions and results in [LS]; this is needed in order to present the algorithms in subsequent sections. The material is taken almost verbatim from [LS].

Let P be a convex polygonal object having k vertices, free to translate and rotate (but not to change its size) in a closed two-dimensional space Q bounded by a collection of polygonal obstacles ("walls") having altogether n corners.

A *free critical* placement of P is one at which it makes simultaneously three distinct contacts with the walls, and is fully contained in Q, so that it cannot penetrate any obstacle.

A (potential) *contact pair* O is a pair (W, S) such that either W is a (closed) wall edge and S is a corner of P or W is a wall corner and S is a (closed) side of P. The contact pair is said to be of type I in the first case, and of type II in the second case.

An actual *obstacle contact* is said to *involve* the contact pair $O = (W, S)$ if this contact is of a point on S against a point on W, and, furthermore, if this contact is *locally free*, i.e., the inner angle of P at S lies entirely on the exterior side of W if S is a corner of P, and the entire angle within the wall region Q^c at W lies exterior to P if W is a wall corner.

The *tangent line* T of a contact pair $O = (W, S)$ is either the line passing through W if W is a wall edge or the line passing through W and parallel to S if S is a side of P (in the second case T depends of course on the orientation of P).

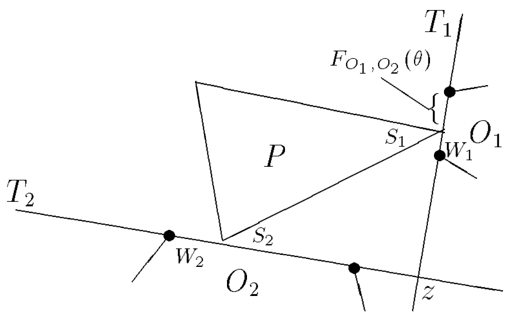

Figure 2.1: A bounding function.

Let O_1, O_2 be two contact pairs. We say that O_2 *bounds* O_1 at the orientation θ if the following conditions hold (see Figure 2.1):

1. There exists a (not necessarily free) placement $Z = (X, \theta)$ of P at which it makes two simultaneous obstacle contacts involving O_1, O_2.

2. As we move P from Z without changing the orientation θ, along the tangent T_1, in the direction of the intersection z of the two tangents T_1 and T_2, the subset $P^\star = \mathrm{conv}(S_1 \cup S_2)$ of P intersects W_2 until S_1 touches W_1.

It is shown in [LS] that for any double obstacle contact, one of the contact pairs always bounds the other. Let O_1 be any contact pair and consider all contact pairs that bound O_1 (at any orientation θ). For each such pair O_2 we define the *bounding function* $F_{O_1,O_2}(\theta)$ over the domain $\Pi = \Pi_{O_1,O_2}$ of orientations θ of P in which O_2 bounds O_1. For each $\theta \in \Pi$, we define $F_{O_1,O_2}(\theta)$ to be the distance from the endpoint of the contact wall farthest from z (the intersection of the tangents) to the contact point of O_1, at the placement $Z = (X, \theta)$ in which P simultaneously makes two obstacle contacts involving O_1, O_2; (see Figure 2.1). Note that Π need not be connected, but it consists of at most five subintervals (this is proved in [LS], Lemma 2.2).

The dependence of the bounding function on a specific endpoint of the contact wall suggests that we group the bounding functions F_{O_1,O_2} of O_1 into two classes, A_L and A_R, so that in each class the functions are related to the same endpoint of the contact wall of O_1.

With each class A_E, $E \in \{L, R\}$, of each contact pair O_1, we associate a function

$$\Psi_{E;O_1}(\theta) = \min\{F_{O_1,O}(\theta) : F_{O_1,O} \in A_E\}.$$

This is the *lower envelope* of the functions in A_E. An intersection of two bounding functions of the same class, F_{O_1,O_2} and F_{O_1,O_3}, that lies on the lower envelope of that class, is called a *breakpoint* of the lower envelope.

Critical free orientations (i.e. orientations of critical free placements) can arise in three situations. The *first kind* of orientations is of critical placements at which two contact pairs simultaneously bound a third one, and both belong to the same class. Each such placement is represented as a breakpoint on some lower envelope. The *second kind* of orientations arise at critical placements where two contact pairs bound a third one but belong to different classes. The *third kind* of orientations arise when no two contact pairs bound a third, but rather at critical placements involving three contact pairs O_1, O_2 and O_3 so that O_1 bounds O_2, O_2 bounds O_3, and O_3 bounds O_1. see Figure 2.2 for an illustration.

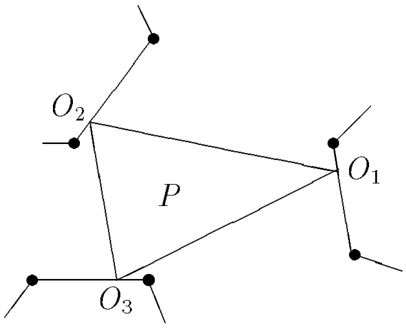

Figure 2.2: A critical contact of the third kind.

These are necessary conditions for a critical free placement of P, that is, one of the three situations must occur at a critical free placement. However, they are not sufficient, and while our algorithm will find every orientation of any of the three kinds, it must also be able to discard critical placements that are not free.

