Dynamic Three-Dimensional Linear Programming

David Eppstein

Department of Information and Computer Science
University of California, Irvine, CA 92717

Tech. Report 91-53

June 3, 1991

Abstract

We perform linear programming optimizations on the intersection of k polyhedra in $\mathbf{R}^3$, represented by their outer recursive decompositions, in expected time $O(k \log k \log n + \sqrt{k} \log k \log^3 n)$. We use this result to derive efficient algorithms for dynamic linear programming problems in which constraints are inserted and deleted, and queries must optimize specified objective functions. As an application, we describe an improved solution to the planar 2-center and 2-median problems.

1 Introduction

Linear programming is a fundamental problem in computer science. Linear programs can be thought of as polytopes in Euclidean space formed by intersecting halfspaces (constraints); the problem is to maximize a linear objective function over the points in the polytope. We introduce the study of *dynamic* linear programs, in which we update the polytope by inserting and deleting constraints, and answer queries in which we optimize a specified objective function.

We consider problems with bounded dimension. The main result in this area is that bounded dimension linear programs can be solved in linear time [8, 13, 14, 17]. Two dimensional dynamic problems can be solved by geometric duality, using a generalization of the convex hull of a planar point set. Offline updates, and updates consisting of insertions only, can be performed in $O(\log n)$ time [11]; arbitrary updates can be be performed in time $O(\log^2 n)$ [16]. These algorithms answer queries in time $O(\log n)$.

Here we consider three dimensional dynamic linear programs. We can also solve certain related non-linear problems, including ray-shooting in the convex hull of a dynamic point set, and maintaining the smallest circle containing a planar point set. If the constraint set is static, queries can be solved in time $O(\log n)$ using a data structure of size $O(n)$ [5, 9, 12]; by expanding the space to $O(n \log\log n)$ we can insert or delete constraints in linear time. For arbitrary updates, we generalize these bounds to trade off between update and insertion times. For insertions only, or offline sequences of insertions and deletions, we achieve polylogarithmic update and query time. Our expected time bounds depend only on the random bits used by the algorithms, and not on the set of constraints.

1.1 New Results

We describe algorithms that solve the following problems.

- Given three polyhedra with at most n faces each, represented by a linear size data structure, optimize a given objective function over the intersection of the polyhedra, in time $O(\log^3 n)$.
- Given k polyhedra represented as above, optimize a given objective function over their intersection, in expected time $O(k \log k \log n + \sqrt{k} \log k \log^3 n)$ and worst case time $O(k^3 \log^3 n)$.

- Given k polyhedra as above, detect whether they have a common intersection in the same time as above. The best previous algorithms for this problem took time $O(k^4 \log^4 n)$ [4].

- For any $\epsilon > 0$, perform insertions and deletions in a dynamic linear program in time $O(n^{1-\epsilon})$ per update, and queries in expected time $O(n^\epsilon \log^2 n)$ or worst case time $O(n^{3\epsilon} \log^3 n)$. We can balance these bounds to achieve $O(\sqrt{n} \log n)$ expected time per operation, or $O((n \log n)^{3/4})$ worst case time per operation.

- Perform a *semi-online* sequence of updates in time $O(\log n \log\log n)$ per update, and expected $O(\log^{7/2} n \log\log n)$ or worst case $O(\log^6 n)$ time per query. In such a problem, each insertion specifies the position in the update sequence of the corresponding deletion [6].

- Perform a sequence of insertions in $O(\log n)$ time per update, and expected $O(\log^{7/2} n \log\log n)$ or worst case $O(\log^6 n)$ time per query.

- For any fixed objective function, perform a semi-online sequence of updates, in expected $O(\log^2 n \log^2 \log n)$ or worst case $O(\log^4 n)$ time each. This algorithm can maintain the minimum circle containing a planar point set; using this we solve the planar 2-center [1] and 2-median [7] in expected time $O(n^2 \log^2 n \log^2 \log n)$. This improves the best previous 2-median algorithm by an almost linear factor, and is also an improvement on the best previous 2-center algorithm.

