BUCKETING ALGORITHMS FOR SORTING, SELECTION AND COMPUTATIONAL GEOMETRY

GARY A. HYSLOP

A DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

IN

APPLIED MATHEMATICAL SCIENCES

UNIVERSITY OF RHODE ISLAND

1993

DOCTOR OF PHILOSOPHY DISSERTATION

OF

GARY A. HYSLOP

APPROVED:

Dissertation Committee

Major Professor Edmund A. Lamagna

[illegible]

Edward J. Carr

Joan M. Peckham

[illegible]

E A [illegible]

[illegible]

DEAN OF THE GRADUATE SCHOOL

THE UNIVERSITY OF RHODE ISLAND

1993

ABSTRACT

In this dissertation we study bucketing algorithms for sorting, selection, Voronoi diagram construction and the closest pair problem. Mathematical analyses of several algorithms are presented. The algorithms are implemented to verify these analyses and to gain insight into their performance on actual machines.

The performances of Distributive Partitioned Sort (*DPS*) and *Quicksort* are compared empirically in a demand paging environment. It is found that *DPS* requires an amount of real memory equal to approximately 40% to 50% of its image size in order to run faster than *Quicksort*. The performance of *DPS* deteriorates rapidly in smaller partitions due to excessive page faulting, while that of *Quicksort* remains fairly constant.

The performance of a variant of Distributive Partitioned Sort is also investigated when the number of buckets is some fraction α of the n items to be sorted. First, a detailed mathematical analysis of the algorithm is presented to determine the expected number of times that each step is executed as a function of both n and α. Then, an implementation of the algorithm is examined for n in the range [10K, 300K] and $0.2 \leq \alpha \leq 1$. The experimental running times are used to ascertain typical constants of proportionality. In so doing, the analysis is adjusted to take into account the effect of a virtual to real address translation buffer. Finally, the value of α which minimizes the running time is determined by a combination of analytical and numerical techniques. It is found that the optimal α lies in the range $0.33 \leq \alpha_{opt} \leq 0.39$ for six different processors on which experiments were conducted. For machines supporting multilevel real memory, α_{opt} varies with n and can drop to as much as 50% of the asymptotic value which holds for both large and small n.

We also present a selection algorithm called *BucketSelect* which runs faster than Floyd-Rivest's *Select* because, while both algorithms use approximately the same number of comparisons, *BucketSelect* uses far fewer data moves (asymptotically zero). Both methods determine an interval in which the kth smallest is expected to lie with high probability. The key difference between the two approaches is that *Select* rearranges the entire data set about two pivot elements, while *Bucket-Select* simply places all items that lie in a critical interval into an auxiliary bucket. In the extremely unlikely event that the item sought does not lie in this critical interval, *Select* is used to find it. The expected running time of *BucketSelect* is governed by the time taken to determine the items in this bucket. A performance evaluation was done which shows that the running time of *BucketSelect* is about 60% of *Select* for finding the median when n is in the range of 50,000 to 250,000 items. The asymptotic percentage is shown to be 53.6%.

We consider two important problems in computational geometry. First, a method which is free from numerical errors is presented for constructing the Voronoi diagram in the plane. The algorithm, which is based on the incremental method of construction, avoids errors by using only integer arithmetic. It performs fewer computations than a similar algorithm that uses floating point arithmetic and produces a correct diagram even when degeneracies occur.

An efficient bucketing algorithm is also given for the closest pair problem. Its expected running time is asymptotically $O(dn)$, where d is the dimension of the space and n is the number of points. We show that for $d \leq 5$, the algorithm performs well for all n but, that when d is large, n must also be large for the algorithm to work efficiently. Expressions are derived for the expected value of the closest interpoint distance within the buckets and the expected number of distance computations performed by the algorithm. The latter is the dominant factor controlling the running time of the algorithm. Empirical results are presented showing that the values obtained for these expressions agree well with the average of these quantities determined from experiments. We also show that the running time of the algorithm is much less than that of several other algorithms for the case when $d = 2$.

Finally, we discuss data transformation methods for dealing with non-uniform data distributions. We conclude that further empirical testing is required to determine which transformation methods work best in practice.

ACKNOWLEDGEMENTS

I would like to thank Professor Edmund A. Lamagna for his numerous valuable suggestions, countless readings of the drafts, and unfailing support over the years. I also wish to thank the other members of my dissertation committee Professors Edward Carney, Edward Grove and John Montgomery. I would also like to thank Professor Bala Ravikumar for suggesting M. Golin's dissertation, which was a valuable inspiration for the analysis in Chapter 6.

I am especially greatful for the help provided by Mr. John Black in the preparation of the graphics figures in this manuscript. I would also like to thank Mr. Philip J. Janus, whose knowledge of VMS was valuable in designing and conducting the experiments, and Mr. Peter Daly who provided access to several newer computers, particularly the DECStation 5000/200 MIPS machine, which were used in the experiments reported in Chapter 3. He also provided support with hardware and implementation concerns.

There are many other people without whom this work would not have been possible. Janet Prichard, the system administrator, kept things running, often under difficult circumstances. She also provided valuable technical support to this researcher. The clerical staff, Ms. Marge White and Ms. Lorraine Berube were a tremendous help throughout my time at U. R. I. Finally, my wife Shirley has been a constant source of support and encouragement throughout this research.

DEDICATED

TO

EDMUND A. LAMAGNA

Advisor, Teacher and Friend

SHIRLEY BOURGEOIS HYSLOP

Loving Wife

BERNARD J. BARRY

Close Friend

PREFACE

The inspiration for this research was the elegant $O(n)$ expected time sorting algorithm by Dobosiewicz called Distributive Partitioned Sort (DPS) [10] and the controversy that resulted from it. We were concerned by the fact that many computer scientists considered sorting to be a closed problem, for which "optimal" $O(n \log n)$ algorithms existed. We also felt that there was a shortage of expected time analyses of these algorithms and a lack of empirical testing to validate the analyses.

In Chapters 2 and 3 we report the results of mathematical anlaysis and empirical testing on DPS and a variant called *Hybridsort*. The results from Chapter 2 have been published in [21]. These results extend earlier studies of DPS by Mr. Philip J. Janus.

The research of Allison and Noga [2], in which the method of buckets was first applied to the selection problem, inspired us to find a algorithm that would outperform the "near optimal" comparison based algorithm of Floyd and Rivest [13]. These results are reported in Chapter 4 and published in [22].

We have followed the research of Asano, Ohya, Sugihara et. al. [3,32,33,38] for the past several years with great interest. This, coupled with our desire to find and efficent algorithm for the construction of Voronoi diagrams in the plane and a concern for numerical accuracy, motivated the research reported in Chapter 5. These results are published in [23]. Our closest pair algorithm, presented in Chapter 6, was inspired by earlier work by Yuval [42], Rabin [35] and Bentley et. al. [4]. The dissertation by Golin [14] provided valuable help in the analysis.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

In this thesis we study a class of problem solving methods known as bucketing algorithms. The problems for which bucketing methods have been successfully applied include sorting, selection and several proximity problems in the area of computational geometry. Many computational problems can be solved by a "divide and conquer" approach wherein a problem of size n is split into two subproblems of approximately half the original size. When, as is often the case, the work needed to merge the two solutions into a global one is $O(n)$, this leads to a recurrence of the form $T(n) = 2T(n/2)+cn$. In such cases, the running time of the algorithm is $O(n \log n)$. In the method of buckets, the data points are assigned to one of αn buckets, where α is a constant. Since the work required to assign each item to a bucket is constant, the time needed to distribute all items is linear. In many cases the number of items in each bucket is bounded by a constant. When this is true, the expected running time of the algorithm is $O(n)$. Usually the worst case complexity is at least $\Omega(n \log n)$, and perhaps even $\Omega(n^2)$. However, the expected complexity is often a more realistic indicator of the performance of such algorithms since the worst cases are extremely unlikely to occur.

We will analyze existing bucketing algorithms more carefully than has been previously done and explore new applications of bucketing methods. In Chapter 2 we compare empirically the performance of Distributive Partitioned Sort (*DPS*) [10] with that of *Quicksort* [18] in a demand paging environment. We analyze mathematically a variant of *DPS* in Chapter 3 where the number of buckets is some fraction α of the n items to be sorted. The optimal value of α is then determined empirically for several implementations. An algorithm which runs faster that Floyd-Rivest's *Select* is presented in Chapter 4. The *Select* algorithm is acknowledged to be close to optimal in the comparison tree model of computation. In Chapters 5 and 6 we investigate two important problems in compupational geometry. A method for constructing the Voronoi diagram in the plane is presented in Chapter 5. This algorithm avoids numerical errors by using integer arithmetic. In addition, it performs fewer computations than a similar algorithm that uses floating point arithmetic. In Chapter 6 we present a method for solving the closest pair problem in d dimensional space in $O(n)$ expected time. It is assumed in Chapters 2 through 6 that the problem inputs are random variables or vectors drawn from a continuous, uniform data distribution. In Chapter 7 we discuss data transformation methods for dealing with non-uniform and unknown distributions. We conclude that further empirical testing is required to determine which methods work best in practice.

1.1. Methods of Analysis

Most analytic work in the literature concentrates on the asymptotic behavior of algorithms. We are interested in a more exact analysis for reasonable input sizes. We have tested actual implementations of the algorithms studied here to verify the analysis and to gain insight into their performance on real machines, rather than abstract computational models.

Decision Trees versus Bucketing

Many classes of algorithms may be represented by a k-ary tree using the decision tree model of computation. Each internal node represents a decision with k alternatives and each external node represents an outcome of the algorithm. If there are j levels in the tree then there can be at most k^j external nodes. When sorting n items there are $n!$ possible outcomes corresponding to all possible permutations of the data. If $S(n)$ is the minimum number of decisions (comparisons) required to sort n elements then, in order for a tree to have a sufficient height to support $n!$ external nodes, we have

$$n! \leq k^{S(n)} \quad \text{or} \quad S(n) \geq \lceil \log_k n! \rceil .$$

Using Sterling's approximation, it can be shown that

$$\lceil \log_k n! \rceil = n\log_k n - n/(\ln k) + {}^1\!/\!_2 \log_k n + O(1) \tag{1.1}$$

Therefore, in the decision tree model of computation the sorting problem is $\Omega(n \log n)$.

In the bucketing model there are no comparisons between items. Thus the lower bound established in the decision tree model no longer holds. Since it is assumed that all of the data can be assigned to one of αn buckets (α a constant) in linear time, the time complexity of distributive sorting is given by the relation $T(n) = cn + \sum_{i=1}^{\alpha n} P(i)T_s(n_i)$, where $P(i)$ is the probability that an item is assigned to the ith bucket and $T_s(n_i)$ is the time required to sort the items in this bucket. Further analysis requires an assumption about the probability distribution of the data. In Distributive Partitioned Sort (*DPS*) [10], the algorithm is applied recursively so $T_s = T$. In *Hybridsort* [29], a secondary sort such as *Heapsort* or *Insertionsort* is used to sort the items in the buckets. Thus $T_s(m)$ may be $O(m \log m)$ or even $O(m^2)$. It has been shown in [6,7] that for many classes of distributions, $T(n) = O(n)$ on the average for both *DPS* and *Hybridsort*.