Remark. In [LS] it is proved that the number of breakpoints along one lower envelope is $O(\lambda_s(kn))$ for some fixed $s \leq 6$ (see the remark after Lemma 2.3 in [LS]). We give a simple argument that shows $s \leq 4$.

Our argument relies on the fact that we can partition the functions $F_{O,O'}$ in a class $A_E(O)$ into two subsets, one arising from contacts O' of type I and the other from contacts O' of type II. As shown in [LS], two functions from the same subset intersect at most twice, and functions from different subsets intersect at most four times. Hence the lower envelope of functions in the same subset has complexity $O(\lambda_4(kn))$ (since they are only partial functions) and since the final envelope of the class is the envelope of these two sub-envelopes, it easily follows that it too has complexity $O(\lambda_4(kn))$.

Remark. The analysis of [LS], when turned into an algorithm, can produce a list of all these critical placements in time $O(kn\lambda_4(kn)\log(kn))$. However, detecting which of these placements is indeed free is not straightforward. In the context of motion planning, as studied in [KS], it is possible to sift out the critical orientations and obtain a subset of *free* critical placements that include all placements reachable from a given initial placement, and perhaps some other non-reachable but free placements. This can be done within the same time bound, $O(kn\lambda_4(kn)\log(kn))$, but cannot guarantee that *all* free placements are found, and is therefore unsuitable for our purpose. This issue is discussed in the algorithms that we give below. In our solution, we do detect all free critical placements, at an extra cost of $O(k \log n)$ per placement. Performing this faster still appears to be an open problem.

2.6 A Sequential Algorithm

In this section we present a sequential algorithm for the fixed size containment problem, that is to determine whether it is possible to place a similar copy of the convex polygonal object P, at some fixed expansion ratio to the original P, in the polygonal environment Q. To solve this decision problem we solve a related problem — finding all the *critical free orientations* of P. If the set of critical free orientations is empty, the solution to our decision problem is "no", otherwise the answer is "yes".

2.6.1 Generating All the Critical Placements

Below we give the algorithm that generates all the critical placements. Each one of them needs to be tested to decide whether it is free, using the algorithm of the next subsection.

The algorithm closely follows the first stages of the algorithm in [KS]. However, the data structures used are simpler, to ease the task of parallelizing the algorithm later. We do not consider critical contacts in which P has only one degree of freedom (a corner of P against a corner of Q, an edge of P against an edge of Q), because they can be handled exactly like the triangle in section 2.4, that also has one degree of freedom in every critical placement.

Step 1: Find all bounding functions. For every two contact pairs O_i, O_j, find the range of orientations Π_{O_i,O_j} in which O_j bounds O_i toward a specific endpoint E of O_i. Split the resulting bounding functions F_{O_i,O_j} into (at most five) "subfunctions", each defined over a connected interval, and add them to the appropriate collection $A_L(O_i)$ or $A_R(O_i)$.

Step 2: Calculate lower envelopes. We describe the calculation of the lower envelope of $A_L(O)$ which is denoted by $\Psi_{L;O}$; $\Psi_{R;O}$ is calculated similarly.

1. Fix a contact pair O and partition $A_L(O)$ into two disjoint subsets A'_L and A''_L of roughly equal size.

2. Compute recursively the two lower envelopes

$$\Psi'(\theta) = \min\{F_{O,O_i}(\theta) : F_{O,O_i} \in A'_L\}$$

$$\Psi''(\theta) = \min\{F_{O,O_i}(\theta) : F_{O,O_i} \in A''_L\}.$$

   Each of the recursive calculations produces a sequence of angular intervals, delimited by breakpoints, in each of which the corresponding partial lower envelope is attained by a single bounding function.

3. Merge these two sequences of intervals to obtain a refined sequence Γ of angular intervals. In the merging process mark every breakpoint in Γ as red if it was originally a breakpoint of Ψ' or as black if it was originally a breakpoint of Ψ''. In addition, maintain a pointer from each red node in Γ to the black interval it lies in (an interval of Ψ'') and from each black node to the red interval it lies in. For each interval $I \in \Gamma$ there exist two unique contact pairs O', O'' with $F_{O,O'} \in A'_L$, $F_{O,O''} \in A''_L$ such that $\Psi'(\theta) = F_{O,O'}(\theta)$, $\Psi''(\theta) = F_{O,O''}(\theta)$ for each $\theta \in I$. By the analysis of [LS] the two functions $F_{O,O'}, F_{O,O''}$ intersect in at most four points (some of which may not belong to I), which can be calculated, as the roots of some quartic polynomial, in constant time. Each of these intersections which lies in I is clearly a breakpoint of $\Psi = \Psi_{L;O}$. Add these points to Γ and mark them as white nodes. Every breakpoint of Ψ is either of this kind (a white node) or is a breakpoint of Ψ' or of Ψ'', i.e. one of the red or black nodes. Now we need to eliminate from Γ the red and black nodes which do not lie on the lower envelope Ψ. For each red (black) node, we follow the pointer to the black (red) interval it lies in, and check which is higher — the red breakpoint in Ψ' or the bounding function on Ψ''. If the former is higher we prune it from the list Γ, otherwise the breakpoint remains in Γ. Thus at the end of the process Γ represents the breakpoints in Ψ.