2 Preliminaries

Our algorithms are based on the *recursive decomposition* of a polyhedron, introduced by Dobkin and Kirkpatrick [5, 12].

The skeleton of a convex polyhedron forms a planar graph, and has an independent set of $\Omega(n)$ vertices of degree at most six. The *inner recursive decomposition* is formed by removing these vertices, taking the convex hull of the remaining points, and continuing recursively. The decomposition has $O(\log n)$ levels, each of which is the polyhedron formed by the convex hull of certain points, and each of which contains the polyhedron at the previous level. The entire decomposition has linear size, and can be computed from the original polyhedron in linear time. Dually, the *outer recursive decomposition* is formed by recursively removing a large independent set of faces, each of which has at most six sides. The decomposition has $O(\log n)$ levels,

each of which is a polyhedron formed by cutting off corners of the previous polyhedron along the faces in the independent set. The following result is well known:

Theorem 1. *Given the outer decomposition of an n-sided polyhedron, the optimum of a given linear objective function can be found in time $O(\log n)$.*

Proof: We start with the first polyhedron in the outer decomposition, and proceed in stages corresponding to the level of the decomposition. At each stage we maintain the point maximizing the function in the polyhedron at that level of the decomposition. Whenever the point is cut off by a face at the next level, the new optimum must be a vertex of the face, and can be found in constant time. □

A dual version of theorem 1 was introduced by Kirkpatrick [12] as an algorithm for performing point location in planar subdivisions. We need a stronger result from [5].

Lemma 1. *Given the outer decomposition of a polyhedron, the intersection of the polyhedron with a given line can be found in time $O(\log n)$.*

Proof: Dually, we are given a line not intersecting the polyhedron, and we must find planes containing the line and tangent to the polyhedron. The outer decomposition dualizes to an inner decomposition. We project the problem onto a plane perpendicular to the line; we now are looking for tangent lines to the polygonal shadow of the polyhedron. The decomposition of the polyhedron carries over to a decomposition of its shadow, and the tangents can be found in stages as in theorem 1. □

3 Three Polyhedra

Our algorithm for optimizing in the intersection of several polyhedra uses as a base case the intersection of three polyhedra. We proceed in stages. At each stage, we will maintain the optimum point on the intersection of three polyhedra, which are each at some level of the outer decomposition of the original three polyhedra.

Initially all three polyhedra have constant size, and we can find the optimum in constant time. At each stage we choose one of the polyhedra, and cut off some of its corners to form the polyhedron at the next level of the

decomposition. If no corner cuts off the previous optimum point, it remains optimal and we proceed to the next stage. Otherwise, the new optimum will be on the face that cut off the old one. We find that point by solving a problem in the plane of the face, using the induced decomposition of the polygons formed as intersections of the polyhedra with the plane. At each stage in the planar problem, the optimum point may be cut off by a line segment, and the new optimum is found by lemma 1.

In both the original problem and the planar subproblems, we must repeatedly test whether the current optimum point has been cut off. This is done by passing a line through the optimum point, and through another point known to be interior to the polyhedron. By lemma 1, we can find the intersection of this line with the polyhedron resulting from cutting off the new set of faces. If the previous optimum is not in this intersection, it has been cut off. The face pierced by the line is the one that cut it off.

Lemma 2. *Given the outer decompositions of three polyhedra, and given a linear objective function, we can find the maximum of the function in the intersection of the polyhedra, in time* $O(\log^3 n)$.

Proof: Each line segment problem takes time $O(\log n)$, by lemma 1. Each planar problem takes time $O(\log n)$ per stage to test whether the optimum has been cut off, and solves one line segment problem per stage. The original polyhedron problem takes time $O(\log n)$ per stage to test whether the optimum has been cut off, and solves one planar problem per stage. □

For two polyhedra, the time can be reduced to $O(\log^2 n)$ because, in the planar subproblems, one polygon will have $O(1)$ sides. Therefore we can find the optimum in the other polygon in $O(\log n)$ time, then combine that with the $O(1)$-size polygon in $O(1)$ stages, each of which takes $O(\log n)$ time.

