MASSACHUSETTS INSTITUTE OF TECHNOLOGY
A. I. LABORATORY

Artificial Intelligence
Memo No. 286

March 1973

THE FINDSPACE PROBLEM*

Gerald J. Sussman

ABSTRACT

The FINDSPACE problem is that of establishing a volume in space where an object of specified dimensions will fit. The problem seems to have two subproblems: the hypothesis generation problem of finding a likely spot to try, and the verification problem of testing that spot for occupancy by other objects. This paper treats primarily the verification problem.

Work reported herein was conducted at the Artificial Intelligence Laboratory, a Massachusetts Institute of Technology research program supported in part by the Advanced Research Projects Agency of the Department of Defense, and monitored by the Office of Naval Research under Contract Number N00014-70-A-0362-0005.

Reproduction of this document, in whole or in part, is permitted for any purpose of the United States Government.

---

*This memo was first issued in August 1971 as A.I. Vision Flash 18.

The FINDSPACE Problem

VISION FLASH 18

by

Gerald Jay Sussman

Massachusetts Institute of Technology

Artificial Intelligence Laboratory

Vision Group

8/3/71

ABSTRACT

The FINDSPACE problem is that of establishing a volume in space where an object of specified dimensions will fit. The problem seems to have two subproblems: the hypothesis generation problem of finding a likely spot to try, and the verification problem of testing that spot for occupancy by other objects. This paper treats primarily the verification problem.

Work reported herein was conducted at the Artificial Intelligence Laboratory, a Massachusetts Institute of Technology research program supported by the Advanced Research Projects Agency of the Department of Defense, and was monitored by the Office of Naval Research under Contract Number N00014-70-A-0362-0002.

The FINDSPACE problem is a computationally hairy but well-defined subproblem of the robot problem. The genesis of the problem name is a function in Terry Winograd's block manipulation program. This function assumes a model of the world consisting of a table with bricks piled on it and takes arguments describing an object to be added to the scene by placing it on a specified surface (either the table or the top of an existing block). If there is room on the specified surface for the given block, FINDSPACE returns a specification of the available space. If there is no available space, the FINDSPACE function returns NIL. Associated with the FINDSPACE program must be routines to update the data base, i.e. the environmental model, when an object is moved in, added to, or removed from the scene. Indeed, since these routines will be called relatively rarely compared with accessing of the data base, it is reasonable that they be expensive and compute a complex representation of the object whose status is being modified if this makes FINDSPACE sufficiently easier. I became involved in the FINDSPACE problem because Terry's program has certain restrictions (which I will described later) which make it fairly useless for dealing with real world situations.

In the real world, a FINDSPACE solution, though it may be heuristic, must be conservative. That is, it must not make the mistake of finding space when there is none, though it may be occasionally wrong by failing to find space when there is some. This is the case because the robot hand may be damaged by attempting to push one object through another. This dictum can best be understood by realizing that the stubbing of a toe reflects a failure of the human FINDSPACE.

The FINDSPACE problem has several subproblems to be solved. How do we represent the objects in the world model so that it is possible to solve the problem with a reasonable degree of efficiency? How can we propose a likely

space for an object? And given a proposed likely space, how can we verify its safety to the desired degree of conservativeness without rejecting too many good answers? Thus the FINDSPACE program may be divided into two subroutines: a proposer and a verifier. Although this paper is basically about verifiers, I will take a short excursion into the realm of proposers. The simplest proposer (which is employed in Winograd's program) is a random solution generator. It generates solutions randomly, constrained only by the input data (the object and the surface to put it on) without regard to the model. It then passes this proposed solution to the verifier. Such a method works very well in a sparse space. A more sophisticated proposer could have some description of the larger areas of free space, roughly analogous to a free storage list in a storage allocation system for a computer. The problem with this idea is that I haven't seen any scheme for doing this that doesn't become very fragmented quickly. Another possible scheme (currently implemented by Eugene Freuder) is a division of space into a set of buckets which are marked when they are occupied; thus only proposals of unmarked buckets may be made.