Note that maintaining the red/black pointers can be done in time proportional to the length of the list (in one pass over the list), and the same time bound applies to the pruning of the redundant nodes.

The merging step can be done in time proportional to the length of Γ, which, by [LS1] and the comments in Section 2, is $O(\lambda_4(kn))$. Hence the calculation of the lower envelope $\Psi_{L;O}$ takes $O(\lambda_4(kn) \log kn)$ time, so all these envelopes can be computed in overall time $O(kn\lambda_4(kn) \log kn)$.

The collection of breakpoints is a superset of all the critical orientations of the first kind; every one of them will later be tested to decide whether it is free, in the manner described in the next subsection.

Step 3: Calculate critical orientations of the second kind. These are orientations at which P makes simultaneously, at some free placement, obstacle contacts involving three distinct contact pairs O_1, O_2, O_3 such that two of them, say O_2, O_3 bound O_1 but with $F_{O_1,O_2} \in A_L(O_1)$ while $F_{O_1,O_3} \in A_R(O_1)$. In this case we first reflect and translate one of the envelopes, so that they both measure the distance from the same endpoint of O_1. Then we merge the lists of breakpoints in $\Psi_{L;O_1}$ and in $\Psi_{R;O_1}$ and compute the intersections of the bounding functions from the two lower envelopes over each resulting interval in the same way as in the previous step. These orientations are added to the list of critical orientations.

Clearly, this step runs in $O(kn\lambda_4(kn))$ time. Again, we will later discard non-free critical orientations found in this step.

Step 4: Calculate critical orientations of the third kind. Finally, we calculate the third and most complex kind of critical orientations. At each such orientation θ, P can make simultaneously a free triple contact involving three distinct contact pairs O_1, O_2, O_3, such that $F_{O_1,O_2} \in A_{E_1}(O_1)$, $F_{O_2,O_3} \in A_{E_2}(O_2)$, $F_{O_3,O_1} \in A_{E_3}(O_3)$, where $E_i \in \{L, R\}$ for $i = 1, 2, 3$, and such that all three functions lie at θ on the corresponding lower envelopes.

To find these orientations we first merge all breakpoint lists from all the lower envelopes calculated in step 2, to obtain a single sorted list Φ consisting of $O(kn\lambda_4(kn))$ refined noncritical intervals. Each interval $I \in \Phi$ has the property that each lower envelope is attained over it by a single bounding function.

Next we find all the critical orientations of the third kind (not necessarily free). For each possible triple contact that the algorithm considers, we find its (at most four) critical orientations, and then test which of these orientations is indeed free.

We start by considering the first interval in Φ, denoted I_0. For each contact pair O_1 and each side $E_1 \in \{L, R\}$, find the unique contact pair O_2 such that $\Psi_{E_1;O_1} = F_{O_1,O_2}$ over I_0. For each $E_2 \in \{L, R\}$, find the unique contact pair O_3 such that $\Psi_{E_2;O_2} = F_{O_2,O_3}$ over I_0. For each $E_3 \in \{L, R\}$ for which $\Psi_{E_3;O_3} = F_{O_3,O_1}$ over I_0 conclude that $(O_1, O_2, O_3, E_1, E_2, E_3)$ is a critical contact, perhaps not free. Compute its critical orientations. Those that lie in I_0 are tested to decide whether they are free, and if so they are reported as such (the algorithm can thus be halted right now with an affirmative answer to the decision problem). The other orientations do not lie in I_0, so we find the interval each of them lies in, by binary search over the sorted list Φ, and test whether the corresponding critical placement is free. This takes $O(kn \log(kn))$ time, excluding the tests for being free, $O(\log(kn))$ time for each contact pair.

Each interval $I \neq I_0$ in Φ can induce new critical triplets but fortunately only a constant number of them. The interval I was formed because its left endpoint represents a break in one of the lower envelopes, say $\Psi_{L;O_1}$. So we need to repeat the process we did at I_0, but this time starting with only one particular contact (O_1) and one lower envelope $(\Psi_{L;O_1})$. Thus every interval I induces only $O(1)$ new candidates for the critical orientations that we seek. Finding the bounding functions on the lower relevant envelopes is now accomplished by a binary search over the list of breakpoints on each envelope. This takes $O(kn\lambda_4(kn) \log(kn))$ time, as each of the $O(kn\lambda_4(kn))$ intervals requires $O(\log(kn))$ time for the binary searches.

2.6.2 Deciding whether a critical orientation represents a free placement

As mentioned above, the set of critical orientations computed so far may contain orientations that correspond to critical placements that are not free, so we need to test each critical placement whether it is indeed free.

To perform this test we use the following simple method. In a preliminary step, we prepare data structures that will enable us to perform this test, at any query placement of P, in time $O(k \log n)$. These data structures depend only on Q. As we are required to perform at most $O(kn\lambda_4(kn))$ such tests, the total time they require is $O(k^2n\lambda_4(kn) \log n)$. As the environment Q is static during the execution of the (largest placement) algorithm, we need to build the data structures only once, "outside" the generic execution of the parallel version of the algorithm.