For k polyhedra, each subproblem involves $O(k \log n)$ stages, and the total time is $O(k^3 \log^3 n)$. In the next section we reduce this time, using a randomized dynamic programming technique.

We also solve ray-shooting queries in the convex hull of a point set. Dually this is a nonlinear program in which the level sets of the objective functions are planes containing a common line. At the lower levels of our algorithm, this modified objective function may not have a feasible solution; geometrically, the pivot line may pass through the intersection of the polyhedra. We modify our algorithm so that, as long as the line passes through the intersection of the polyhedra, we simply test whether we have cut off the region containing the line, using lemma 1. Once the line is cut off, the

optimum point is found on the cutting face, and we proceed as before. The analogous modification must also be performed in the planar subproblems.

4 Several Polyhedra

We now consider optimizing in the intersection of k polyhedra. The optimum point is the intersection of at least three constraint planes, and so can be found as the result of some three-polyhedron optimization. For any polygon P, either the remaining polygons contain a triple that determines the optimum for the entire set, or P is necessarily a member of such a triple. We test P by removing it from the set and computing the optimum point in the remaining set, then checking whether that point is in P. If so, this gives the optimum we are seeking. If not, we continue until three polyhedra prove themselves necessary; then the optimum point can be found in the intersection of just those three polyhedra.

Let the polyhedra be numbered P_1, P_2, ..., P_k, and let $S(x, y, z)$ denote the optimum on the intersection of P_1, P_2, ..., P_z, P_{y+z}, P_{x+y+z}. Let $I(x, y, z)$ denote the optimum in the intersection of P_z, P_{y+z}, and P_{x+y+z}. Then $S(1, 1, k-2)$ is the result we are seeking, and is computed by the following dynamic program.

if $z = 1$ **then** $S(x, y, z) = I(x, y, z)$
else if $y = 1$ **then begin**
    if $S(y, 1, z-1) \in P_{x+y+z}$ **then** $S(x, y, z) = S(y, 1, z-1)$
    else if $S(x+y, 1, z-1) \in P_{y+z}$ **then** $S(x, y, z) = S(x+1, 1, z-1)$
    else if $S(x, y+1, z-1) \in P_z$ **then** $S(x, y, z) = S(x, y+1, z-1)$
    else $S(x, y, z) = I(x, y, z)$
end else begin
    if $S(x, y+1, z-1) \in P_z$ **then** $S(x, y, z) = S(x, y+1, z-1)$
    else if $S(x+y, 1, z-1) \in P_{y+z}$ **then** $S(x, y, z) = S(x+1, 1, z-1)$
    else if $S(y, 1, z-1) \in P_{x+y+z}$ **then** $S(x, y, z) = S(y, 1, z-1)$
    else $S(x, y, z) = I(x, y, z)$
end

The cases for $y = 1$ and $y > 1$ differ only in the order of the tests; this affects the algorithm's running time but not its correctness.

If the constraints have a solution, the algorithm will correctly compute it. Otherwise we will at some point compute a result that does not satisfy all constraints. Our probabilistic analysis depends on feasibility, so we

check that each computed optimum $I(x, y, z)$ is feasible for $S(x, y, z)$, in time $O(z \log n)$. Whenever we detect infeasibility, we halt the entire computation. With this check, our time bounds will hold for both feasible and infeasible problems; in particular our algorithms can be used to check whether a set of polyhedra has a non-empty intersection. This increases our worst case time to $O(k^3 \log^3 n + k^4 \log n)$; for a worst-case $O(k^3 \log^3 n)$ algorithm that checks feasibility, one could use the algorithm of lemma 2 extended to k polyhedra instead of three.

We select the initial numbering of the polyhedra uniformly at random. Let us examine how this can be used to determine the probability of various events happening in the computation of cell $S(x, y, z)$.