Getting to the verifier, various restrictions may be imposed on the problem to make it more tractable. In Winograd's program, for example, the only objects allowed are rectangular prisms, hereafter called bricks (he does allow pyramids, but from FINDSPACE's point of view they are bricks containing the pyramids), with all dimensions parallel to the coordinate axes; furthermore, no brick may ever be rotated. This model is too restricted for a real world robot since it cannot expect to find its world all lined up for it. In this model an object is represented as a single point (the lower, forward, lefthand corner) and a set of dimensions (length, width, and depth). The central function in Winograd's verifier is a function called GROW, which

takes a proposed point and grows it along the direction of the coordinate axes to become a rectangle of maximal size which intersects no other object. It then tests the size of the result to see if it fits the given object, and if it does it returns the center of the grown region.

I have written a verifier, called RCLEAR(recursive clear), which allows the model to contain bricks in any orientation, thus relaxing the most important restriction of the previous program. In the model each brick is represented as a list (P_0 E1 E2 E3), where P_0 is the centroid of the brick and E1, E2, E3 represent the edges of the brick. Ei is a list (d v), where d is the square of the length of the corresponding edge and v is the unit vector pointing in that direction. The general philosophy of RCLEAR is: in order to determine if the given object could fit in the given place in the scene, imagine it in place and check if it intersects any other object. The critical function here is one called DISJOINT1, which takes two representations of bricks and returns T if they are disjoint and NIL otherwise. Upon entry, DISJOINT1 computes the radii r_1 and r_2 of the smallest spheres completely enclosing each brick and the distance d between the centroids of the bricks (they are the centers of the spheres). If $d>r_1+r_2$ the bricks are disjoint. If not, it constructs r_1' and r_2', the radii of the largest spheres about the centroids completely enclosed by the bricks (they are half of the length of the minimum side). If $d<r_1'+r_2'$ the bricks overlap and the algorithm returns NIL. If neither test is passed, there is uncertainty in the result, and to remove this uncertainty we must apply a finer filter. We find the largest edge, say E1 of brick B1, of all six edges of the two bricks. We then divide B1 into two bricks, B11 and B12, by halving E1. DISJOINT1 then returns (AND (DISJOINT1 B11 B2) (DISJOINT1 B12 B2)).

Note how clever this algorithm is; if at any level it can decide immediately that the objects are disjoint by applying a gross test, it does so. Furthermore, the recursion lets it zero in on the trouble spot very fast. For example, if the bricks are close together at one end of one of them, then it will recurse deeply only on subbricks at that end. Another thing to note is that each test is very fast, so it never considers very long two bricks at opposite ends of the table or obviously overlapping. In fact, the harder it is to decide the question, the harder DISJOINT1 works. Furthermore, it never takes a square root because it always keeps its distances squared. DISJOINT1, however, has one serious difficulty; it never terminates if the objects just touch, rather than overlap or are separate. To avoid this there is an extra parameter (a free variable called FUZEPS (the epsilon of fuzziness)) which it compares to the length of the maximum edge before it recurses and if MAX<FUZEPS it conservatively returns failure. This also makes FINDSPACE somewhat hairier because it must avoid applying DISJOINT1 to the proposed spot and the brick whose surface we are finding space on.

Another problem with RCLEAR is that it too is severely restricted in that it only works for bricks and would be difficult to extend to other classes of objects. We could write a version of DISJOINT1 which worked for tetrahedra, since it is very easy to calculate the circumscribed and inscribed spheres of a tetrahedron. Furthermore, a tetrahedron can be split easily into tetrahedra, making the recursive step possible. Since any polyhedron may be triangulated into tetrahedra, it becomes possible to represent any polyhedron as a union of tetrahedra. Thus two polyhedra are disjoint if and only if all the pairwise combinations of a tetrahedron from each are disjoint. This is very general but it suffers from the disadvantage that the algorithm to triangulate an object may be in general very hairy. This is not so terrible

if we are dealing with many objects because the data base building function may be arbitrarily expensive since most objects are stationary. Another disadvantage is that an object such as a brick will be represented by say 5 (I do not know if this is minimal) tetrahedra, and thus a new program would be perhaps 25 times as expensive as RCLEAR. Although this is only a constant factor, it looks pretty bad to me.