Asymptotics versus Microanalysis

Much theoretical work in algorithm design is concerned with finding optimal algorithms in the asymptotic sense. For example, in the decision tree model, we know from above that sorting is $\Omega(n \log n)$. Moreover, there are several known algorithms that are optimal in that they perform $O(n \log n)$ comparisons in the worst case or on the average. For instance, *Quicksort* is optimal in the average case while *Heapsort* and *Mergesort* are optimal in the worst case as well as the average case. Unfortunately, there is no way to compare such asymptotically optimal algorithms against each other. It takes a more detailed analysis to show that *Quicksort* is superior to the others on the average [28].

One approach which is often used to compare algorithms with the same asymptotic complexity is to count some "dominant" operation, such as comparisons. This dominance is usually based on frequency of execution rather than unit execution time. The disadvantage of this method is that other instructions which are executed less often may be considerably more expensive and thus significant. For example, exchanges done by comparison/exchange sorts and selection algorithms are neglected in most analyses. An exchange may be over three times as costly as a comparison since it requires three data moves. Some inputs may result in few exchanges while others may require almost as many exchanges as comparisons.

Another problem with asymptotic analysis is that there are practical limitations on the size of n, such as available space. While it is true that asymptotic results hold for large n, the range of n for which such analyses are valid depends upon the problem. For example, suppose the execution time of a particular algorithm is given by the equation

$$T(n) = c_1 n + c_2 n \log n. \tag{1.2}$$

Using asymptotic analysis the algorithm is considered $O(n \log n)$, regardless of the relative magnitudes of c_1 and c_2 or the actual range of n which may be of interest. Suppose the problem size is restricted to the range $n_0 \leq n \leq n_1$. Then $T(n) \leq n(c_1 + c_2 \log n_1)$. If $c_2 \log n_1 \ll c_1$ for the relevant values of n, then the algorithm is linear in this range even though it is $O(n \log n)$ asymptotically.

An alternative to asymptotic analysis is microanalysis. One technique is to determine the time constants for an actual or hypothetical computer. In Knuth's monumental work [27,28], algorithms are expressed in the assembly language of the hypothetical MIX computer. Each MIX instruction is assigned an execution time (most are one or two units). A program is analyzed by counting the execution frequencies of each instruction and computing a sum weighted by the execution times. This approach can be very tedious if a program is long. The method that we will use in this work is to express the algorithm in pseudocode, which is close to a common high-level language such as Pascal or C, and to assign a time constant to each part of the algorithm whose steps have common execution frequencies. These time constants are then measured for representative implementations.

Data Representation and Models of Computation

The way that real numbers are represented in a computer may not only affect the analysis of an algorithm but actually determine whether the algorithm will produce correct results. In theoretical models of computation, a common assumption is that operations on real numbers can be performed in constant time. Computers can represent only a small subset of the rationals using "floating point" numbers containing a fixed number of bits. In this case, the assumption that operations can be performed in constant time is realistic. Many analyses rely on the data points being distinct. For example, distributive algorithms are frequently analyzed by assuming that the inputs are random variables from a continuous distribution. This implies that all inputs are distinct since there is zero probability of drawing two identical real numbers from a continuous distribution. For small n, the probability of two inputs being equal is usually very small. However, as n gets large, there are an increasing number of identical inputs which will eventually invalidate the analysis.

Another issue is computational accuracy. Suppose that a certain geometric algorithm is required to determine if a line passes through a particular point. If floating point arithmetic is used, it may be possible only to determine whether the line intersects a sphere of radius δ surrounding the point. Since in a formal model of computation, this determination could be made exactly, a theoretically correct algorithm may fail to produce correct results when implemented on a computer using floating point arithmetic.

Expected Case vs. Worst Case

Many results in algorithm analysis deal with worst case performance. According to Preparata and Shamos [34], this arises from the mathematical intractability of dealing with the average case and from the difficulty of constructing a probability model which fits the input data. Despite these difficulties, there are several reasons for doing expected case analyses. First, while an algorithm may require a long time to solve the least favorable instances of a problem, the time required on the average may be appreciably shorter. From a practical perspective, the average behavior is the more significant measurement of the algorithm's performance when many instances of a problem have to be solved. Next, consider what would happen if one placed too much emphasis on worst case at the expense of expected case analysis. Then, such popular algorithms as *Quicksort* would not be used despite the fact that, if the partitioning element is randomly chosen, the worst case is very unlikely to occur. In the case of distributive algorithms presented in later chapters, the probability of the worst case occurring is even more remote. For example, Dobosiewicz [11] states that a typical worst case n element input vector for Distributive Partitioning Sort is a sequence of factorials. We will concentrate exclusively on expected case analyses in this dissertation.

1.2. Sorting

Although sorting is probably the most commonly studied problem in computer science, much remains to be learned. In the most general sense the sorting problem may be defined as follows: Given n elements from a set having linear order, arrange them in non-decreasing order. Sorts can be classified as *external*, where there are more records than can be stored in memory at one time, and *internal*, in which all records are stored in random access memory. External sorting procedures have diminished in importance over the years with the advent of inexpensive RAM and virtual memory systems. We will study only internal sorting methods in this work.

Comparison Sorting

Comparisons are the principal operation in all sorting algorithms that conform to the decision tree model. In practical algorithms, either the records must be moved or a table of pointers kept which specifies the sorted sequence. As n increases, the number of operations required to rearrange the records usually does not grow faster than the number of comparisons. Although such operations are not important in asymptotic analysis, they must be taken into account in a more detailed analysis.

There are several known sorting algorithms which are optimal in the asymptotic sense with respect to the decision tree model. Of these, it is generally agreed that the one with the smallest time constant in the expected case is *Quicksort*. Since we will be doing expected case analysis exclusively in this work, we will use *Quicksort* as a "benchmark" against which to compare the performance of distributive sorting algorithms.

Distributive Sorting

When the inputs to a sorting algorithm are integers in a fixed range, it is possible to sort in $O(n)$ time by introducing bit shifts and indirect addressing into the model of computation. This is because computers represent a finite subset of integers using a fixed number of bits. The *RadixSort* algorithm given below [1] indeed runs in $O(n)$ time. Machines also represent a finite subset of real numbers using a fixed number of bits. Assuming that the same number of bits are used to represent these real numbers as are used to represent integers, the corresponding subsets are in one to one correspondence. Thus, the representable real numbers can also be sorted in $O(n)$ time using *RadixSort*.

```
Algorithm RadixSort
// It is assumed that the inputs are unsigned integers composed of $d$ digits. Each digit has $b$
   possible values, $1, 2, \ldots b-1$. //
 1. Assign each integer to one of $b$ buckets, numbered from 0 to $b-1$, such that the integer is placed
    in the bucket whose number is the least significant digit of the integer.
 2. Reassemble the data such that the items in each bucket are contiguous and the data from
    different buckets are in the same order as the bucket numbers.
 3. Repeat $d-1$ times steps 1 and 2. Each time use the next least significant digit for bucket
    assignment.
```

RadixSort is clearly not data dependent since steps 1 and 2 operate in time proportional to n for all inputs and are each performed d times. The storage requirement is proportional to $b+n$ while the time constant is proportional to dn. In most applications, the time constant and space requirements are both high.

Fortunately, there are practical alternatives which sacrifice $O(n)$ worst case performance in return for much more attractive space requirements and a considerably lower expected case time constant. In 1978 Dobosiewicz published the Distributive Partitioned Sort algorithm (*DPS*) [10].

Algorithm DPS.

```
1. Find the maximum (max), minimum (min) and median (med) of the data set.
2. Divide the ranges [$min$, $med$] and ($med$, $max$] each into $n/2$ equal sized buckets.
3. Assign each data value to one of the buckets $b$ as follows:
       if $X_i < med$ then $b = \left\lceil \frac{X_i - min}{med - min} n \right\rceil$
       else $b = \left\lceil \frac{X_i - med}{max - med} n \right\rceil$
4. Apply the algorithm recursively on all buckets containing more than one item.
```

Dobosiewicz [10] showed that the expected running time is $O(n)$ for uniformly distributed data. Assuming the elements are distinct, the worst case performance of the algorithm is $O(n \log n)$. This is because the use of the median guarantees that each pass produces a bucket of size at most $n/2$. Although the median can be determined in $O(n)$ time, the constant of proportionality to do so is high. M. van der Nat [41] suggested replacing the median selection with a two-way merge procedure. If one is willing to accept the premise that the bad cases are extremely unlikely to occur, the median selection can be eliminated altogether. The expected case complexity remains $O(n)$ and the proportionality constant is reduced. Alternatively, a simpler measure of the central tendency such as the midrange can be used to decrease the likelihood of poor performance [25].

Several empirical comparisons have been made between variants of *DPS* and *Quicksort* [10,20,25]. With uniformly distributed data, *DPS* outperforms *Quicksort* for file sizes above a small threshold value in all such studies when system effects, such as demand paging, are ignored.

Van der Nat [41] showed that the $O(n)$ expected time performance holds for uniform data as long as, at each stage of the algorithm, the number of buckets into which data is distributed is proportional to n. Several authors have observed that unnecessary overhead is incurred when the *DPS* algorithm is applied recursively to buckets containing a small number of items. Huits and Kumar [20] suggested using straight *Insertionsort* to sort buckets containing less than 20 items. Sedgewick [36] used such an approach in his implementation of *Quicksort* to reduce the overhead incurred in sorting small subfiles recursively. He found nine to be an optimal "cutoff". Janus and Lamagna [25] also employed this method in their investigation of non-uniform data distributions. Meijer and Akl [29] introduced *Hybridsort* which eliminates secondary distribution passes altogether. They suggested using an $O(n \log n)$ worst case secondary sort like *Heapsort* to insure that the overall algorithm is $O(n \log n)$ in the worst case. Unfortunately, this approach is inefficient when there are many buckets containing a small number of items due to the high overhead in using *Heapsort* to sort small buckets.

A practical alternative is to use an $O(n^2)$ secondary sort. *Insertionsort* is a good choice because it can be applied to the entire data set more efficiently than to the individual buckets. Devroye [7] showed that even if the secondary sort is $O(n^2)$, there are a wide class of distributions for which *Hybridsort* is $O(n)$. The problem is that in some cases the time constant can be quite large. As will be discussed in Chapter 7, $O(n)$ data transformations may be used to reduce the time constant and to augment the class of distributions for which $O(n)$ behavior can be achieved.

Hybridsort has an important advantage over *DPS* in that it is easily adaptable to sorting alphanumeric keys. A fixed number of most significant bits of the key can be mapped, using an order preserving transformation, to an integer. This integer is then used for the distribution step. Finally, a comparison-based secondary sort is used to complete the job.