First assume the corresponding set of polyhedra has a common intersection (i.e. a feasible solution). The first **if** statement will fail exactly when P_{x+y+z} does not supply one of the three faces forming the optimum vertex; there are $z + 2$ possible faces so this happens with probability at most $3/(z + 2) = O(1/z)$ (less if one polyhedron supplies more than one face or the vertex is formed by more than three faces). Similarly the second test fails with probability $O(1/z^2)$ and the third test fails with probability $O(1/z^3)$. This assumes we know nothing more about the cell; if z, $y + z$, or $x + y + z$ has been chosen as a function of the algorithm's behavior the probabilities increase appropriately. We use these probabilities in two ways. First, only a small number of the $O(k^3)$ possible cells are actually used in computing $S(1, 1, k - 2)$. Second, an even smaller number of those cells involve computation of $I(x, y, z)$.

Alternately, assume the computation of $S(x, y, z)$ involves an infeasibility. Then by Helly's theorem, some set of four polyhedra have no common intersection. The computation will get past the first **if** statement only if no infeasibility is detected. This happens whenever polyhedron x must necessarily be a member of such a quadruple, with probability at most $4/(z+2) = O(1/z)$. Similarly the second test fails with probability $O(1/z^2)$ and the third fails with probability $O(1/z^3)$ as in the feasible case. So from now on we can ignore infeasibility, except for the cost of testing each new optimum.

We divide the cells of S into categories. The *spine* consists of cells $S(1, 1, z)$. The *branches* consist of cells $S(x, 1, z)$ with $x > 1$; branch b consists of cells with $x + y + z = b$. The *stems* consist of the remaining cells, $S(x, y, z)$ with $y > 1$. Stem (b, c) consists of cells $S(x, y, z)$ with $y + z = c$ and $x + y + z = b$.

Each spine cell can potentially use the value of a previous spine cell, a

branch cell, and a stem cell. Each cell on branch b can use the value of a spine cell, a previous cell on the same branch, and a stem cell on some stem (b, c). Each cell on stem (b, c) can use the value of a previous cell on the same stem, and branch cells on branches b and c. All spine cells are used in the computation, so we need not worry whether they are also used by some branch cells. Except for the possibility of stem (b, c) using cells in branch c, the remaining dependencies form a tree-like structure.

Lemma 3. *In each chosen stem (b, c), the expected number of cells in which a branch cell is used or a computation of $I(x, y, z)$ is made is $O(\log c)$. The total expected cost for verifying feasibility is $O(c \log n)$.*

Proof: The choice of b and c determine $x + y + z$ and $y + z$, but z is left undetermined, so each cell $S(x, y, z)$ has probability $O(1/z)$ of failing the first **if** statement. The expected number of cells in which this happens is $\sum_{z=1}^{c} O(1/z) = O(\log c)$. If a computation of $I(x, y, z)$ is made, the cost of verifying feasibility is $O(z \log n)$. The expected cost per cell is $O(\log n)$, so the total expected cost is $O(c \log n)$. □

Lemma 4. *In all branch cells, and all stem cells reachable from branch cells, the total expected number of other cells used is $O(k \log k)$, the total expected number of optimizations performed is $O(k)$, and the total expected cost of verifying feasibility is $O(k \log k \log n)$.*

Proof: Assume a fixed value of x; we will add the expectations over all values of x so this does not bias the selection of cells. The probability of cell $(x, 1, z)$ failing the first and second tests, and therefore using a stem cell, is $O(1/z^2)$. If it does use a stem cell, it will use all z cells in the stem, will use $O(\log z)$ other branch cells, will perform $O(\log z)$ optimizations, and will cost $O(z \log n)$ for feasibility tests both in the stem and at the branch cell itself. So the expected number of stem cells used is $O(1/z)$, the expected number of branch cells and optimizations is $O(\log z/z^2)$, and the expected cost of feasibility testing is $O(\log n/z)$. The probability that $(x, 1, z)$ uses another branch cell directly, and therefore the expected number of such cells used, is also $O(1/z)$. Summing over all z gives $O(\log x)$ stem and branch cells used, $O(\log x \log n)$ total cost for feasibility testing, and $O(1)$ optimizations performed. Summing over the k possible values of x gives the results claimed. □

We have proved that the expected total number of cells visited in the algorithm is $O(k \log k)$, and the expected time for testing feasibility is $O(\log n)$ per cell. To finish the analysis, we bound the cells in which an optimization occurs. By the above analysis this number is $O(k)$ but we can prove much tighter bounds.