Instead of going all the way to arbitrary polyhedra, we can get some mileage by restricting ourselves to arbitrary convex polyhedra. Any such object can be represented as a system of linear inequalities, each of which determines the halfspace bounded by a plane of which a face of the object is a segment and which is true on the interior of the object. Thus the question of whether or not two convex polyhedra are disjoint is the same as the question of when a system of linear inequalities (the union of the systems which determine each object) is inconsistent. Until the recent visit of Michael Rabin, I thought that this problem was very hard, and I did not consider it as a possible approach. Dr. Rabin, however, showed me that one could consider this a linear programming problem as follows: Let two convex polyhedra P and Q be represented by the following inequalities, where $f_{pi}=0$ ($f_{qi}=0$) is the equation of the ith face of p (q).

$$f_{p1}=p_1+p_{1x}x+p_{1y}y+p_{1z}z\geq 0, \quad f_{p2}=p_2+p_{2x}x+p_{2y}y+p_{2z}z\geq 0, \text{ etc.}$$

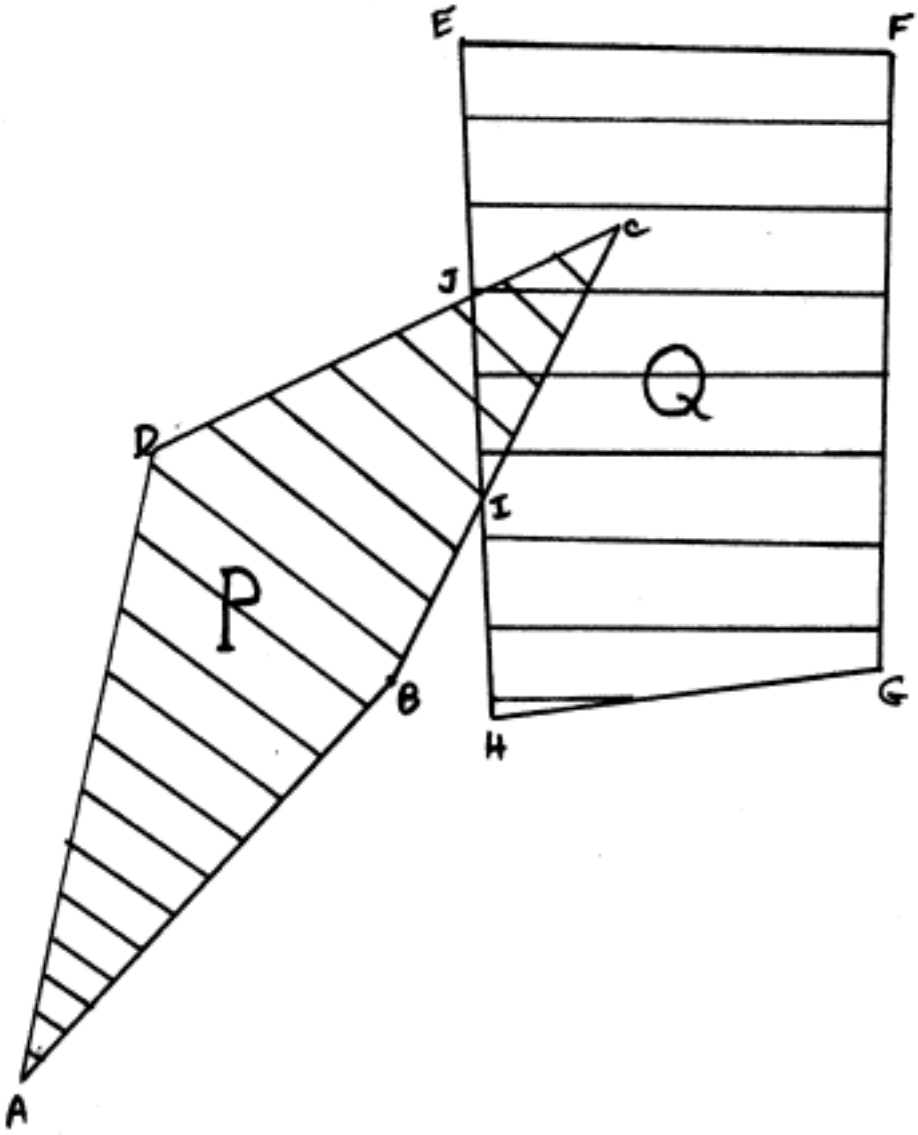
$$f_{q1}=q_1+q_{1x}x+q_{1y}y+q_{1z}z\geq 0, \quad f_{q2}=q_2+q_{2x}x+q_{2y}y+q_{2z}z\geq 0, \text{ etc.}$$