Even if one is willing to assume that either the distribution underlying the input data is uniform or that an $O(n)$ data transformation can be applied to smooth the data, there are still several issues that require investigation. Although it is known that the asymptotic behavior of *Hybridsort* is $O(n)$ in the expected case, there are several factors which must be considered when a detailed analysis is done. These include the optimal number of buckets to use and the possible consequences of memory management effects such as demand paging and caching. In Chapter 2 we investigate the effects of demand paging by comparing the performance of *DPS* and *Quicksort* in a demand paging environment. *Quicksort* accesses items in an array systematically by working from the front and back toward the middle. On the other hand, *DPS* references items randomly through pointers. It would seem that, due to this "locality of reference," *Quicksort* would perform better under demand paging because it references only a small number of pages at one time.

Most modern computers have memory caches which allow more rapid access to data that are used frequently and slower access to data used infrequently. Therefore, the assumption that a data value may be assigned to a bucket in constant time, independent of the value or order of the data, may not hold. In Chapter 3 we investigate these memory management effects on several computing platforms.

In Chapter 3 we also examine α, the ratio of buckets to n, to determine its optimal value. In order for *Hybridsort* to be $O(n)$, it is only necessary that α be $O(1)$. A detailed analysis reveals the following tradeoff. Overhead is required to initialize each bucket and to process each non-empty bucket. In addition, the greater the number of buckets, the less locality of reference. On the other hand, the work done by the secondary sort increases when the number of buckets decreases. Even if the sort is carried out in real memory, hardware effects such as data caches may affect the algorithm's performance. Since a detailed analysis depends on machine constants, we have tested *Hybridsort* on several platforms. It was found that that the optimal α does not vary greatly from machine to machine. Finding a good value for the parameter α may save considerable space as well as optimize the running time of the algorithm.

1.3. Selection

Many situations, including numerous statistical applications, call for selecting one or more ranked items. One example is finding confidence intervals about the median. Another application is to use ranked data from a small sample to divide items to be sorted into approximately equal proportions.

In this section we will employ the notation used in [12]. Let X be a finite set of distinct elements which are ordered such that for $x_1, x_2 \in X$ either $x_1 < x_2$ or $x_2 < x_1$. We define $k\theta X$ to be the kth smallest element of X and $x\rho X$ to be the rank of x in X. Thus $(x\rho X)\theta X = x$.

It has been known for several years that the selection problem is $\Theta(n)$ in the worst case [1]. In

1961, Hoare [17] published a selection algorithm called *Find*.

Algorithm Find

```
1. Select an element, $u$, at random from $X$.
2. Partition $X$ into $A = \{x \in X : x < u\}$, $B = \{u\}$, and $C = \{x \in X : x > u\}$.
3. If $u\rho X = k$ then return $u$
   else if $u\rho X < k$ apply the algorithm recursively to $C$
   else apply the algorithm recursively to $A$.
```

Although the performance of the algorithm is $O(n^2)$ in the worst case, it is shown in [1] to be at least five times faster on average than an efficient $O(n)$ worst case algorithm.

In 1975, Floyd and Rivest [13] published Algorithm *Select* which, by proper choice of partitioning elements, achieves an even better, and close to optimal, expected case performance with respect to the number of comparisons.

Algorithm Select

```
1. Select a sample $S$ from $X$ whose cardinality, $s$, is $o(n)$.
2. Find $k(s/n)\theta S$ by applying the algorithm recursively and compute its standard deviation.
3. Select $u$ and $v$ from $S$ such that $u\rho S$ and $v\rho S$ are $d$ standard deviations less than and greater
   than $k$ respectively.
4. Partition $X$ into $A = \{x \in X : x < u\}$, $B = \{x \in X : u \leq x \leq v\}$ and $C = \{x \in X : x > v\}$.
5. If $u\theta X \leq k \leq v\theta X$ apply the algorithm recursively to $B$
   else if $u\theta X > k$ apply the algorithm recursively to $A$
   else apply the algorithm recursively to $C$.
```

Because $k\theta X$ is expected to lie in B with high probability and the expected cardinality of B is $o(n)$, the running time of the algorithm is governed by the time required to perform step 4.

Allison and Noga [2] first applied the method of buckets to the selection problem. Their algorithm involved distributing the items into $c\sqrt{n}$ buckets and determining the bucket in which the desired item was located by counting the bucket cardinalities. The empirical results that they gave showed that *Select* was superior in all cases.

We have done extensive research which indicates that the use of multiple buckets in an attempt to find an algorithm that is faster than *Select* is doomed to failure because of the overhead incurred when maintaining the buckets as linked lists. However, the following variation produces an algorithm that is faster on average than *Select*. We observe that *Select* identifies the members of all three sets A, B and C despite the fact that it relies on the item being sought belonging to B with high probability for its superior expected time performance. We therefore propose that only the members of B be isolated. In the unlikely event that $k\theta X \notin B$, *Select* can be applied to the entire data set. A single bucket, which can be implemented as an array, can be used to store the members of B. The operations required to isolate B are $o(n)$ and thus asymptotically zero. The dominant part of the algorithm is, therefore, the number of operations required to isolate B.

Since only one bucket is used, membership in the bucket should be determined by comparisons. The algorithm will perform the same number of comparisons, on the average, as *Select* but with far fewer data moves. It will always outperform *Select* as long as $k\theta X$ is successfully isolated in B. In Chapter 4 we analyze an implementation of this algorithm and compare its performance to *Select*.

1.4. Computational Geometry

Our research has important implications in the rapidly growing field of computational geometry, which concerns the algorithmic study of geometric problems. One particular class of problems for which the method of buckets is well suited is that involving the proximity of points. Two problems in this class that we investigate are Voronoi diagram construction and closest pair.

Construction of the Voronoi Diagram

The Voronoi diagram may be defined as follows: Given n points in the plane, partition the plane into n regions where each region, R_i, is the space that is closer to point i than the other $n-1$ points. The ith point is referred to as the *generator* of the ith region. The regions R_i are convex polygons which are called Voronoi regions. The vertices of these regions are called Voronoi points and the boundaries of the regions are called Voronoi polygons.

The problem of constructing the Voronoi diagram in the plane has received considerable attention in the literature over the last fifteen years. Several methods [33,19,34] based on the principle of divide and conquer have been proposed for Voronoi diagram construction that achieve optimal $O(n \log n)$ worst case performance in the decision tree model. Using the method of buckets it is possible to achieve $O(n)$ behavior, at least on the average. Bentley, Weide and Yao [4] were the first to show this. Their algorithm, however, is impractical to implement. A practical method achieving $O(n)$ average case performance was published by Ohya et. al. [32,33,3]. Their method is based on the incremental construction algorithm of Green and Sibson [15] in which generators are added one at a time. If the generators are introduced in arbitrary order, the construction has a worst case performance of $O(n^2)$. However, if the generators are added in a certain canonical order, Ohya et. al. have shown that, under suitable assumptions about the data distribution of the generators, it is possible to achieve expected $O(n)$ performance.

A major problem with the incremental method concerns the computational accuracy and how to handle so-called degeneracies. The incremental construction algorithm adds a new generator by first finding its nearest neighbor, p, among the generators already added. It is then necessary to find the edges in the Voronoi polygon of p that intersect the perpendicular bisector of p and the new generator. A degeneracy occurs when this bisector intersects a vertex of the Voronoi polygon. If the computation is precise, this degeneracy can be handled, otherwise the algorithm may not terminate. Sugihara and Iri [38] proposed a solution to the problem in which topological properties of the diagram are given highest priority and numerical computations are employed only to resolve ambiguities. Using this approach, they are able to construct a diagram for one million generators using single-precision arithmetic. The problem is that many important properties of the diagram are sacrificed. For example, they require that there be as many Voronoi regions as generators but it is possible, depending on the computational accuracy employed, that some regions may have more than one generator while others have none.

The work of Sugihara and Iri [38] shows clearly the relationship between the generators and the Voronoi points. Since only a finite subset of rational numbers can be represented on a real computer, it is reasonable to impose a restriction on the precision of the generators.

In Chapter 5 we discuss improvements to the incremental method of Voronoi diagram construction which handle correctly so-called "degeneracies" and eliminate computational inaccuracies. To accomplish this we avoid using floating point arithmetic. We will show that, if integer coordinates are used for the generators, the Voronoi points may be represented by an integer triple (two rational coordinates with a common denominator). The exact precision required to compute and to represent the Voronoi points are easily calculated functions of the precision of the generators. Computations are restricted to addition, subtraction, multiplication and comparison of integers.

The Closest Pair Problem

The closest pair problem may be stated as follows: Given n points in d dimensional space, find the two points that are the closest. A brute force approach is to find the minimum of all pairwise distances between points using $O(n^2)$ comparisons. Optimal decision tree algorithms usually employ a divide and conquer approach. The idea is to first divide the space into two subsets of points. Next, the minimum inter-point distance in each subset is found by applying the algorithm recursively. The results are then combined by processing a slice at the junction of the subsets which can be shown to contain at most a constant number of points. Bentley, Weide and Yao [4] have given $O(n)$ expected time algorithms for solving a number of closest point problems in d dimensional space. For example, they show how under certain conditions on the probability density function that the *all nearest neighbors* problem can be solved in linear expected time. Since one of the pairs of nearest neighbors is the closest pair, it immediately follows that the closest pair problem can also be solved in $O(n)$ expected time using $n-1$ additional comparisons. It may still be possible, however, to obtain a more efficient closest pair algorithm since finding all nearest neighbors is not a prerequisite for solving the problem. Of course, such an algorithm would be still $O(n)$, but with a smaller constant of proportionality.

In Chapter 6 we present a closest pair algorithm, *CPair*, that is more efficient than existing algorithms and is easily generalized to higher dimensions. *CPair* is based on a theorem by G. Yuval [42] which may be stated as follows: It is possible to construct $d+1$ partitions of d space with every pair of points less that δ apart lying in the same region in at least one partition. Yuval used this observation as the basis for an $O(n \log n)$ worst case divide and conquer algorithm. However, by coupling this theorem with the method of buckets, we have developed *CPair* which does run in linear expected time.

Rabin [35] published a probabilistic closest pair algorithm in 1976 that is based on Yuval's theorem and which has linear expected time performance. His method is to use a large number of buckets whose size is determined by the closest pairwise distance among points in a random sample. The key idea is that the number of *non-empty* buckets is bounded above by a constant times n. At the expense of considerable overhead, the problem is reduced to sorting a sequence of integers, each corresponding to the index of a non-empty bucket. Rabin claims that this can usually be done by hashing in $O(n)$ time. The difficulty is that, even for uniformly distributed data points, the integer sequence is highly non-uniform.

CPair solves the closest pair problem in d dimensional space more efficiently by using a number of buckets proportional to n, but no more than $n-1$, to insure that at least one bucket contains two points. The first distribution stage determines a tentative closest pair and the distance between them. At most $d+1$ subsequent distributions are required. At each distribution stage, the majority of points used in the previous stage are eliminated from consideration for closest pair. The complexity of this algorithm is $O(dn)$ for sufficiently smooth data compared to the exponential growth with d

of the methods given by Bentley, Weide and Yao [4].