If a critical placement is not free, then either a vertex of Q lies inside P, or an edge of P intersects an edge of Q. To test whether the first situation occurs we insert the n vertices of Q into a data structure that supports fast counting of points inside a query polygon. We use the technique of [PY, Ed], which uses $O(n^2)$ storage, $O(n^2)$ preprocessing, and can answer a query in time $O(k \log n)$ for a query polygon with k sides. Given a placement of P, we query the number of points inside it and declare the placement non-free if any such point is found. To test whether the second situation arises, we preprocess Q for segment intersection queries, that is, given a query segment, determine quickly whether it intersects an edge of Q. For this we use the technique of [Ch1]. Again, this can be implemented with $O(n^2)$ storage, $O(n^2)$ preprocessing, and can answer a segment intersection query in $O(\log n)$ time. For each critical placement of P, we query this structure with each edge of P, and declare the placement as non-free if any such intersection is found. If none of these bad situations are detected, the placement is free.

We have thus shown:

Theorem 5 *Given a convex polygon P with k sides, and a polygonal environment Q with n edges, we can compute all free critical placements of P inside Q in time $O(k^2 n \lambda_4(kn) \log(kn))$.*

2.7 A Parallel Algorithm

We now present a parallel version of the algorithm, or rather comment on how to perform each step in parallel. Recall that we do not need a strong parallel computation model. All we seek is a scheme in which many independent comparisons are performed at each parallel step. Thus we ignore synchronization and other bookkeeping problems, use Valiant's weak model of parallel computation [Va], and perform tasks in a sequential manner when they do not involve comparisons but only manipulation of pointers.

Step 1 can clearly be carried out by $O(k^2 n^2)$ processors in $O(1)$ parallel time, with each processor calculating one bounding function.

Step 2 is performed using a divide and conquer strategy. The divide phase and the recursive calls can be done in parallel. The merge phase can be done using Valiant's merging algorithm [Va] which runs in $O(\log \log(kn))$ parallel time using $O(\lambda_4(kn))$ processors per envelope. Once the merge is done, maintenance of the red/black pointers can be done serially because this is a mere manipulation of pointers and involves no comparisons. The testing of red (black) breakpoints against their containing black (red) non-critical intervals can be done in $O(1)$ parallel time using $O(\lambda_4(kn))$ processors per envelope. The subsequent pruning of breakpoints can be done serially, because it involves no comparisons. The total time required to compute all the lower envelopes is thus $O(\log(kn)\log\log(kn))$ using $O(kn\lambda_4(kn))$ processors.

Step 3 also uses a merge, but only once for each contact pair, and the calculation of envelope intersections over all contact pairs can clearly be done in parallel. The total parallel time for this step is therefore $O(\log\log(kn))$ using $O(kn\lambda_4(kn))$ processors.

Step 4 first requires the merging of all $O(kn)$ lower envelopes into one sorted list of breakpoints Φ. This can be done recursively. We divide the lower envelopes into two collections of roughly equal size, compute the two merged lists of breakpoints Φ', Φ'' and merge them. The merging step takes $O(\log\log(kn))$ parallel time using $O(kn\lambda_4(kn))$ processors, so the whole process takes $O(\log(kn)\log\log(kn))$ time using $O(kn\lambda_4(kn))$ processors.

The handling of the first interval $I_0 \in \Phi$ can be done in parallel using $O(kn)$ processors and $O(\log(kn))$ time. All the other intervals are each assigned a single processor and the time it takes to find the new critical triple contacts is $O(\log(kn))$. The test to decide whether a critical orientation is free is performed in the same manner as in the sequential case. We use k processors to perform the $O(k)$ queries in $O(\log n)$ time. The data structures depend only on Q and so can be preprocessed just once, outside the generic parallel scheme.

We conclude that all critical free orientations can be calculated in parallel, under Valiant's comparison model, in time $O(\log(kn)\log\log(kn))$ using $O(k^2n\lambda_4(kn))$ processors.

2.8 The Overall Algorithm

We now apply Megiddo's technique to our problem, using the algorithms of sections 2.6 and 2.7. We run the parallel algorithm generically, without specifying the expansion ratio δ. We resolve comparisons made by the parallel algorithm by using our sequential algorithm, in the manner explained in the Introduction chapter.

The only fine point is to verify that comparisons involve only evaluations of signs of low degree polynomials in the unspecified δ. Indeed, a free placement of P in Q has to satisfy a set of algebraic constraints (see [SS]). In our case these constraints are mainly algebraic inequalities that describe the disjointness of P and Q. Computing a critical triple contact amounts to setting three inequalities as tight constraints (i.e. equalities), and solving these three equations in three unknowns (the (x, y, θ) coordinates of P), discarding solutions which are not locally free. Computing a breakpoint thus reduces to computing the critical triple contact placement associated with the three contact pairs that define the breakpoint. Evaluating a bounding function at a given orientation amounts to setting the two constraints involved in the corresponding two contact pairs to be tight, and adding a third constraint that the slope of the line passing between two fixed points in P will be at the given orientation. This will give us the desired placement of P, and we can calculate the value of the bounding function which is simply an affine transformation of the placement. Using the standard transformation $t = \tan(\theta/2)$, all contact constraints, and thus all functions of δ computed by the algorithm, become algebraic, and no trigonometric functions need be used.