Since P is an object, it is not inconsistent and must have a solution, say (x_1,y_1,z_1) which satisfies it. Thus we can employ linear programming techniques, the simplex method, for example, to find a solution of P which optimizes the form f_{q1} (we actually need not optimize f_{q1}, we can stop whenever $f_{q1}\geq 0$). If the best we can do leaves f_{q1} negative, then the systems

are mutually inconsistent and thus the objects are disjoint. If we can make f_{q1} positive, then we have a solution of the constraints $\{f_{q1}\}$U P and we can use linear programming to repeat the process on f_{q2} with respect to the new constraints $\{f_{q1}\}$U P. We continue this process until we come up with an inconsistency or until all the f_{qi} are satisfied, in which case we decide that P and Q are simultaneously satisfiable, i.e. they intersect.

This turned out to be a common linear programming technique which I found formalized in Linear and Convex Programming by Zukhovitsky and Avdeyeva, as an algorithm (chapter 2.2 - The Simplex Method for Finding a Basic Solution for a System of Linear Inequalities). I have realized a form of this algorithm in a program called FEASGEN, which generates a solution to a system of linear inequalities if there is one. I have embedded this program in a verifier called SIMPCLEAR. Comparisons in efficiency of RCLEAR and SIMPCLEAR are included in the appendix. It is evident, however, that a great deal might be gained in the efficiency of FEASGEN by a judicious choice of the f_{qi} to be tested at any time. Furthermore, if at any time we get a point which satisfies all the remaining f_{qi} (the regions have the point in common) we want to know that immediately, and in fact if we ever come up with a point which satisfies any f_{qj} besides the f_{qi} we are currently considering we want to bring f_{qj} into the list of constraints so as to narrow our search space as fast as possible. I illustrate this as follows: P and Q intersect over the area CIJ. Suppose that the initial proposal we have for P is the vertex A (a worst case). When we choose a function corresponding to a side of Q to increase with respect to the constraints of P, it would not be optimal to choose EF or FG because they are both satisfied everywhere on P. If, however, we increase HG (or EH)

with respect to P we will get to D (or B) after one step (the simplex algorithm steps along the vertices of the object). (Note that in either case HG will become positive, becoming a constraint which cuts off vertex A).



In the next step the only plane worth increasing is EH, which gets us a solution. The algorithm which I have coded contains all of these hacks and thus does not waste time with irrelevancies. The only real overhead is that it carries EF and FG along throughout the calculations because I do not know how to decide that they are subsumed by P.

Why have we gone to so much trouble to find a clever method for determining the disjointness of objects; surely we can do this simply by analytic geometry? If two objects intersect such that one is not wholly

contained within the other, then an edge of one must intersect a face of the other. The straightforward way to check this is to determine the intersection of every edge of one with every face of the other by solving three simultaneous equations in three unknowns (two defining the edge and the third, the face) and then checking that the resulting point is in the correct range. (By Cramer's rule, a 3x3 set of equations takes four determinants, each taking 14 operations; but by collecting common sub-expressions, this can be reduced to a total of 47 operations.) Thus, since a brick has six faces and twelve edges, this method would require 144 such tests (plus a check that one vertex of each is not in the other to exclude the wholly contained case) to determine that two bricks are disjoint. If they are not disjoint the condition would be determined earlier by some particular edge intersecting a particular face in range. In the worst case, however, this test would require more than 144x47=6768 operations (we have ignored the range tests) to prove two bricks disjoint.