Lemma 5. *The expected number of optimizations is* $O(\sqrt{k} \log k)$.

Proof: The spine contains $O(1)$ expected optimizations. We divide the branches and stems into two cases, as b is less than or greater than $\sqrt{k} \log k$. In the first case, lemma 4 tells us that the total number of optimizations is $O(\sqrt{k} \log k)$.

In the second case, we bound the number of branches in which some cell is used. If a branch is chosen, the probability of each cell $S(x, y, z)$ using the corresponding stem is $O(1/z)$. If it does use the stem, the expected number of optimizations is $O(\log z)$, so the total expected number per branch cell is $O(\log z/z)$. Summing over the branch gives an expected $O(\log^2 b)$ optimizations per selected branch.

Branch b is selected if and only if some other branch cell $(x, 1, b-1)$ fails its first two tests, and uses a stem cell. There are $O(k)$ possibilities for x, so the total probability of selection is $O(k/b^2)$. Therefore the expected number of selected branches with $b > \sqrt{k} \log k$ is

$$\sum_{b=\sqrt{k}\log k}^{k} O\left(\frac{k}{b^2}\right) = O\left(\frac{k}{\sqrt{k}\log k}\right) = O\left(\frac{\sqrt{k}}{\log k}\right).$$

Recall that each chosen branch has an expected $O(\log^2 b) = O(\log^2 k)$ optimizations; therefore the total number of optimizations with $b > \sqrt{k} \log k$ is $O(\sqrt{k} \log k)$. □

Theorem 2. *We can compute the optimum of a linear function over the intersection of* k *polyhedra in time* $O(k \log k \log n + \sqrt{k} \log k \log^3 n)$.

Proof: $O(k \log k)$ cells are used, and $O(\sqrt{k} \log k)$ of those cells compute $I(x, y, z)$. Each use of a cell takes time $O(\log n)$ to test points for polygon containment. Each computation of $I(x, y, z)$ takes time $O(\log^3 n)$. □

5 Dynamic Data Structures

The previous section shows how solutions to a static linear programming query problem may be combined into the solution for the union of the constraints (intersection of the polytopes), with polylogarithmic overhead. This is reminiscent of *decomposable searching problems* [2, 6, 15]. In such a problem, we again ask query problems in some set; a problem is decomposable if the answers to queries in disjoint sets can be combined in constant time to give the answer in the union of the sets. If so, k sets can be combined in time $O(k)$. In our case, we cannot simply combine the answers, but we provide an algorithm for solving the combined problem efficiently. This suffices for deriving dynamic linear programming algorithms.

First, we consider the fully dynamic case, in which any constraint may be inserted or deleted. Such a problem can be solved by rebuilding the recursive decomposition after each update, with logarithmic time per query and linear time per update. We now show how to balance these bounds.

Theorem 3. *For any fixed $\epsilon > 0$, we can perform insertions and deletions in a dynamic linear program in $O(n^{1-\epsilon})$ time per update, and expected $O(n^{\epsilon} \log^2 n)$ or worst case $O(n^{3\epsilon} \log^3 n)$ time per query.*

Proof: We maintain the recursive decompositions of $O(n^{\epsilon})$ polyhedra, each of size $O(n^{1-\epsilon})$. Each update involves changing $O(1)$ polyhedra, one in which the update occurs and some others to keep the sizes of the polyhedra balanced. We maintain the polyhedra in linear time per update using a data structure with size $O(n \log \log n)$, based on Chazelle's polyhedron intersection algorithm [3]. Each query takes expected time $O(n^{\epsilon} \log^2 n + n^{\epsilon/2} \log^4 n)$, and worst case time $O(n^{3\epsilon} \log^3 n)$, by theorem 2. □

In semi-online algorithms [6] we are told, when we insert a constraint, the time (position in the update sequence) at which it will be deleted. As a special case, we may be given in advance the sequence of updates, without knowing the coordinates of each constraint until it is inserted. This includes the offline problem, in which we also know in advance the coordinates of the constraints. It also includes the dynamic problem in which only insertions are allowed; the sequence of events is then simply a list of insertions.