Since the only place where dependence on δ can arise is in the coefficients of the constraints, and since the functions of δ are polynomials of the first or second degree, we are assured that all the equations in δ we have to solve during the algorithm are algebraic equations of bounded degree. We assume that this kind of equations can be handled in constant time.

The running time of the algorithm can easily be deduced by "plugging in" the running time of the sequential and parallel algorithms into the analysis of Megiddo's technique. The fact that $\delta^\star$ is the left endpoint of the final interval is justified as in section 1.5. This establishes our main result:

Theorem 6 *Given a convex polygon P with k sides, and a polygonal environment Q with n edges, we can compute a placement of the largest possible similar copy of P inside Q in time $O(k^2 n \lambda_4(kn) \log^3(kn) \log\log(kn))$.*

2.9 Conclusions

In this chapter we have applied Megiddo's parametric search technique to a variety of extremal polygon placement problems. In addition, we presented a decision algorithm for the general fixed-size polygon containment problem, which improves upon results obtained in previous papers that studied related problems.

Our work raises a few open problems. One is to improve our algorithm by about an order of k, bringing its complexity close to the motion-planning algorithm of [KS]. We believe that Megiddo's technique can be applied to many other extremal containment problems. As an example we mention the problem of finding the largest stick (line segment) that can be placed inside a simple polygon, and the problem of finding the largest stick that can be placed in a polyhedral environment in 3-space. Finally, can our technique be turned into a motion-planning algorithm, that finds a "highest-clearance" path among obstacles, as in [CK1]?

Chapter 3

Experimental Results

3.1 Goals and Overview

This chapter presents experimental results concerning parametric search algorithms. The experiments had two major goals:

- To study the feasibility and complexity of coding algorithms involving parametric searching. In particular, to develop techniques to code versions of parallel algorithms that are intended to serve as the "generic" algorithm in a parametric search algorithms.

- To study experimentally the behavior of parametric search algorithms. This is especially important because it is clear that there are heuristics that lower the running time of the algorithms, yet are difficult to analyze. In addition, the behavior of parametric search algorithms incorporating a sequential "generic" algorithms was studied.

Two algorithms were coded. These are the simple preliminary example from the Introduction (Section 1.2) and the slope selection algorithm. The algorithms share the same generic parallel sorting algorithm of [Va] and parametric search service routines. Only the code that performs the serial comparisons is different. Therefore others algorithms that rely on a parallel sorting algorithms (including median finding as the preliminary example and maximum finding) can be implemented rather easily. Versions in which the generic algorithm is a sequential sort (using both merge sort and quick sort) were also coded.

All the programs are written in the C programming language.

3.2 The Structure of the Programs

In this section the general structure of the slope selection algorithm is described. This is meant mainly to describe the "skeleton" of a parametric search program, so less attention is paid to problem-specific routines.

3.2.1 The overall structure

The top level routine (after reading the input) is a simulation of a parallel sort. We have used Valiant's parallel sorting algorithm [Va]. The details of the simulation will be discussed in detail later.

The program keeps track of the level in which the parallel algorithms is in. That is, the program lets each "processor" try to perform one comparison. This comparison is not answered however, but collected by the program. After all the "processors" generate their comparisons, the program resolves all the comparisons gathered and runs the same parallel step again. This time all the questions are answered and the parallel algorithm can proceed to the next parallel step.

More specifically (see Figure 3.1), the main parallel sort is the routine `sim_parallel_sort`. When a comparison is about to be made in the preparatory stage (when questions are not yet answered) it calls the routine `add_comparison`. This routine will simply add the comparison to a data structure holding the set of comparisons (an array). When all "processors" have produced their comparisons, the sorting routine calls `resolve_comparisons` that resolves all the comparisons. We will describe how this is done later. In the second running of the parallel step, when all comparisons have been resolved, the sorting algorithm calls the routine `comparison` that returns the outcome of the comparison.

The routine `resolve_comparisons` implements the parametric search. In our program, it simply generates the array of test values (x-coordinates of intersections of pairs of lines), sorts this array and performs a binary search over it. The routine that does the actual binary search and is also responsible for updating the parameter bounds is `binary_search_parameter`. The decisions in the binary search are based on

- the relation of the test value to the parameter bounds, if is not inside the bounds, or