The linear programming method should take about n iterations, for n the number of constraints, to determine feasibility. Now each iteration is a Jordan elimination on an nx(d+1) matrix, where d is the dimension of space. A Jordan elimination takes (n-1)xd subtractions and 2x(n-1)xd multiplications for a total of 3x(n-1)xd operations. So for bricks, FEASGEN should take about 3x11x3x12=1188 operations to determine disjoint-ness. Though I don't know about the worst case, the data on FEASGEN, as explained in the appendix, seems to indicate that the worst case is when the objects are disjoint, which takes almost a constant amount of time to compute, whereas the amount of time it takes to decide that two objects are not disjoint is always less and varies considerably, just as in the

exhaustive intersection method examined earlier. This would lead me to surmise, since the observed behavior of FEASGEN matches the a priori behavior of the unwritten exhaustive search, that FEASGEN is really only a very clever bookkeeping method for doing the exhaustive search, with built-in range-testing and wholly-within case exclusion.

Nick Horn and I have slightly investigated other methods. For example, we have conjectured, but not proved, that any two disjoint convex polyhedra can be separated by a plane determined by a vertex of one and an edge of the other. Since it is easy to determine if two points are on the same side of a plane, this may be a useful technique, but we are not yet convinced.

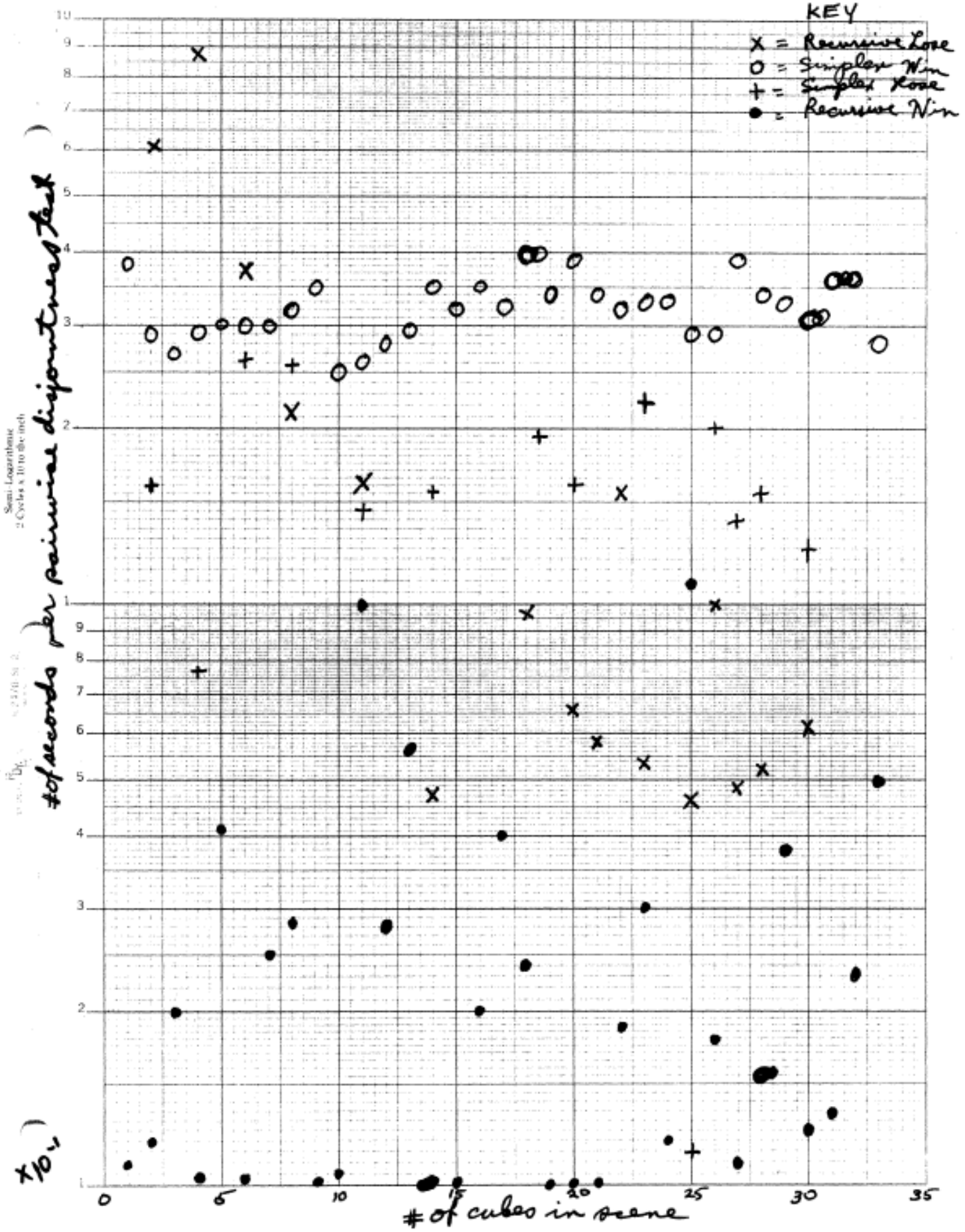
Appendix: A Comparison of RCLEAR and SIMPCLEAR*