1.5. Data Transformations

When the data distribution is known, it is possible to apply a transformation to the inputs in such a way that the transformed data is uniform. This was demonstrated in the case of sorting by Meijer and Akl [29]. However, there are times when it is difficult to assume a particular probability distribution for all instances of the problem. Moreover, the relative frequency of problem instances may change in an unpredictable way. One remedy is to employ a probabilistic approach, such as the one used in the selection algorithms in Chapter 4 and which Rabin has applied to the closest pair and other problems [35]. Another method is to adapt the transformation methods to the situation where the probabilty density is unknown. We discuss the issue of dealing with non-uniform data distributions further in Chapter 7.

CHAPTER 2

PERFORMANCE OF DISTRIBUTIVE PARTITIONED SORT IN A DEMAND PAGING ENVIRONMENT

2.1. Introduction

Distributive Partitioned Sort, or *DPS*, is an internal sorting technique described in Dobosiewicz [10]. Empirical evidence presented there and in Janus and Lamagna [25] suggests that the algorithm runs considerably faster than several commonly used sorts including *Quicksort* [18,36].

Unfortunately, previous experiments with *DPS* were conducted in such a way that all of the data and pointers resided in main memory during the entire sorting process. Such an assumption may not be realistic, in practice, when large or even moderate sized files are sorted on virtual memory computer systems [5].

In this chapter we investigate the effect of demand paging on *DPS*, comparing its performance to that of *Quicksort*, as the amount of real memory available to the sorts is reduced.

2.2. Experimental Design

Two algorithms were studied – *DPS* with midrange selection as described by Janus and Lamagna [25] and the version of *Quicksort* described by Sedgewick [36]. The algorithms were coded in Pascal for implementation on a VAX-11/750 running Version 4.2 of VMS. The data to the sorts consisted of uniformly distributed real numbers in the unit interval.

Each datum was a single precision real number occupying four bytes of storage. The bucket headers and linked list pointers for *DPS*, and the pointers to the endpoints of partitions in *Quicksort*, were implemented as longword integers occupying four bytes. *DPS* requires approximately $3n$ longwords, or $12n$ bytes, to sort n items since it associates one pointer with each datum and employs a number of buckets equal to the size of the file. *Quicksort* uses only $4n$ bytes for data and has a maximum stack size of $\log_2 n$ pairs of endpoints occupying at most $8 \log_2 n$ bytes.

The experiments were conducted in such a way that the sorts ran stand-alone on the system. The only other processes executing were essential operating system routines like the swapper. For each run, the *working set* W_s, or maximum amount of physical memory available to the process, was specified as a parameter to the operating system. There is a minimum working set, W_m, of about 95 page frames that is required to load and execute an object module on the system.

The *capacity* W_c, or minimum number of pages that must be allocated for *DPS* to reside entirely in main memory, was determined for each file size considered. Experiments were then run on both *DPS* and *Quicksort* with working sets expressed as percentages W of this capacity (less W_m) given by $(W_s - W_m)/(W_c - W_m)$.

The virtual address space of a process running under VMS consists of three parts:

(1) pages residing in physical memory allocated to the process,

(2) pages residing in physical memory but not available to the process, called the *page file*, and

(3) pages residing on disk.

A *page fault* occurs whenever a reference is made to a location in a page of the second or third type. Each page on the VAX contains 512 bytes. Details of the VAX/VMS memory management system can be found in [8].

In most of the experiments, all page faults were of the real memory to real memory type. The handling of an in-memory page fault consists of updating the allocation of storage between pages of the first and second types enumerated above. The time required by the operating system to process such a fault is considerably less than that to handle one involving disk access.

Because large, uninitialized data areas are initially paged to the page file under VMS, it is actually impossible to run *DPS* fault-free since all pointers are outside of real memory when the sort begins. In order to obtain a fault-free running time, it is therefore necessary to reference all of the data and pointers once before sorting. This causes all of the required pages to reside in memory

Working set W	n=10000		n=20000		n=40000		n=80000	
	DPS	QS	DPS	QS	DPS	QS	DPS	QS
100	4.30	5.23	8.67	11.14	17.86	23.84	36.85	50.46
90	4.42	5.22	8.96	11.14	18.54	23.87	38.21	50.49
80	4.46	5.21	9.22	11.13	19.40	23.85	39.93	50.41
70	4.55	5.22	9.30	11.15	19.29	23.86	39.97	50.35
60	4.57	5.22	9.63	11.15	20.07	23.85	40.55	50.37
50	4.76	5.22	10.09	11.14	22.40	23.83	48.50	50.55
40	5.18	5.22	12.44	11.17	28.34	23.87	63.10	50.45
30	6.77	5.25	17.60	11.25	41.17	24.04	92.18	50.92
20	11.48	5.28	34.04	11.39	70.64	24.22	152.13	51.20
10	25.47	5.37	61.99	11.49	118.84	24.50	224.37	52.04
W_m	-	5.50	-	12.13	-	26.92	-	59.46

Table 2.1. Run times (in seconds).

Working set W	n=10000		n=20000		n=40000		n=80000	
	DPS	QS	DPS	QS	DPS	QS	DPS	QS
100	0	0	0	0	0	0	0	0
90	199	0	441	0	950	0	1931	0
80	236	2	838	0	2156	0	4656	0
70	369	2	951	0	2147	0	4702	0
60	412	3	1436	0	3305	0	8207	0
50	715	3	2108	2	6779	0	17572	0
40	1212	3	5329	3	15912	2	39723	0
30	3118	60	12308	137	35507	269	88881	634
20	8072	84	31699	305	78808	469	176350	939
10	20519	198	57839	483	129840	931	276113	2143
W_m	-	295	-	770	-	1778	-	4139

Table 2.2. Number of page faults.

allocated to the process, and no further paging will occur if the working set is at capacity.

2.3. Results and Conclusions

Table 2.1 shows the running times of *DPS* and *Quicksort* for four different file sizes: $n = 10{,}000$, 20,000, 40,000, 80,000. Table 2.2 shows the number of page faults generated. Some variation in the run times was observed due to a degree of randomness in the behavior of the system when processing page faults. This effect is more noticeable with very small working sets or very large file sizes. To

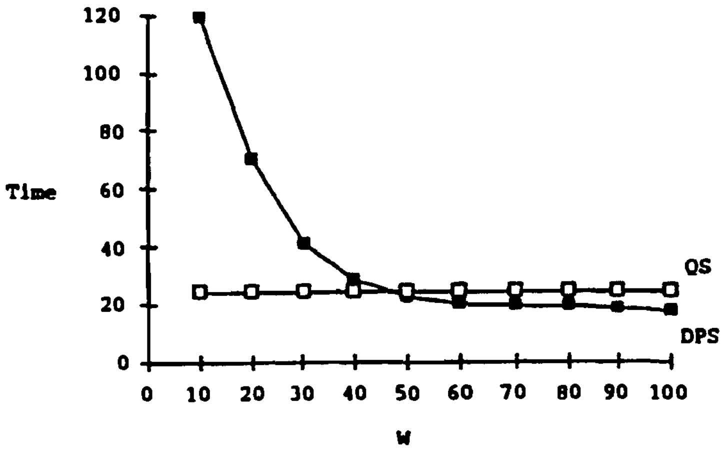


Figure 2.1. Run times for $n = 40{,}000$.

mitigate this difficulty, the data reported in the tables are the averages of five runs on the same input.

Figure 2.1 plots the times for $n = 40{,}000$. It can be seen that the crossover point for the two algorithms occurs just below $W = 50\%$, with *Quicksort* outperforming *DPS* in smaller partitions. Below the crossover, the running time of *DPS* rises dramatically due to excessive page faulting, or *thrashing*. The curves obtained for the other values of n are similar.

The results also show that the running time of *Quicksort* increases very slowly as the size of the memory partition is reduced. In fact, one should not expect any rise at all until $W \approx 33\frac{1}{3}\%$ is reached since *Quicksort* requires only about one-third the space of *DPS*. Theoretically, the running time of *Quicksort* will rise as sharply as that of *DPS* if the partition is reduced enough, or if the file size n becomes sufficiently large. However, this will not occur because the partition cannot be smaller than W_m. At W_m, the running time of *Quicksort* is only 5% greater than its fault-free value for $n = 10{,}000$ and 18% greater for $n = 80{,}000$.

The rise in the running time of *Quicksort* is small because of a phenomenon known as *locality of reference*. *Quicksort* has a high degree of locality since it accesses items in an array by systematically working from the front and the back toward the middle. At any point, only three pages of data are *reference*. *Quicksort* has a high degree of locality since it accesses items in an array by systematically working from the front and the back toward the middle. At any point, only three pages of data are of interest – those containing the pivot and the items scanned by the forward and backward moving pointers. These pointers cross page boundaries infrequently. *DPS*, on the other hand, accesses data items randomly via linked lists of pointers. This greatly increases the number of page faults generated.

In conclusion, while *DPS* runs faster than *Quicksort* when all of the information associated with the sorts resides in physical main memory, its performance in a demand paging environment may be somewhat less impressive. This is caused by two principal factors. First, *DPS* uses approximately three times as much storage as *Quicksort*. Secondly, *DPS* accesses data randomly via linked lists of pointers, while *Quicksort* exhibits an extremely high locality of reference.

Empirical results presented here indicate that an amount of memory equal to about 40% to 50% of that required for the data plus pointers must be available for *DPS* to outperform *Quicksort*. These results were obtained when most page faults were of the real memory to real memory type. The performance of *DPS* would have suffered more if disk accesses were required to resolve page faults.

CHAPTER 3
SORTING BY DISTRIBUTIVE PARTITIONING WITH FEWER THAN *n* BUCKETS

3.1. Introduction

Distributive Partitioned Sort, or *DPS*, is a fast internal sorting technique first described by Dobosiewicz [10]. The method has an expected running time which is linear in the number of items n to be sorted when it is applied to data with probability densities that are bounded and have compact support [7]. Comparison-based sorts require $\Omega(n \log n)$ key comparisons and cannot achieve linear running times.

One of the problems associated with *DPS* is the extra storage required for pointers. In the distribution phase, the items are placed into one of n buckets, implemented as linked lists. This requires $2n$ pointers — n list headers plus one pointer to place each item on a list. It has been reported in Chapter 2 that at least 40% of the memory required for the input data and pointers must be available for *DPS* to outperform *Quicksort* in a typical demand paging environment.

One way to reduce the space overhead is to use fewer buckets. *DPS* can be speeded by using a final insertion sort pass to order buckets containing multiple items below some cutoff (9 or 10 items works well). As shown in [25], the probability of a bucket being above such a cutoff is extremely small when a transformation is employed to smooth the data. This suggests that fewer than n buckets might be used. In fact, the run time could actually be reduced since there is an overhead associated with maintaining each bucket.