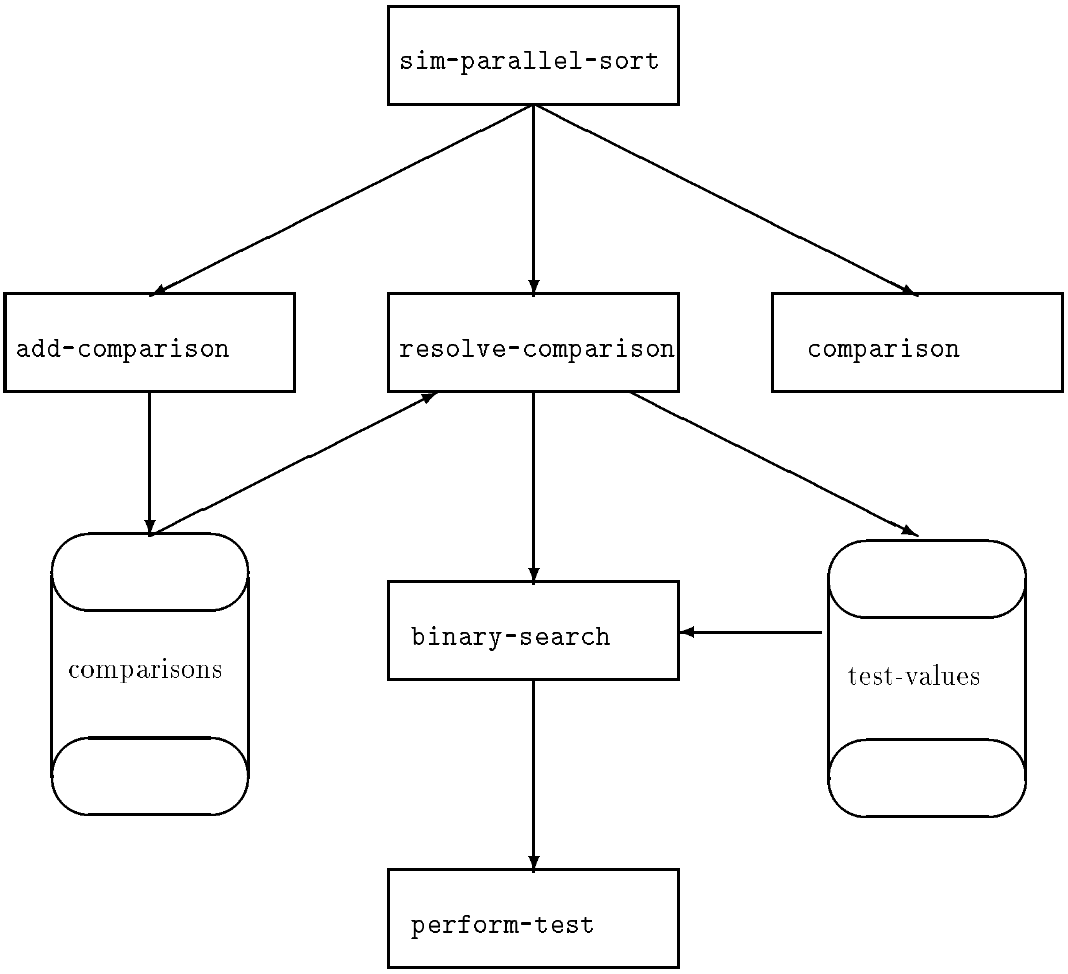

Figure 3.1: General structure of the program.

- the outcome of the sequential algorithm (in the slope selection case — generating a permutation and counting its inversions), obtained by calling the routine `perform_test`.

The level of problem dependency increases as we go down this calling chain, but we believe that problem-dependent code is still well separated from generic code in the program. The first problem specific detail (apart from the obvious requirement that the generic parallel algorithm is sorting) is in `resolve_comparisons` where it generates the set of test values from the comparisons collected. Here we make the assumption that the input data to be sorted are lines in the plane and that the test values are their intersections. The next and inevitable problem dependency is in `perform_test`, the routine that is supposed to do the serial test. Note that there is no problem specific coding in other routines such as `binary_search_parameter`.

3.2.2 Simulating the parallel sort

There are two problems in coding the simulation of the parallel sort. The first is to perform the algorithm in "levels" of parallel computation, although the algorithm of [Va] is specified as a recursive process. The second problem is a rather technical one, and it concerns the separation of comparisons that are going to be accumulated but not answered from comparisons that can be answered right away.

The solution to the first problem was to maintain data structures representing tasks to be performed in the next parallel step, and then to iterate on that data structure and perform the tasks one by one. These tasks generate of course the set of tasks for the next parallel step. As the parallel algorithm is Valiant's parallel merge sort, the tasks are always merging pairs of sorted lists. The algorithm begins with a set of n singletons, and in each iteration it groups the set into pairs, it then merges each pair, and transforms back the merged pairs into a flat set, that is regrouped and remerged until the entire set is sorted. The merges themselves are also done recursively, by cutting long lists into short fragments, merging pairs of fragments and concatenating the sorted fragments into a sorted list.

The second problem has a more technical flavor. We solved it by dividing each parallel step into phases. In each phase every processor asks at most one question (this is merely a more careful statement of Valiant's algorithm).

Between the phases operations other than comparisons are made (pointer manipulations and so on). To execute a parallel step (merging pairs of sorted lists in a suitable data structure) we call a routine that performs the first phase only. This routine however does not do the comparisons, but only notifies the parametric search algorithm that they need to be resolved (i.e. it calls `add_comparison`). Now all these comparisons are resolved by calling `resolve_comparisons`. The merging routine then calls another routine, that performs the first phase, performing actual comparisons as needed by calling `comparison`, and then collects the comparisons to be made in the second phase.

Next, a third-phase routine performs the first two phases and collects the comparisons of the third phase. This is continued until the last phase routine, which can now carry out the whole parallel step to completion.