Theorem 4. *We can perform a semi-online sequence of insertions and deletions in a dynamic linear program, in time $O(\log n \log \log n)$ per update, and expected $O(\log^{7/2} n \log \log n)$ or worst case $O(\log^6 n)$ time per query.*

Proof: We first divide the sequence of events into pieces, each of which corresponds to $O(n)$ insertion and deletion events. We can process each piece as a whole without worrying about the number of events becoming much larger than the number of constraints.

We use a *segment tree* for the lifetimes of the constraints [6]. This is a balanced binary tree, having as leaves the insertions and deletions of constraints. Each constraint is listed in $O(\log n)$ internal nodes, which together cover exactly all the leaves for which the constraint is active. At each time, the set of active constraints is found as the union of the sets at the nodes that are ancestors of the leaf representing the present time.

Each constraint is involved in $O(\log n)$ polyhedra. Each polyhedron must be formed from the intersection of the corresponding constraints, and then recursively decomposed. Intersecting the constraints at each level would take $O(n \log n)$ time per polyhedron, but with appropriate merging of smaller polyhedra this time can be reduced to $O(n \log \log n)$. Therefore the total time spent performing updates is $O(n \log n \log \log n)$. This amortized bound can be made worst case using techniques of Overmars and van Leeuwen [15].

The query times follow from theorem 2 with $k = O(\log n)$. □

For insertions only, the polyhedron at each node of the segment tree is simply the intersection of two smaller polyhedra, which can be found in linear time [3]. So the time per insertion becomes $O(\log n)$; again, this can be made worst case. The semi-online and insertion-only algorithms can perform queries more quickly, at the expense of slower updates, by increasing the branching factor of the segment tree.

We can also make our algorithms persistent; this means that each update creates a new version of the data structure, and queries specify which version is to be queried. For the semi-online algorithm this extension is straightforward. The fully dynamic algorithm can also be made persistent, using the persistent semi-online algorithm as a subroutine [10].

6 Fixed Objective Function

Consider a semi-online problem in which there is only one objective function of interest. We may assume that, after each update, we perform an optimization with this objective function, so that all operations are updates.

We solve such a problem by mixing the algorithms of lemma 2 and theorem 2. More precisely, we use techniques from the lemma to reduce the whole problem to a sequence of planar subproblems; then we solve each

subproblem using a randomized dynamic programming technique similar to that of the theorem. Within the dynamic program we solve two-polygon subproblems using the $O(\log^2 n)$ algorithm used as a subroutine in lemma 2.

Lemma 6. *We can compute the optimum of a linear function over the intersection of k polygons in the plane, represented by their recursive decompositions, in expected time $O(k \log n + \log^2 k \log^2 n)$.*

Proof: Let the polygons be numbered P_1, P_2, ..., P_k, and let $S(x, y)$ denote the optimum on the intersection of P_1, P_2, ..., P_y, P_{x+y}. Let $I(x, y)$ denote the optimum in the intersection of P_z and P_{x+y}. Then $S(1, k-1)$ is the result we are seeking, computed by the following dynamic program.

if $y = 1$ **then** $S(x, y) = I(x, y)$
else if $S(1, y-1) \in P_{x+y}$ **then** $S(x, y) = S(1, y-1)$
else if $S(x+1, y-1) \in P_y$ **then** $S(x, y) = S(x+1, y-1)$
else $S(x, y) = I(x, y)$

The program's correctness follows from the fact that the optimum of the system of polygons is also the optimum of some pair of polygons. Each cell takes time $O(\log n)$ to perform polygon containment tests. Each computation of $I(x, y)$ takes $O(\log^2 n)$ time to perform and $O(y \log n)$ time to test for feasibility. As in the three-dimensional program, if we order the polygons randomly then each containment test will fail with probability $O(1/y)$, either when the polygon being tested is a necessary part of a pair of polygons determining the optimum, or when it is a necessary part of a triple with empty common intersection.

