A NOTE ON
RABIN'S NEAREST-NEIGHBOR ALGORITHM

by

Steve Fortune[†]
John Hopcroft[†]

TR78-340

Department of Computer Science
Cornell University
Ithaca, N.Y. 14853

[†]Research was supported under grant number ONR N00014-76-C-0018.

A NOTE ON RABIN'S NEAREST-NEIGHBOR ALGORITHM

by

Steve Fortune
John Hopcroft

78-340

Department of Computer Science
Cornell University
Ithaca, N.Y. 14853

Abstract

Rabin has proposed a probabilistic algorithm for finding the closest pair of a set of points in Enclidean space. His algorithm is asymptotically linear whereas the best of the known deterministic algorithms take order nlogn time. We show that at least part of the speed up is due to the model rather than the probabilistic nature of the algorithm.

1. Introduction

One notion that has received some attention recently is that of a probabilistic algorithm. An algorithm is probabilistic if at certain steps it chooses a number randomly to determine the next step and at all other steps it is deterministic. Rabin [3] has proposed a probabilistic algorithm for finding the closest pair of a set of points in Euclidean space. His algorithm has the property that for any fixed input the expected running time (taken over all possible sequences of random choices) is linear. It is possible, of course, that for a particularly bad sequence of random choices the algorithm will take longer. The best of the known deterministic algorithms for this problem has running time $O(n \log n)$ [2, 4, 5]

We wish to raise the question of whether the speed of Rabin's algorithm is due to its probabilistic nature or due to the assumed underlying model of computation. In this model the input data are real numbers; the usual arithmetic operations take unit time. In addition, there is a special operation, described below, which performs an operation similar to hashing but is guaranteed to only require unit time. We show, however, that under this model there is a deterministic $O(n \log\log n)$ algorithm. Hence at least part of the speedup is due to the model. It would be interesting to know if the deterministic version could be further improved.

We note that other fast probabilistic algorithms have been proposed [1, 3, 6], but they have the property that for some sequence of random choices an incorrect answer could result. Rabin's algorithm is apparently the only known example of an error-free probabilistic algorithm which runs faster than the deterministic equivalent. It would be interesting to find other examples of this phenomenom.

2. The Algorithm

We describe the special operation, call it FINDBUCKET, in more detail. Given a set of real numbers and an interval size, we wish to partition the set into blocks such that each block is nonempty and contains the reals falling in one of the intervals. To do this we have a table of buckets where the size of the table is as large as the set of numbers. FINDBUCKET, given the arguments a real number r and the interval size, returns the index of the bucket into which r is to be stored. FINDBUCKET returns the same index for all reals falling in the same interval; it is guaranteed to return different indices for reals falling in different intervals. We assume no other relationship between the real and the index returned by FINDBUCKET. Rabin suggests that FINDBUCKET be implemented by dividing the real by the interval size, truncating the quotient to an integer, and hashing on the integer. This would require constant expected time.

The algorithm we describe is completely general and the extension to k-dimensional space will be obvious. For simplicity we will assume that the points are real numbers appearing on the line.

The algorithm is based on the following observation: suppose we can find an interval size such that at most one point falls within each interval, and such that there are two nonempty adjacent intervals. Then to determine the closest pair we need only examine, for each point, the intervals surrounding the point. The number of operations is bounded by the number of surrounding intervals; on the line it is 2n, in the plane 8n, and in k-dimensional space $(3^k-1)n$. We write a recursive procedure FINDINT to find the appropriate interval size.

FINDINT is called with a set of n points as parameter. It selects an initial interval size by dividing the difference between the maximum and minimum point by n. It then removes a point from the set and uses FINDBUCKET to determine into which bucket the point is to be placed. This process continues until some bucket has $\sqrt{n}$ points, at which time FINDINT is called recursively for each bucket which contains two or more points. The new interval size is set to the minimum of the interval sizes returned by the recursive calls. At this point, all buckets are emptied and the scan begins again with the first point not yet processed.

When all points have been scanned, and a tentative interval size found, the entire set of points is placed into buckets at the current interval size. FINDINT is then called recursively on all buckets containing two or more points; the output of the algorithm is the minimum of the size returned by all recursive calls. A pidgin ALGOL description of FINDINT is given below.

Input: A set S of n points.

Output: An interval size such that no two points fall in the same interval and such that there are two adjacent nonempty intervals.

```
PROCEDURE FINDINT(S):
  intervalsize ← (max(S) - min(S)) / n;
  T ← S;
  WHILE T is not empty DO
    WHILE all buckets have fewer than √n points and
          T is not empty DO
      remove a point r from T;
      B ← FINDBUCKET(intervalsize, r):
      insert r into bucket B;
      END;
    FOR each bucket B containing more than one element DO
      intervalsize ← min(intervalsize, FINDINT(B))
      END:
    empty all buckets;
    END:
FOR each r in S DO
  B ← FINDBUCKET(intervalsize, r):
  insert r into bucket B;
  END;
FOR each bucket B containing more than one element DO
  intervalsize ← min(intervalsize, FINDINT(B));
  END;
RETURN(intervalsize):
END
```

To analyze the running time of the algorithm, first note that exclusive of recursive calls, FINDINT takes time cn, for some c. Recursive calls of size less than 256 take only a constant amount of time; we will assume that the cost of such calls are absorbed into the constant c. We show by induction that for $n \geq 256$, FINDINT takes total time dn loglogn, for some d.