Meijer and Akl [29] have described an algorithm called *Hybridsort* having a single distribution pass. Buckets containing more than one item are ordered with a secondary sort whose run time is proportional to $m \log m$, where m is the number of items to be ordered. Devroye [7] shows that even a simple $O(m^2)$ secondary sort leads to $O(n)$ behavior for *Hybridsort* over a wide class of data distributions including the uniform. In this chapter we analyze an implementation of *Hybridsort* called *BucketSort* which uses insertion sort to accomplish the secondary sorting. Insertion sort has the desirable property that it can be applied to an array of appended buckets with no more work than applying it to the members of each bucket separately.

We examine the performance of *BucketSort* when the number of buckets used is some fraction α, $0.2 \leq \alpha \leq 1$, of the file size n. First, an overview of the algorithm is presented. A detailed rendering of the procedure is then analyzed to determine the expected number of times each step is executed as a function of α and n assuming the data to be sorted are uniformly distributed. We restrict ourselves to this case since it has been shown that many nonuniform distributions may be transformed into the uniform distribution with only a small penalty in the running time [25,29]. The execution counts are related to actual running times by examining several implementations of the algorithm. In doing so we develop a model which considers the effect of multilevel memory on the time to access data. The value of α which minimizes the running time is determined using numerical techniques for several different machines.

3.2. Analysis of the *BucketSort* Algorithm

The *BucketSort* algorithm consists of six steps:

1. initialize an array of αn bucket headers,
2. determine the maximum and minimum elements of the n items to be sorted,
3. distribute the n data items into buckets,
4. chain the buckets into a single linked list,
5. rearrange the array of data items according to the order of the corresponding pointers in the list, and
6. perform a single insertion sort pass over the entire data array.

In this section, the salient features of each phase are presented and an analysis of its expected running time is developed. Expressions for the running time of each step are given under the assumption that the time to access data from memory is constant. A pseudo-coded version of the algorithm appears in Appendix A, where each line is annotated with the expected number of times that it is executed.

Step 1. Bucket Initialization

In this phase, the headers for each of αn buckets, maintained as singly linked lists, are assigned null values. The running time is given to a close approximation by $T_1 = c_1 \alpha n$.

Step 2. Determining the Maximum and Minimum

To locate the maximum and minimum, each consecutive pair of items is compared. The larger is then compared to the current maximum, and the smaller to the current minimum. Hence the total number of comparisons is $3/2\, n$. Assuming all $n!$ permutations of the data are equally likely to occur, the probability that the ith pair contains the largest (or the smallest) of the first i pairs is $1/i$. Thus the expected number of times the maximum (or minimum) is updated in the $n/2$ comparisons it makes is given by $\sum_{i=2}^{n/2} 1/i = H_{n/2} - 1$, where H_m is the mth harmonic number. Since $H_{n/2} \ll n$, the work done in making comparisons will dominate the time spent in updating the maximum and minimum, and we have $T_2 = c_2 n$.

Step 3. Bucket Distribution

The range of values is divided into αn equal length intervals. Then each item is placed in a bucket by computing the fraction of the way into the range it lies. The running time for this phase is proportional to the number of items being distributed, $T_3 = c_3 n$.

Analysis of Item Distribution into Buckets

To obtain execution counts for the bucket chaining and insertion sort phases, the distribution of items into buckets must be examined. Since the input is uniformly distributed, the number of items in any bucket is a binomially distributed random variable with parameters n and $\frac{1}{\alpha n}$. $F(i, n, \alpha)$, the expected fraction of buckets containing exactly i items, is given by

$$F(n, i, \alpha) = \binom{n}{i} \left(\frac{1}{\alpha n}\right)^i \left(\frac{\alpha n - 1}{\alpha n}\right)^{n-i} \tag{3.1}$$

	α								
	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
$B(\alpha)$	0.199	0.289	0.367	0.432	0.487	0.532	0.571	0.604	0.632
$D(\alpha)$	2.812	1.544	0.981	0.681	0.500	0.383	0.303	0.246	0.203

Table 3.1. Values of $B(\alpha)$ and $D(\alpha)$.

When $\frac{1}{\alpha n}$ is small and n is large, (3.1) may be approximated closely by the Poisson distribution which is independent of n [26]. In this case (3.1) may be written as

$$F(i,\alpha) = \frac{1}{\alpha^i i!} e^{-1/\alpha}.$$

Step 4. Bucket Chaining

The buckets are chained together into a single linked list by copying the header for each into the tail of the preceding one. The implementation consists of a loop that is executed once for each of the αn buckets. The expected number of nonempty buckets which must be processed is given by $B(\alpha)n$, where $B(\alpha) = \alpha(1 - F(0,\alpha))$. Typical values for $B(\alpha)$ are shown in Table 3.1. The number of pointers which must be traversed is $(1-\epsilon)n \approx n$, where ϵ is the miniscule fraction (expected value $\frac{1}{\alpha n}$) of items in the last bucket. The running time of this step is closely approximated by a linear combination of αn, $B(\alpha)n$ and n, $T_4 = [c_{4,\alpha}\alpha + c_{4,b}B(\alpha) + c_{4,n}]n$.

Step 5. The MacLaren Routine

In this phase, the items in the data array are reordered according to a list of pointers using a technique developed by M. D. MacLaren [28, p. 596] that does not use any additional storage. The implementation consists of a loop that is executed once for each data item. Within this is an inner loop that is executed once for each item that must be displaced from its original position in the array. The only items which are not displaced are those that complete a cycle in the permutation. The expected cycles can be shown to be H_n [27, p. 176 and 28, p. 596], and so the inner loop is executed $n - H_n$ times. Since $H_n \ll n$, the running time of this step can also be regarded as directly proportional to n, $T_5 = c_5 n$.

Step 6. Insertion Sort

A single insertion sort pass over the entire array is used to order the items in each bucket. Since the buckets are properly ordered with respect to one another, an item will not be moved by this process beyond the boundaries of the bucket into which it is placed. Thus, a single insertion sort pass acts as a sequence of independent sorts on the individual buckets. Two quantities are of importance here — the number of items requiring insertions and the number of data moves.

Consider the case when a bucket containing m elements is ordered by insertion sort. When the jth item is processed, the $j-1$ succeeding elements in the bucket have already been ordered. Since any permutation of the items within a bucket is equally likely to occur, the probability that the jth item is smallest is $1/j$ and so an insertion will be performed with probability $1-1/j$. Summing from $j = 1$ to m gives the expected number of insertions performed on a bucket of size m,

$$\sum_{j=1}^{m}(1 - 1/j) = m - H_m.$$

To obtain the expected number of insertions per bucket, $D(\alpha)$, this quantity must be multiplied by the expected fraction of buckets containing m items, $F(m, \alpha)$, and summed over all possible values of m,

$$D(\alpha) = \sum_{m=1}^{n} \frac{1}{\alpha^m m!} e^{-1/\alpha}(m - H_m).$$

This sum is best dealt with by treating each term separately. The first component is easy,

$$\sum_{m=1}^{n} \frac{m}{\alpha^m m!} e^{-1/\alpha} \approx \frac{1}{\alpha} e^{-1/\alpha} \sum_{m=0}^{\infty} \frac{(1/\alpha)^m}{m!} = 1/\alpha$$

from the Taylor series expansion for $e^{1/\alpha}$. The second component reduces to the exponential generating function, $G(z)$, for the harmonic numbers with $z = 1/\alpha$,

$$G(z) = \sum_{m=0}^{\infty} H_m \frac{z^m}{m!},$$

when $e^{-1/\alpha}$ is factored out of the sum. Therefore, the expected number of insertions per bucket is given by

$$D(\alpha) = 1/\alpha - e^{-1/\alpha} G(1/\alpha).$$

Unfortunately, no closed form in terms of elementary functions exists for $G(z)$ and $D(\alpha)$ must be determined numerically. The harmonic numbers are defined by the recurrence

$$H_m = H_{m-1} + \frac{1}{m}$$

for $m \geq 1$ with $H_0 \overset{\text{def}}{=} 0$. Multiplying both sides by $mz^m/m!$ and summing over all $m \geq 1$ yields

$$\sum_{m=1}^{\infty} mH_m \frac{z^m}{m!} = z \sum_{m=1}^{\infty} H_{m-1} \frac{z^{m-1}}{(m-1)!} + \sum_{m=1}^{\infty} \frac{z^m}{m!}.$$

Again using the Taylor series for e^z and letting

$$G'(z) = \frac{dG(z)}{dz} = \sum_{m=1}^{\infty} mH_m \frac{z^{m-1}}{m!},$$

this is equivalent to

$$zG'(z) = zG(z) + e^z - 1.$$

The resulting differential equation may be solved using the integrating factor e^{-z},

$$\frac{d}{dz}[G(z)e^{-z}] = e^{-z}G'(z) - e^{-z}G(z) = \frac{1-e^{-z}}{z}$$

and so we obtain

$$G(z) = e^z\left(e^{-1}G(1) + \int_1^z \frac{1-e^{-t}}{t}dt\right).$$

$D(\alpha)$ can now be computed numerically by taking a sufficient number of terms in the series expansion starting with an initial value of $G(1) = 2.50259$. Representative values are presented in Table 3.1.

The expected number of data moves in insertion sort is given by

$$\sum_{j=1}^{m} (j-1)/2 = m(m-1)/4,$$

where m is the number of items in a bucket [28, p. 82]. To obtain the expected number of data moves per bucket, $E(\alpha)$, this quantity must be multiplied by the fraction of buckets containing m items, $F(m,\alpha)$, and summed over all possible values of m,

$$E(\alpha) = \sum_{m=0}^{n} \frac{1}{\alpha^m m!} e^{-1/\alpha} \frac{m(m-1)}{4}.$$

This may be simplified to

$$E(\alpha) = \frac{e^{-1/\alpha}}{4} \sum_{m=0}^{n} \frac{m(m-1)}{\alpha^m m!} \approx \frac{e^{-1/\alpha}}{4\alpha^2} \sum_{m=2}^{\infty} \frac{(1/\alpha)^{m-2}}{(m-2)!} = \frac{1}{4\alpha^2}$$

by again recognizing that the second sum is just the Taylor series expansion for $e^{1/\alpha}$.

The insertion sort consists of a main loop that is executed $n-1$ times. As each new item is considered, it is compared with its successor to determine whether an insertion must be performed. From the analysis given above, the expected number of insertions is $D(\alpha)\alpha n$. The first data move is made immediately after the test, and so these instructions are executed $D(\alpha)\alpha n$ times. The remainder of the $E(\alpha)\alpha n$ data moves expected are made within a *while* loop, and so the expected number of iterations through the body of the loop is $(E(\alpha) - D(\alpha))\alpha n$. Adding these contributions together, the overall running time of the insertion phase may be expressed as $T_6 = (c_{6,d}\alpha D(\alpha) + c_{6,e}\alpha E(\alpha) + c_{6,n})\, n$.