The *spine* consists of cells with $x = 1$, and the *branches* consist of the remaining cells. If a cell in branch $x + y$ is used, all such cells may be used; the time will be $O((x + y) \log n)$ for containment tests, $\sum O(\log^2 n/y) = O(\log(x+y) \log^2 n)$ for optimizations, and $\sum O(y \log n/y) = O((x+y) \log n)$ for feasibility tests. All cells used from the branch will be spine cells or cells in the same branch; this tree-like structure makes the analysis much simpler than that for theorem 2. The probability of spine cell $(1, y)$ using branch $1+y$ is $O(1/y)$, so the total time is $\sum O((\log y \log^2 n + y \log n)/y) = O(k \log n + \log^2 k \log^2 n)$. □

Theorem 5. *Given a fixed objective function, we can perform a semi-online sequence of updates in amortized time $O(\log^2 n \log^2 \log n)$ per update.*

Proof: We again use a segment tree. For each node in the tree, we compute the optimum in the intersection of the node and its ancestors, using a mixture of the algorithms of lemma 2 and theorem 2. We start with the reduction from the general problem to a collection of planar subproblems. Because we know the optimum at the node's parent, this only involves a number of subproblems logarithmic in the size of the node. The size of the polyhedron at a node of the segment tree is proportional to the number of its descendents in the tree; the logarithm of this quantity is simply the number of levels between the node and the leaves. Summing over all nodes, a sequence of n updates involves $O(n)$ planar subproblems. We solve each subproblem using lemma 6. In our case, $k = \log n$, and the time is $O(\log^2 n \log^2 \log n)$. □

A special case is the maintenance of the minimum diameter circle containing a dynamic planar point set. If we project point (x, y) onto the paraboloid $(x, y, x^2 + y^2)$, circles containing all points project into planes cutting above the convex hull of the projected points. Dually, this becomes a problem of optimizing a particular non-linear objective function on the intersection of a set of half-spaces corresponding to the points.

Corollary 1. *We can maintain the smallest circle containing a planar point set, subject to a semi-online sequence of updates, in amortized expected time $O(\log^2 n \log^2 \log n)$ per update.* □

Proof: All polyhedra used in the algorithm of theorem 5 are formed as intersections of some subset of the halfplanes, and therefore correspond to sets of input points. For such polyhedra, the smallest circle objective function behaves like a linear function in the property we need, namely that if a point lies outside the smallest circle for some other points, then the smallest circle for the new set of points is tangent to that outside point. The main difference between this problem and a linear problem is that in some cases the optimum point can fall on an edge of a polyhedron, corresponding to the situation in which the smallest circle is the diameter circle of two points. This leads to some additional cases in the recursive decomposition algorithms for three-dimensional and planar optimization, but does not change the time complexity by more than a constant factor. □

This can be used to solve the planar 2-center problem. In this problem, we must find two circles of minimum radius that together cover a point set. The best previous algorithm solves this problem in time $O(n^2 \log^3 n)$ [1].

Similarly, we can solve the 2-median problem [7], in which the sum of the radii is minimized; the best previous algorithm took time $O(n^3)$. Indeed our techniques allow us to optimize any monotone combination of the radii of the two circles.

Corollary 2. *We can solve the planar 2-center and 2-median problems in expected time* $O(n^2 \log^2 n \log^2 \log n)$.

Proof: We use the following (well known) duality technique to transform the problem into one of searching through a planar subdivision.

Given any two circles covering a point set, either they do not intersect, in which case they can be separated by a line, or otherwise the points in them can be separated by the line connecting the two intersection points of the circles. In either case we can assume that the two circles cover two disjoint sets separated by a line. Two such lines are equivalent if they separate the same sets of points.

The space of lines in the plane can be transformed by geometric duality into a space of points; the original points are simultaneously transformed into lines. The equivalence classes of lines in the primal problem become cells in the arrangement formed by the lines in the dual problem (input points in the primal problem). Two adjacent cells differ by the crossing of a single dual line (primally, the movement of a single input point from one side to the other of a separating line).