Besides implementing RCLEAR and SIMPCLEAR, I also implemented a testing procedure to make it possible to compare their behavior. The testing program generates a brick with a random length, width, depth, position, and orientation in a sharply delimited cubical space. It then applies RCLEAR and SIMPCLEAR to this brick. If they disagree, the program types out an error message and halts until told to proceed. If they agree that the new brick is not clear of other bricks in the space, it repeats the process. If they agree that the new brick is clear of all other bricks in the space, it inserts the new brick into the space and repeats. Since both RCLEAR and SIMPCLEAR work by testing the pairwise disjointness of their argument and the bricks in the model, a common parameter which may be used to analyze the performance of the program is the average time per pairwise comparison of two bricks during the verification process.

The programs were written in LISP and run interpretively (due to the fact that the compiler often produces hard-to-find bugs, programs are not worth compiling unless they are to be run many times so that the time saved justifies the resultant extra debugging time). They therefore ran with a monstrous overhead (especially since they spend much of their time number-crunching) and the time per brick-pair comparison ranged from .1 to 10 sec. RCLEAR had much better best cases and much worse worst cases in general than SIMPCLEAR. Furthermore, as space filled up with bricks, RCLEAR became better and better with respect to SIMPCLEAR. If the new brick was destined to be

*see graph and raw data at end of paper.

discovered clear, it is likely that RCLEAR would spend less than .3 sec. finding that out. This is probably due to the fact that RCLEAR hardly blinks an eyelash at a brick far away because it will be flushed without any recursion, allowing RCLEAR to concentrate on the more difficult, close-by cases. Sometimes, however, a bad case can happen in which RCLEAR recurses for a long time. These cases, however, are rare, and tend to get averaged out when the number of bricks is large. They also tend to happen more often when the brick is to be rejected rather than accepted. This tends to suggest that the circumscribed spheres buy more than the inscribed spheres per recursion. I think that because SIMPCLEAR does more arithmetic than RCLEAR it was hurt more by LISP inefficiency and that if both were hand-compiled into MIDAS, SIMPCLEAR would be much more competitive. Perhaps the best algorithm would combine the two ideas, using the spheres to weed out obvious cases and turning over tough nuts to FEASGEN.


KEY
X = Recursive Lose
O = Simplex Win
+ = Simplex Lose
● = Recursive Win
# of seconds per pairwise disjointness test
x10⁻¹
# of cubes in scene
0
5
10
15
20
25
30
35
Semi-Logarithmic
2 Cycles x 10 to the inch