Let us call the process of one iteration of the WHILE loop of lines 5-9, i.e. until some bucket gets $\sqrt{n}$ points, a level. Clearly, there are at most $\sqrt{n}$ levels. Let us call the interval size at line 15 the tentative interval size. It is possible that more than one point can fall into a bucket at the tentative interval size. This can happen if two points would fall into the same bucket but were processed at different levels. However, each such bucket can contain at most one point from each level. Hence the number of points per bucket at line 19 is bounded by the number of levels, i.e. $\sqrt{n}$.

From the above discussion it is clear that FINDINT never solves a recursive problem of size bigger than $\sqrt{n}$. If all subproblems were disjoint, it would be easy to show that the running time was O(n loglogn). However it is possible that a point might appear in a recursive call at both lines 11 and 20. Hence a more complicated analysis is necessary.

For the following, let us also say that the processing of lines 19 through 22 is a level by itself. We wish to count the number of intervals which are guaranteed to be nonempty at the interval size at the end of each level. This will enable us to bound the total work involved in the recursive calls.

Suppose the first recursive call at some level has b points. After the call, the points are all separated by the new interval size. But since all the points were in one interval at the start of the level, after the call there are b-1 new nonempty intervals. Suppose the second call at the level has c points. It is possible that these points have already been partially separated by the first call. But since all c points were in one interval at the start of the level, after the second call there are a total of (b-1) + (c-1) new nonempty intervals. Continuing in this way for all recursive calls at a level and all levels, we get that there are at least

$\sum_{i=1}^{k} (b_i - 1)$ nonempty intervals, where k is the number of recursive calls and b_i the size of the ith recursive call. As at the end of the algorithm there are n nonempty intervals, we have

(1) $$\sum_{i=1}^{k} (b_i - 1) \le n$$

The amount of work involved in the recursive calls is bounded by

(2) $$\sum_{i=1}^{k} d b_i \log\log b_i$$

(Note that we have already absorbed the cost of recursive calls of size less than 256 into general overhead. But (2) is certainly an upper bound on the amount of work necessary.) We wish to maximize (2) subject to (1), and the constraint that $2 \le b_i \le \sqrt{n}$ for each i.

We claim (2) is maximized when each b_i except possibly one is $\lfloor\sqrt{n}\rfloor$. To see this first note that if $r \ge s > 2$

$$(r+1) \text{ loglog } (r+1) + (s-1) \text{ loglog } (s-1) \ge r \text{ loglog } r + s \text{ loglog } s$$

hence (2) is augmented by increasing a b_i by 1 at the expense of decreasing a smaller b_j by 1. Second, if $b_i=2$, for some i, since 2 loglog 2 = 0, (2) is augmented by deleting b_i and adding 1 to some other b_j. (1) is then still satisfied.

In this case (all but possibly one $b_i = \sqrt{n}$),
$k \le \left\lceil \frac{n}{\lfloor\sqrt{n}\rfloor - 1} \right\rceil \le \sqrt{n} + 4$ for sufficiently large n.

So we have

$$\begin{aligned} \sum_{j=1}^{k} db_i \text{ loglog } b_i &\le d\sqrt{n} \cdot \text{loglog } \sqrt{n}\, (\sqrt{n} + 4) \\ &\le dn \text{ loglogn} - dn + 4d\sqrt{n}\, (\text{loglogn} - 1) \\ &\le dn \text{ loglogn} - \frac{d}{2}n \end{aligned}$$

where the last line holds as $n \ge 256$. The total work involved is thus bounded by cn + dn loglogn $- \frac{dn}{2}$ which is less than dn loglogn if d>2c.

[1] Angluin, D. and L.G. Valiant. "Fast Probabilistic Algorithms for Hamiltonian Circuits and Matchings", Proc. of the Ninth Annual ACM Symposium on the Theory of Computing (1977), pp. 30-41.

[2] Bentley, J.L. and M.I. Shamos. "Divide-and-Conquer in Multidimensional Space", Proc. of the Eighth Annual ACM Symposium on Theory of Computing (1976), pp. 220-230.

[3] Rabin, M. "Probabilistic Algorithms", appearing in Algorithms and Complexity (1976), Academic Press, N.Y., N.Y., J. Traub, Ed.

[4] Shamos, M.I. "Geometric Complexity", Proc. of the Seventh Annual ACM Symposium on Theory of Computing, (1975), pp. 224-233.

[5] Yuval, G. "Finding Nearest Neighbours", Information Processing Letters, Vol. 5,3 (1976), pp. 63-65.

[6] Solovay, R. and V. Strassen. "Fast Monte-Carlo Test for Primality", SIAM Journal on Computing Vol. 6,1 (1977) pp. 84-85.