3.3. Experimental Results

When an algorithm is actually implemented, there are several factors which can greatly influence its running time. Among these are the programming language, compiler, operating system and, of course, the hardware of the underlying computer. Our algorithm was implemented on several hardware platforms using either Pascal or C. In what follows, we shall describe in some detail the results obtained on a VAX-11/750 running VMS Version 4.7 with the algorithm coded in Pascal. We chose this machine for our experimental work because it was available for our use "stand-alone" so the influences of a multiprogramming environment could be eliminated. Since this machine is antiquated by today's standards, we will summarize the results obtained for a number of newer computers at the end of this section. The data obtained for the VAX-11/750 are particularly interesting due to one of its hardware features, an address translation buffer. Most modern computers have a multilevel memory hierarchy and some have memory management schemes that cause *BucketSort* to perform in a fashion that closely resembles its behavior on the VAX-11/750.

In the experiments, random integers over the range $(0, 10^9)$ were sorted; these are implemented

	$n\ (\times 10^3)$									
α	10	30	50	80	120	150	170	200	250	300
0.20	1.082	3.462	6.288	10.871	16.626	21.062	23.893	28.905	36.662	44.718
0.25	1.070	3.410	6.215	10.570	16.446	21.033	24.109	28.725	36.610	44.568
0.30	1.055	3.394	6.174	10.520	16.509	21.113	24.398	28.826	36.704	44.400
0.35	1.053	3.382	6.169	10.522	16.612	21.236	24.458	29.014	36.717	44.569
0.40	1.062	3.388	6.176	10.515	16.697	21.344	24.541	29.157	36.963	44.628
0.50	1.066	3.402	6.211	10.700	16.918	21.567	24.563	29.224	37.221	44.965
0.75	1.093	3.489	6.417	11.120	17.400	22.138	25.373	30.082	38.019	46.118
1.00	1.120	3.590	6.662	11.435	17.883	22.716	25.772	30.840	38.932	46.984

Table 3.2. Execution times (in seconds) for selected values of α and n.

on the VAX in four bytes. Each pointer also requires four bytes, so the overall space requirement is about $(8+4\alpha)n$ bytes. The CPU times were determined experimentally for each step. Values of n ranged from 10,000 to 300,000 in increments of 10,000, while α ranged from 0.2 to 1.0 in increments of 0.05. This range enabled accurate measurement of the CPU time to 0.01 seconds, while sufficient real memory was available to prevent page faults. The times for each (α, n) pair were obtained by averaging experimental results from five runs on each of five different input sets. It may be seen from the representative run times in Table 3.2 that the maximum and minimum times for each n vary by only 6% to 8%, indicating that the effect of α on run time is not appreciable. If one arbitrarily chose $\alpha = 0.2$ for all n, a space savings of slightly over 25% would result while the running time would be still be less than that for $\alpha = 1$.

Table 3.3 presents timings for each step of the algorithm with $\alpha = 0.6$ and n in the range [10,000, 100,000]. These are typical of those obtained with other values of α. For a fixed value of α, the running times of steps 1, 2 and 6 are proportional to n as predicted by the analysis in Section 3.2. However, the times for steps 3 through 5 display a departure from linearity due to the effect of multilevel memory causing the access times for data to vary. Thus, determining the constants for steps 1, 2 and 6 is relatively straightforward, but for steps 3 through 5 we will have to examine the memory structure of the machine more carefully. The constants c_1 and c_2 were ascertained by

	n ($\times 10^3$)									
Step	10	20	30	40	50	60	70	80	90	100
1	0.041	0.080	0.118	0.157	0.198	0.233	0.272	0.313	0.350	0.388
2	0.102	0.202	0.301	0.402	0.503	0.600	0.701	0.799	0.903	1.001
3	0.237	0.476	0.716	0.965	1.219	1.518	1.852	2.184	2.527	2.863
4	0.169	0.339	0.523	0.771	1.037	1.309	1.576	1.846	2.120	2.390
5	0.358	0.772	1.284	1.883	2.500	3.132	3.767	4.408	5.064	5.706
6	0.168	0.330	0.493	0.657	0.822	0.986	1.150	1.314	1.471	1.639
Total	1.074	2.199	3.434	4.834	6.280	7.729	9.318	10.864	12.434	13.986

Table 3.3. Execution times (in seconds) for each step, $\alpha = 0.6$.

taking an average of the run times over the entire range of n and α, normalized by dividing by αn and n, respectively. Thus, $T_1 = 6.505 \times 10^{-6}\alpha n$ and $T_2 = 10.02 \times 10^{-6} n$. The constants for step 6 were determined by performing a linear regression. For each value of α, the experimental times were averaged over the entire range of n to give

$$T_6 = (6.935\alpha D(\alpha) + 9.593\alpha E(\alpha) + 10.31) \times 10^{-6} n.$$

Effect of Multilevel Memory

To determine the coefficients c_3, c_4 and c_5, we must develop a cost model for data access time which accounts for the behavior of the memory management hardware in our experimental environment. The machine has a virtual to real address translation buffer to decrease the time required to access data from main memory [9]. The execution time for each data access may be modeled by the equation

$$t_h p_h + (1 - p_h) t_m = t_m - (t_m - t_h) p_h, \tag{3.2}$$

where p_h is the probability that the real address referenced is in the translation buffer, and t_h and t_m are the times in the case of a "buffer hit" and "buffer miss", respectively ($t_m \approx 2t_h$ on the VAX).

The running times of steps 3 through 5 are governed by three kinds of processing: 1) sequential array access, 2) random array access, and 3) other references (including scalar variables and array data already in the buffer). For the first and third types, p_h is close to one because references repeatedly access the same page. For the second type, suppose an array of size r is accessed randomly with a buffer addressing b words. When $r < b$, most of the array's page translations will fit in the buffer and p_h is again close to one. When $r > b$, most of the b locations in the buffer are occupied by translations to the array and p_h is about b/r. Since r is proportional to n in this algorithm, it can be shown from (3.2) that the contribution of random array references to the running time may be approximated by a linear function in n when $r > b$. A "break point" between these two distinct linear regions occurs when r is close to b.

The real memory access time varies according to whether the virtual to real address translation is resident in the translation buffer and, in the case of a memory read, whether the data value is resident in the memory cache. Because the memory cache is small (4K bytes), only the effect of the translation buffer can be seen in the data. The buffer holds 256 page translations which map to 256 $\times$ 128 = 32,768 four byte words since there are 128 words per page. The translation buffer creates the illusion of a cache that holds up to $b = 32{,}768$ words.

Bucket Distribution

In this step, the bucket head array of size $r = \alpha n$ is accessed randomly; the data and link arrays are accessed sequentially. The break point occurs at about $\alpha n = 30{,}000$. The running time of this step can be expressed as

$$T_3 = \begin{cases} 23.90 \times 10^{-6} n & \alpha n \leq 30{,}000 \\ 33.32 \times 10^{-6} n - \frac{0.2826}{\alpha} & \alpha n \geq 30{,}000. \end{cases}$$

Bucket Chaining

In this phase, the linked list of size $r = n$ is accessed randomly. Most random references occur at line 8 (see Appendix A) where the pointer holding the *current* position in the list is updated. Thus, $c_{4,n}$ is discontinuous while $c_{4,\alpha}$ and $c_{4,b}$ are not. A break point occurs at about $n = 30{,}000$. The run time of this step may be modeled by

$$T_4 = \begin{cases} (5.905\alpha + 13.10B(\alpha) + 7.214)n \times 10^{-6} & n \leq 30{,}000 \\ 0.2164 + ((5.905\alpha + 13.10B(\alpha))n + 16.87(n - 30{,}000)) \times 10^{-6} & n \geq 30{,}000. \end{cases}$$

The MacLaren Routine

In this step both the data and link arrays, each of size $r = n$, are accessed randomly. When $n \leq b/2$, the translations to both arrays fit in the buffer and p_h is close to one. For $b/2 < n < b$, the translations to the link array dominate the buffer because it is referenced twice as often as the data array. When $n > b$, the hit probabilities for both arrays are inversely proportional to n and decrease rapidly. The running time in each region may be approximated by a linear function. The break points occur at $n = 17{,}000$ and $37{,}000$. Thus we have

$$T_8 = \begin{cases} 35.75 \times 10^{-6} n & n \leq 17{,}000 \\ 0.6078 + 53.45 \times 10^{-6}(n - 17{,}000) & 17{,}000 \leq n \leq 37{,}000 \\ 1.677 + 64.83 \times 10^{-6}(n - 37{,}000) & n \geq 37{,}000. \end{cases}$$

Determination of Optimal Alpha

The equation for the overall running time, $T(n, \alpha)$, is obtained by adding the times for the individual steps. The optimal value of alpha, $\alpha_{opt}(n)$, minimizes $T(n, \alpha)$ for a fixed value of n. Because of the form taken by the run time for each step, we may write $T(n, \alpha) = \hat{T}_1(n) + \hat{T}_2(n, \alpha)$, where $\hat{T}_1$ is the portion of the time for all steps that does not vary with α and $\hat{T}_2$ is the portion that does. We seek the solution of

$$\frac{\partial T(n, \alpha)}{\partial \alpha} = \frac{\partial \hat{T}_2(n, \alpha)}{\partial \alpha} = 0. \tag{3.3}$$

$\hat{T}_2(n, \alpha)$ may be written as

$$\hat{T}_2(n, \alpha) = \begin{cases} f(\alpha) n \times 10^{-6} & \alpha n \leq 30{,}000 \\ f(\alpha) n \times 10^{-6} - \frac{0.2826}{\alpha} & \alpha n > 30{,}000, \end{cases} \tag{3.4}$$

where $f(\alpha) = (12.41 + 6.935 D(\alpha) + 9.593 E(\alpha))\alpha + 13.10 B(\alpha)$. Substituting (3.4) into (3.3), we have

$$0 = \begin{cases} f'(\alpha) & \alpha n \leq 30{,}000 \\ f'(\alpha) + \frac{0.2826 \times 10^{6}}{\alpha^2 n} & \alpha n > 30{,}000. \end{cases}$$

When $\alpha n \leq 30{,}000$, the optimal value of α is the solution to $f'(\alpha) = 0$ and is independent of n. Since $D(\alpha)$ does not have a closed form, numerical techniques were used to find that $\alpha_{opt} = 0.37$ in this range.