3.3 Experiments

Experimentation with the programs included running the programs (for median root finding, our preliminary example and for slope selection) on random data. By random data we mean on lines $y = ax + b$ where all the real parameters (a's and b's) are drawn independently from some distribution. We have used only uniform distribution on $(0, 1)$ and exponential distribution with parameter 1.

For evaluating the results, we note that the asymptotic running time bounds of both algorithms are $O(n \log^3 n \log\log n)$. This is so because the parallel algorithm A_p in both cases is Valiant's sorting algorithm [Va] that runs in $O(\log n \log\log n)$ parallel time and uses $O(n)$ processors, and the sequential algorithm A_s runs in $O(n \log n)$ time in both cases (we use sorting in both cases, although a linear median finding algorithm [BFPRT] could be utilized in the prelinimary median root finding; the slope selection algorithm uses inversion counting algorithm that runs in $O(n \log n)$ in addition to sorting).

During the execution, the program counts how many times `add_comparison` is called (this is the row "Total number of tests" in the tables below), and how many times `perform_test` is called (this is the row "Actual tests made"). The ratio between these numbers roughly represents the saving in running time gained by maintaining the parameter lower and upper bounds.

As can be seen in the tables, the saving in running time is huge (a factor of more than 30 in the tables) in the parallel versions. The improvement is most notable however in the serial versions. Practically, the short running times of the serial versions, which are very easy to code, suggests that it may be wise in practice to run these instead of the complicated parallel versions or the costly enumeration methods.

Number of lines	1600	3200	6400	12800	25600
Total number of tests	493.5	590.0	685.6	788.7	908.1
Actual tests made	15.2	16.4	17.0	18.2	18.6

Table 3.1: Median root; parallel version; uniform distribution

Number of lines	1600	3200	6400	12800	25600
Total number of tests	37,523	82,556	178,652	392,238	839,799
Actual tests made	25.9	29.1	31.5	32.7	35.5

Table 3.2: Median root; serial version; uniform distribution

Number of lines	1600	3200	6400	12800	25600
Total number of tests	502.0	589.7	694.9	806.3	909.9
Actual tests made	21.9	24.1	28.1	28	31.8

Table 3.3: Slope selection; parallel version; exponential distribution

3.4 Conclusions

The results presented in this section indicate that Megiddo's parametric search technique [Me] is practically feasible both in terms of the complexity of coding and in terms of actual running times.

Number of lines	1600	3200	6400	12800	25600
Total number of tests	500.8	585.8	690.6	789.9	907.3
Actual tests made	22.3	23.4	28.4	27.5	30

Table 3.4: Slope selection; parallel version; uniform distribution

Number of lines	1600	3200	6400	12800	25600
Total number of tests	27,062	56,062	151,385	307,566	637,312
Actual tests made	39.6	34.7	46.7	52.2	57.8

Table 3.5: Slope selection; serial version; uniform distribution

The use of a simple heuristic (using the bounds of the interval where the sought parameter is known to lie, to avoid unnecessary costly comparisons) is extremely effective. This conclusion is valid however only under the input distributions tested. Moreover, the effectiveness of this heuristic is also demonstrated for parametric search algorithms where the generic algorithms are serial.

The latter are very simple to code. They seem to be efficient in practice (although this may be the effect of the input distribution), and are certainly better than the brute-force enumeration methods (i.e. generating the $\binom{n}{2}$ intersection points and ranking them in the slope selection problem) that have similar running times.

There are two remarks that must be added. One is that the comparisons in our programs depend upon linear functions. This simplifies the coding considerably. Dealing with roots of higher degree polynomials is possible, but is non-trivial and time-consuming.

The second point is the model of numerical computation, discussed also in the Introduction. Our programs use hardware floating point operations, thus real numbers are not represented exactly. Programs where reals are represented exactly may run considerably slower. Another unfortunate result of this fact is that our experimental data is not extremely reliable. Inputs with about 25000 lines are almost sure to contain numerically unstable situations (e.g. nearly parallel lines) that may lead to erroneous results.

Bibliography

[AASS] P. K. Agarwal, B. Aronov, M. Sharir and S. Suri, Selecting distances in the plane, *Proc. 6th ACM Symp. on Computational Geometry*, 1990, 321–331.

[AB] F. Avnaim and J. D. Boissonnat, The polygon containment problem: 1. Simultaneous containment under translation, Technical report 689, INRIA Sophia Antipolis, June 1987.

[AB1] F. Avnaim and J. D. Boissonnat, Polygon placement under translation and rotation, *5th Annual Symp. on Theoretical Aspects of Computer Science*, Lectures Notes in Comp. Science 294, Springer-Verlag, New-York, 1988, 322–333.

[ACGOY] A. Aggarwal, B. Chazelle, L. Guibas, C. Ó'Dúnlaing and C. Yap, Parallel computational geometry, *Algorithmica*, **3** (1988), 293–327.

[AKS] M. Ajtai, J. Komlós and E. Szemerédi, An $O(n \log n)$ sorting network, *Combinatorica*, **3** (1983), 1–19.

[AS] P. K. Agarwal and M. Sharir, Planar geometric location problems, DIMACS Technical Report 90-58, August 1990.