The dual line arrangement has $O(n^2)$ cells, and can be computed in time $O(n^2)$. If we perform a depth first search of adjacent cells, we will visit every cell, producing a sequence of $O(n^2)$ dual line crossings (insertions of input points from one set and deletions from the other set). We can compute this sequence in advance, then use an offline algorithm for maintaining the minimum containing circle as above, to compute the minimum containing circle for the sets of points on each side of each possible separating line. The total time will be $O(n^2 \log^2 n \log^2 \log n)$.

Then we can combine the circle radii for each separating line by summing them in the 2-median problem, or taking their maximum in the 2-center problem, and compare all lines in time $O(n^2)$. This gives us the line corresponding to the optimal solution; the actual circles can then be found in linear time, or remembered from their earlier offline computation. □

7 Conclusions

We have described algorithms for optimizing linear and certain non-linear objective functions on intersections of polyhedra; we then used these algorithms to solve dynamic linear programs analogous to the solution of dynamic decomposible searching problems.

We know of a number of directions for improvement. We would of course like to improve our time bounds. Perhaps our dynamic programming algorithm can be made to visit or optimize fewer cells, or perhaps a different algorithm would work better. We would also like to close the gap between our randomized and worst case time bounds. Finally, we would like to say something about generalizations to higher dimensions, and to any fixed dimension. The dynamic programming approach will work in any dimension, but its analysis becomes progressively more difficult. Recursive decomposition depends on the planarity of polyhedral skeletons, so it does not generalize. Perhaps some more complicated data structure will take its place.

References

[1] P.K. Agarwal and M. Sharir. Planar geometric location problems and maintaining the width of a planar set. 2nd ACM-SIAM Symp. Discrete Algorithms (1991) 449–458.

[2] J.L. Bentley and J.B. Saxe. Decomposable searching problems I: static-to-dynamic transformation. *J. Algorithms* 1 (1980) 301–358.

[3] B. Chazelle. An optimal algorithm for intersecting three-dimensional convex polyhedra. 30th IEEE Symp. Found. Comput. Sci. (1989) 586–591.

[4] D. Dobkin, personal communication.

[5] D.P. Dobkin and D.G. Kirkpatrick. Fast detection of polyhedral intersection. 9th Int. Colloq. Automata, Languages, and Programming, Springer-Verlag LNCS 140 (1982) 154–165.

[6] D.P. Dobkin and S. Suri. Dynamically computing the maxima of decomposable functions, with applications. 30th IEEE Symp. Found. Comput. Sci. (1989) 488–493.

[7] Z. Drezner. The planar two-center and two-median problems. *Transportation Science* 18 (1984) 351–361.

[8] M.E. Dyer. Linear algorithms for two and three-variable linear programming. *SIAM J. Comput.* 13 (1984) 31–45.

[9] H. Edelsbrunner and H. Maurer. Finding extreme points in three dimensions and solving the post-office problem in the plane. *Inf. Proc. Lett.* 21 (1985) 39–47.

[10] D. Eppstein. Persistence, offline algorithms, and space compaction. Manuscript, 1991.

[11] J. Hershberger and S. Suri. Offline maintenance of planar configurations. 2nd ACM-SIAM Symp. Discrete Algorithms (1991) 32–41.

[12] D. Kirkpatrick. Optimal search in planar subdivisions. *SIAM J. Comput.* 12 (1983) 28–35.

[13] N. Megiddo. Linear-time algorithms for linear programming in $\mathbf{R}^3$ and related problems. *SIAM J. Comput.* 12 (1983) 759–776.

[14] N. Megiddo. Linear programming in linear time when the dimension is fixed. *J. ACM* 21 (1984) 114–127.

[15] M.H. Overmars and J. van Leeuwen. Dynamization of decomposable searching problems yielding good worst case bounds. 5th GI Fachtagung Theoretische Informatik, Springer-Verlag LNCS 104 (1981) 224–233.

[16] M.H. Overmars and J. van Leeuwen. Maintenance of configurations in the plane. *J. Comput. Syst. Sci.* 23 (1981) 166–204.

[17] R. Seidel. Linear programming and convex hulls made easy. 6th ACM Symp. Comput. Geom. (1990) 211–215.