Figure 3.1 shows a plot of α_{opt} versus n. While $\frac{\partial \hat{T}_2(n, \alpha)}{\partial \alpha}$ is discontinuous at $\alpha n = 30{,}000$, α_{opt} is still equal to 0.37 until $n = \frac{30{,}000}{0.37} = 81{,}000$. As n increases above 81,000, the discontinuity causes

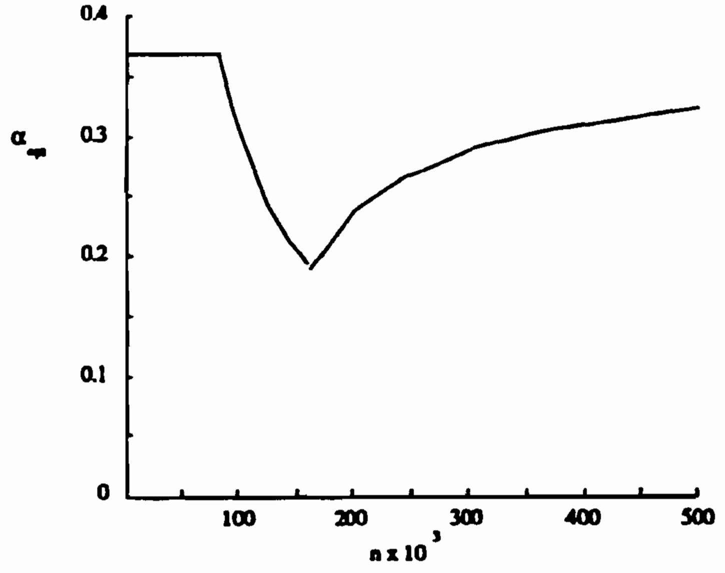


Figure 3.1. α_{opt} **for the VAX-11/750.**

$\frac{\partial \hat{T}_2(n,\alpha)}{\partial \alpha}$ to change sign until an n is reached such that $\frac{0.2826 \times 10^6}{\alpha^{*2} n} < |f'(\alpha^*)|$, where $\alpha^* = 30{,}000/n$ is the value of α at the point of discontinuity. This occurs at $n = 155{,}000$. For n between 81,000 and 155,000, $\alpha_{opt}(n)$ is $30{,}000/n$ and thus decreases with n. For $n > 155{,}000$, $\alpha_{opt}(n)$ increases slowly to approach once again a value of 0.37 asymptotically. The data access time does not vary for very large n since the probability of a buffer hit is close to zero and almost every random reference to memory accesses data not in the buffer. For α_{opt} to be within 10% of the asymptotic value of 0.37, our model predicts that n must be close to six million.

Results for Other Machines

The insertion sort step always dominates the overall running time when α is small while the bucket overhead (initialization and append steps) dominates when α is large. Thus, the graph of $T(n,\alpha)$ versus α is always convex when $\alpha \leq 1$ for a fixed value of n. This implies there is a unique $\alpha_{opt}(n)$ which minimizes the run time, although its value is machine dependent. Moreover, since

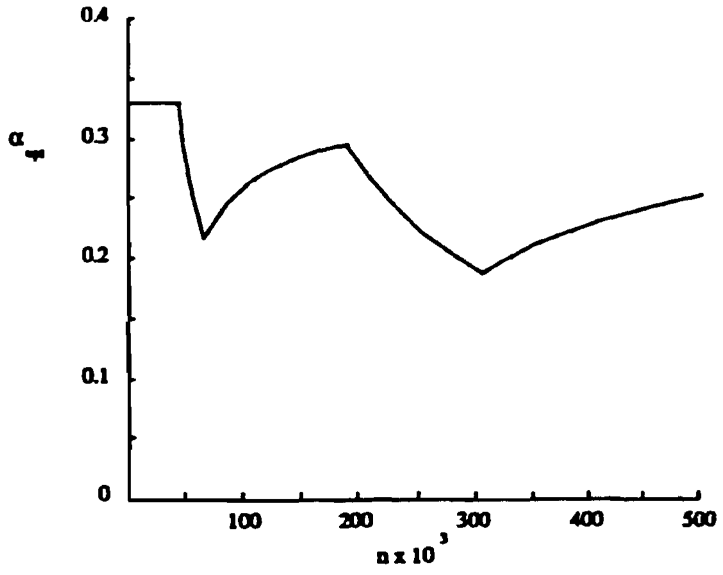


Figure 3.2. α_{opt} for the DECStation 5100/200, MIPS.

α_{opt} does not vary with n when the data access time is constant, any observed variation of $\alpha_{opt}(n)$ is due to transitory memory management effects peculiar to a specific machine. In such cases it is observed that $\alpha_{opt}(n)$ approaches an asymptotic value, α_{opt}, for both large and small n where the model of constant data access time applies.

Experiments were also conducted on several newer computers. The DECstation 5000/200 is a MIPS machine with an R3000 processor, a 64K byte data cache and an address translation buffer capable of addressing 64K words [39], twice that of the VAX-11/750. The curve obtained for $\alpha_{opt}(n)$ is shown in Figure 3.2. There are two dips caused by the cache and the translation buffer, respectively. The VAXstation 3100 has a 32K byte memory cache but no translation buffer. Its curve is similar in shape to that of the VAX-11/750 except that the dip is less pronounced and occurs at a smaller value of n (see Figure 3.3). Experiments were also conducted on a SUN 3/60, which has no cache or translation buffer. As predicted by the constant data access time model, the curve of T versus n is

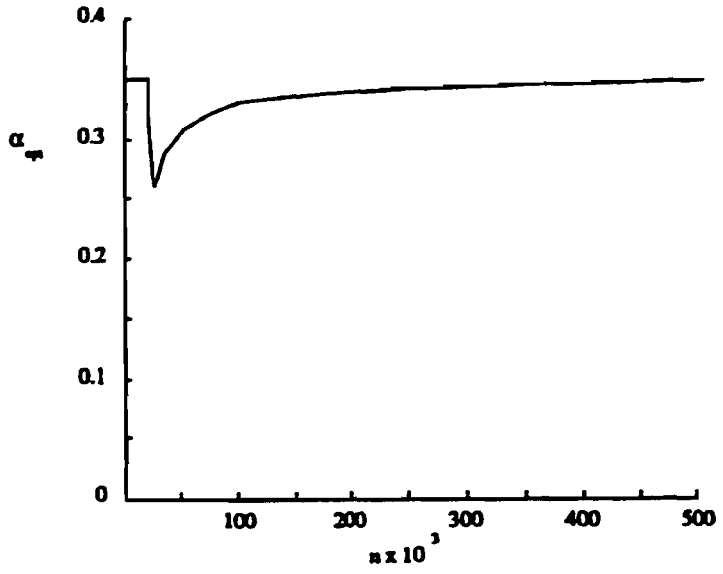


Figure 3.3. α_{opt} for the VAXStation 3100.

linear and $\alpha_{opt} = 0.39$ does not vary with n. The asymptotic value of α_{opt} for these four machines is confined to the narrow range [0.33, 0.39]. Two other machines were tested, the SUN 386i/150 and the SUN 4/60. While the lack of a controlled environment prevented an accurate determination of $T(n, \alpha)$, the average value of α_{opt} was found to be 0.36 ± 0.04 for the 386i/150 and 0.39 ± 0.02 for the 4/60.

3.4. Conclusions

Although it has been suggested that a number of buckets other than n be used in conjuction with variations of *DPS* [7,29], previous analyses have ignored the details associated with implementing the buckets as linked lists. Doing so, leads to the false conclusion that the running time will continue to decrease indefinitely as α increases. A more careful microanalysis of the algorithm using the model of constant data access time reveals that there is an optimal value of α, α_{opt}, which is machine dependent. The analysis can be further refined for machines with multilevel memory hierarchies to show that α_{opt} actually varies with n. For such machines, a constant data access time applies when n is small and is approached asymptotically when n is large. At values of n between these extremes, $\alpha_{opt}(n)$ is smaller than its limiting value due to the fact that the running time of the bucket distribution step, actually increases with α.

The value of α_{opt} does not vary greatly from machine to machine in the absence of transitory memory management effects. For the machines considered here, the asymptotic value of α_{opt} was in the narrow range [0.33, 0.39]. In fact, if unit weights are given to each data assignment and to each comparison and zero weights are given to all other instructions, the resulting α_{opt} is 0.36 — squarely in the middle of the observed range. This and the fact that the run time varies slowly with α indicates that $\alpha_{opt} = 0.36$ could be used with confidence for most machines.

Moreover, the execution time at $\alpha = 0.2$ was less than that for $\alpha = 1$ on all of the machines investigated at any fixed value of n. Choosing $\alpha = 0.2$ results in a space savings of over 25% while still reducing the execution time from its value at $\alpha = 1$, albeit by a small amount. If a greater space savings is desired, one could decrease α even further until a value for this parameter is found which yields a run time equal to that for $\alpha = 1$. On the DECstation 3100, a machine typical of those investigated, this value was found to be $\alpha = 0.13$, resulting in a space savings of approximately 30% over that for $\alpha = 1$.

CHAPTER 4

AN APPLICATION OF THE METHOD OF BUCKETS TO THE SELECTION PROBLEM

4.1. Introduction

There are many situations which call for selecting one or more ranked items including numerous statistical applications. One example is finding confidence intervals about the median. Another example from computer science is distributive sorting. Here, ranked data from a small sample can be used to divide the items to be sorted into approximately equal proportions.

The worst case performance of the selection problem is $\Theta(n)$ [1]. In 1961, Hoare published a selection algorithm called *Find* [17]. Although the performance of the algorithm is $O(n^2)$ in the worst case, it is shown in [1] to be at least five times faster on average than an efficient $O(n)$ worst case algorithm. In 1975, Floyd and Rivest published Algorithm *Select* [13] which, by proper choice of partitioning elements, achieves an even better, and close to optimal, expected case performance with repect to the number of comparisons. Allison and Noga first applied the method of buckets to the selection problem [2]. Their method was to distribute the items into $c\sqrt{n}$ buckets and to determine the bucket in which the desired item is located by counting the bucket cardinalities. Empirical results in [2] showed that *Select* was superior in all cases.

We present here an algorithm called *BucketSelect*, based on the partitioning strategy of *Select*, that applies the bucketing principle to the problem of selecting the kth smallest of n items. Since this problem is symmetrical to that of finding the kth largest, we will only consider the case where $k \leq n/2$. When only one ranked item is sought, the use of multiple buckets is not feasible because of the overhead incurred to maintain buckets as linked lists. Therefore, we use only one bucket which can be maintained in an array.

We will employ the notation used in [12]. Let X be a finite set of distinct elements which are ordered such that for $x_1, x_2 \in X$ either $x_1 < x_2$ or $x_2 < x_1$. We define $k\theta X$ to be the kth smallest element of X and $x\rho X$ to be the rank of x in X. Thus $(x\rho X)\theta X = x$. The number of items in the set X is denoted by $|X|$. As it will be clear from the context, we will also use X to stand for the array in which the set X is stored and X_k to refer to the set element occupying the kth position in the array.