[ASS] P. Agarwal, M. Sharir and P. Shor, Sharp upper and lower bounds on the length of general Davenport-Schinzel sequencees, *J. Combin. Theory Ser. A*, **52** (1989), 228–274.

[BFPRT] M. Blum, R.W. Floyd, V.R. Pratt, R.L. Rivest and R.E. Tarjan, Time bounds for selection, *J. Comput. System Sci.*, **7** (1972), 448–461.

[Ch] B. Chazelle, The polygon containment problem, in *Advances in Computing Research, Vol I: Computational Geometry*, (F.P. Preparata, ed.), JAI Press, Greenwich, Connecticut (1983), 1–33.

[Ch1] B. Chazelle, Reporting and counting segment intersections, *J. Comp. Sys. Sci.*, **32** (1986), 156–182.

[CK] L. P. Chew and K. Kedem, Placing the largest similar copy of a convex polygon among polygonal obstacles, *Proc. Fifth ACM Symposium on Computational Geometry*, 1989, 167–173.

[CK1] L. P. Chew and K. Kedem, High-clearance motion planning for a convex polygon among polygonal obstacles, Technical report 90-1133, Dept. of Computer Science, Cornell University, June 1990.

[Co] R. Cole, Parallel merge sort, *27th IEEE Symp. on Foundations of Computer Science*, 1986, 511–516.

[Co1] R. Cole, Slowing down sorting networks to obtain faster sorting algorithms, J. Assoc. Comput. Mach., **34** (1987), 200–208.

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

[CY] R. Cole and C. Yap, A parallel median algorithm, *Inf. Process. Lett.*, **20** (1985), 137–139.

[Ed] H. Edelsbrunner, *Algorithms in Combinatorial Geometry*, Springer-Verlag, Berlin (1987).

[EGS] H. Edelsbrunner, L.J. Guibas and J. Stolfi, Optimal point location in a monotone subdivision, *SIAM J. Comput.*, **15** (1986), 317–340.

[Fe] W. Feller, *An Introduction to Probability Theory and Its Applications*, Vol II, 2nd edition, Wiley, New York (1971).

[Fo] S. Fortune, Fast algorithms for polygon containment, *Proc. 12th International Colloquium on Automata, Languages and Programming*, Lecture Notes in Comp. Science 194, Springer-Verlag, New-York, 1985, 189–198.

[FJ1] G. N. Frederickson and D. B. Johnson, Finding k-th paths and p-centers by generating and searching good data structures, *J. Algorithms*, **4** (1983), 61–80.

[FJ2] G. N. Frederickson and D. B. Johnson, Generalized selection and ranking: sorted matrices, *SIAM J. on Computing*, **13** (1984), 14-30.

[Fr] G. N. Frederickson, Optimal algorithms for partitioning trees and locating p-centers in trees, manuscript, 1990.

[GRS] L. Guibas, L. Ramshaw and J. Stolfi, A kinetic framework for computational geometry, *24th IEEE Symp. on Foundations of Computer Science*, 1983, 100-111.

[HS] S. Hart and M. Sharir, Nonlinearity of Davenport-Schinzel sequences and of generalized path compression schemes, *Combinatorica*, **6** (1986), 151–177.

[KS] K. Kedem and M. Sharir, An efficient motion-planning algorithm for a convex polygonal object in two-dimensional polygonal space, *Discrete and Computational Geometry* **5** (1990), 43–75.

[LS] D. Leven and M. Sharir, On the number of critical free contacts of a convex polygonal object in two-dimensional polygonal space, *Discrete and Computational Geometry* **2** (1987), 255–270.

[LS1] D. Leven and M. Sharir, Planning a purely translational motion for a convex object in two-dimensional space using generalized Voronoi diagrams, *Discrete and Computational Geometry* **2** (1987), 9–31.

[Ma] J. Matoušek, Randomized optimal algorithm for slope selection, manuscript, 1991.

[Me] N. Megiddo, Applying parallel computation in the design of serial algorithms, *J. ACM* **30** (1983), 852–856.

[Me1] N. Megiddo, Linear-time algorithms for linear programming in R^3 and related problems, *SIAM J. Comput.* **12** (1983), 759–776.

[Me2] N. Megiddo, Linear programming in linear time when the dimension is fixed, *J. Assoc. Comput. Mach.*, **31** (1984), 114–127.

[Me3] N. Megiddo, The weighted Euclidean 1-center problem, *Math. of Operations Reseach*, **8** (1983), 498–504.

[PS] Y. Perl and S. T. Schach, Max-min tree partitioning, *J. Assoc. Comput. Mach.*, **28** (1981), 5–15.

[PY] M.S. Paterson and F.F. Yao, Point retrieval for polygons, *J. Algorithms*, **7** (1986), 441–447.

[Re] A. Reiser, A linear selection algorithm for sets of elements with weights, *Inf. Process. Lett. 7*, (1978), 159–162.

[SS] J. T. Schwartz and M. Sharir, On the Piano Movers Problem: II. General techniques for computing topological properties of real algebraic manifolds, *Advances in Applied Mathematics* **4** (1983), 298–351.

[To] S. Toledo, Extremal polygon containment problems, *Proc. 7th ACM Symp. on Computational Geometry*, 1991, to appear.

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