BucketSelect is based on Floyd and Rivest's method [12] for choosing two members of X, u and v, such that $k\theta X$ satisfies $u \leq k\theta X \leq v$ with high probability. The elements are chosen by picking random samples, S, of size s from X and applying the selection algorithm recursively to each sample. Their method is to partition the dataset, X, into three subsets

$$\begin{aligned} A &= \{x \in X : x < u\} \\ B &= \{x \in X : u \leq x \leq v\} \\ C &= \{x \in X : x > v\} \qquad (4.1) \end{aligned}$$

The difference between our algorithm and Floyd-Rivest's *Select* algorithm is the way in which the partitioning is done. Both implementatious store the elements of X in an array of size $|X|$. In *Select* the partitioning is done by rearranging the array elements using Hoare's *Partition* algorithm [17]. In *BucketSelect* all elements of B are placed in an auxilliary array and the items in A and C are ignored. The kth smallest item is expected to be in set B with high probability. In the unlikely event that this item is not in B then *Select* is applied to X, otherwise *BucketSelect* is applied recursively to the set B. When n is small, neither of these algorithms should be used because of the overhead involved, and so small subsets are partitioned using the algorithm *Find* [17]. We will denote this cutoff value by n_c. The cost of incorporating *Partition* into *Select* to rearrange the array elements is considerable, especially when k is close to $n/2$. When $k = n/2$ the running time of *BucketSelect* decreases from 64% to 59% of that for *Select* when n increases from 50,000 to 250,000 items, with an asymptotic percentage of 53.6%.

First we discuss how u and v are selected to insure that $P(k\theta X \notin B)$ is small and, at the same time, $E(|B|) = o(n)$. Next we examine the optimal sample size, s, and the method of choosing the sample. We show that the implementation of *Select* requires a larger sample when $k < n/2$ than *BucketSelect* and the same sample size when $k = n/2$. We also note that the samples used to find u and v in *BucketSelect* are independent while those in *Select* are not. Finally we present an analysis of the two algorithms. We first derive equations for the expected running times of the partitioning phases for *BucketSelect* and *Select*, T_{B1} and T_{S1}. These equations involve time constants which are implementation dependent. The constants are measured for a typical implementation with the result that $T_{S1} > T_{B1}$ for all k and n, with the maximum difference occurring when $k = n/2$. We then derive equations which neglect only times that are $o(s)$, where s is the sample size, and show that the $O(s)$ terms which are included constitute a significant part of the running time for both algorithms when n is as high as 250,000.

4.2. Method of Selecting u and v

In order to determine the partitioning elements u and v, a sample S of size s is selected from X. The probability that $i\theta S = k\theta X$ is given by

$$P(i\theta S = k\theta X) = \frac{\binom{k-1}{i-1}\binom{n-k}{s-i}}{\binom{n}{s}} \tag{4.2}$$

since there are $\binom{k-1}{i-1}$ ways to choose the $i-1$ items that are less than $i\theta S$ from the $k-1$ items that are less than $k\theta X$, $\binom{n-k}{s-i}$ ways to choose the $s-i$ items that are greater than $i\theta S$ from the $n-k$ items that are greater that $k\theta X$, and there are $\binom{n}{s}$ possible samples with each possible ordering considered equally likely to occur. The probability density defined by (4.2) is known as a hypergeometic distribution and corresponds to the classical probability model of sampling without replacement. When n is large compared to s, (4.2) approaches a binomial distribution which corresponds to sampling with replacement. Using this model, we can take $p = k/n$ to be the probability that any $x \in S$ is less than or equal to $k\theta X$. The probability that any particular group of i items in S is less than or equal to $k\theta X$ while the rest are greater is $p^i(1-p)^{s-i}$. Since there are $\binom{s}{i}$ possible choices for the i items, the probability that exactly i items are less than or equal to $k\theta X$ is given by the binomial density $b_{s,p}(i) = \binom{s}{i} p^i (1-p)^{s-i}$, with mean sp and standard deviation $\sigma = \sqrt{sp(1-p)}$.

To insure that $u \leq k\theta X \leq v$ with high probability, we wish to find a greatest integer i_1 and a smallest integer i_2 such that $u = i_1\theta S$, $v = i_2\theta S$ and

$$P(i_1\theta S > k\theta X) = \epsilon_1 \text{ and } P(i_2\theta S < k\theta X) = \epsilon_2,$$

$$\epsilon_1, \epsilon_2 \ll 1.$$

We note that $E(upX) = i_1\dfrac{n+1}{s+1} \approx i_1 n/s$, $E(vpX) \approx i_2 n/s$ and $E(|B|) \approx (i_2 - i_1)n/s$. $P(i_1\theta S > k\theta X)$ equals the probability that at most $i_1 - 1$ items are less than $k\theta X$ and is given by

$$\epsilon_1 = P(i_1\theta S > k\theta X) = \sum_{j=0}^{i_1-1} b_{s,p}(j) = B_{s,p}(i_1 - 1) \tag{4.3}$$

where $B_{s,p}(i)$ is the cumulative binomial density. Similarly, $P(i_2\theta S < k\theta X)$ equals the probability that at least i_2 items are less than $k\theta X$. Since $P(i_2\theta S = k\theta X)$ is very small, a close approximation to this probability is given by

$$\epsilon_2 = P(i_2\theta S < k\theta X) = \sum_{j=i_2}^{s} b_{s,p}(j) = 1 - B_{s,p}(i_2 - 1). \tag{4.4}$$

Thus the probability ϵ that $k\theta X$ does not lie between u and v is given by

$$\epsilon = \epsilon_1 + \epsilon_2 = B_{s,p}(i_1 - 1) + 1 - B_{s,p}(i_2 - 1). \tag{4.5}$$

It is well known [26] that the binomial density approaches the normal asymptotically provided $p = \Theta(s)$. Hence for large s, $B_{s,p}(i) \approx \Phi_{sp,\sigma}(i)$, where Φ is the cumulative normal density. If we choose a factor d such that i_1 and i_2 are d standard deviations to the left and right of the mean (i.e., $i_1 = \lfloor sp - d\sigma \rfloor$ and $i_2 = \lceil sp + d\sigma \rceil$) then (4.5) becomes

$$\epsilon \approx 2(1 - \Phi_{sp,\sigma}(sp + d\sigma)) = 2(1 - \Phi_{0,1}(d)). \tag{4.6}$$

Under the assumption that (4.6) holds, $E(|A|) = k - d\sigma n/s$, $E(|B|) = 2d\sigma n/s$ and $E(|C|) = n - k - d\sigma n/s$. If we let $\bar{a} = E(|A|), \bar{b} = E(|B|)$ and $\bar{c} = E(|C|)$ then $\bar{a} = k - \frac{\bar{b}}{2}$ and $\bar{c} = n - k - \frac{\bar{b}}{2}$. We choose d to be a slowly growing function of n such that as n gets large $\bar{b}$ and ϵ are small. Letting

$d = \sqrt{2}(2 + \ln n/n_\epsilon)^{1/2}$ insures that $\epsilon = o(\frac{1}{n})$ and is sufficiently small for fairly small values of n ($d \geq 2$).

When p is small, it is possible for i_1 to be less than 1. This implies that there is no $u \in S$ such that (4.3) holds for ϵ_1 sufficiently small. In this case we take $u = -\infty$, implying $A = \emptyset$ and $\epsilon_1 = 0$. Similarly, when p is close to 1, i_2 could be greater than s implying that there is no $v \in S$ such that (4.4) holds for ϵ_2 sufficiently small. We then take $v = +\infty$ implying that $C = \emptyset$ and $\epsilon_2 = 0$.

4.3. Choice of Sample

Sample Size

For both algorithms the choice of sample size, s, involves the following considerations. First, s should be optimized with respect to $\bar{b}$. Since the work done by the recursive calls in *BucketSelect* and *Select* grows linearly with both s and $\bar{b}$, we wish to pick s such that the respective growth rates are the same. For *BucketSelect* we define

$$g_B(n,p) = \left(\frac{p(1-p)}{2}\right)^{1/3} n^{2/3}(2 + \ln(n/n_\epsilon))^{1/3}.[1]$$

Then

$$s = \kappa g_B(n,p) \tag{4.7}$$

and

$$\bar{b} = \frac{4}{\sqrt{\kappa}} g_B(n,p) \tag{4.8}$$

The choice of κ requires knowledge of the time constants for the implementation and will be discussed in the next section.

The second consideration involved in choosing s is that it be large enough for the normal approximation to the binomial to apply. As noted in previously, there is a cutoff value, n_ϵ, below which neither *BucketSelect* nor *Select* will be used. An exact determination of n_ϵ is extremely difficult

[1] The factor $(2 + \ln(n/n_\epsilon))^{1/3}$ insures that $\epsilon = o(\frac{1}{n})$. Floyd and Rivest incorrectly used $\ln^{1/3} n$ in their analysis [12] and neglected it entirely in the computation of s in [13].

and involves the expense of computing s, σ and d, the running time of *Find* and the depth of the recursion (which is a function of the initial n). Any value between 500 and 2,000 could be used without significantly affecting the results of either algorithm, and so we shall choose $n_c = 1{,}000$ for both algorithms. If we let $n = n_c = 1{,}000$ and $\kappa = 1$ in (4.7), we get $s = \lceil 100(p(1-p))^{1/3} \rceil$ since s must be a positive integer. When $p = 0.5$, $s = 63$; when $p = 1/n_c = 0.001$, its smallest possible value, $s = 10$. For $n = 1{,}000$ the largest value of $P(k\theta X \notin B)$ occurs when we are forced to pick $i_1 = 0$ and $i_2 = 1$. The largest probability consistent with this choice is $p = 0.008$, yielding $P(k\theta X \notin B) = 0.148404$. Although this probability is fairly high, the consequences of a "bad split" are minimal for this n because the running time of *BucketSelect*, *Select*, and *Find* are approximately the same. The probability drops sharply with increasing p. For instance when $p \geq 0.02$, $P(k\theta X \notin B) \leq 0.101079$ and when $p \geq 0.03$, $P(k\theta X \notin B) \leq 0.075902$. When $p = 0.5$, $P(k\theta X \notin B)$ falls to 0.032766. Although it cannot be proven that $P(k\theta X)$ is smaller than 0.148404 for all p when $n > n_c$, experimental evidence indicates that the methods work well for larger n even if $p = 1/n$.

Choice of Sample Elements

In *Select* the partitioning is done in two stages. At each stage s contiguous items surrounding X_k are selected from the array X. Thus, most of the sample elements from the determination of u are reused in choosing v. In certain cases this can result in a "bad split" (i.e., $k\theta X \notin B$), especially when s is small. The formula for σ that is used in the Floyd-Rivest analysis, $\sigma = \frac{1}{2}\sqrt{\frac{s(n-s)}{n}}$, somewhat alleviates the problem of bad splits since it results in the use of a larger sample size than actually required. The factor of $1/2$ comes from taking $p(1-p)$ equal to its maximum value of $1/4$. The $\sqrt{\frac{n-s}{n}}$ factor arises from the exact formula for the standard deviation derived from the hypergeometric distribution (4.2). Retaining this factor is not consistent with the use of the normal approximation but its effect is small since it is always close to one ($n \gg s$).

In the implementation of *BucketSelect*, the samples used to pick u and v are disjoint. For convenience we pick the first sample to be the first S elements of X and the second to be the next s

```
$Select(X[\,], k, n)$
{ if $(n > n_c)$
  { compute $s$, $\sigma$, $d$ and $S$
    if $(k \leq n/2)$
      compute $i = i_2$;
    else
      compute $i = i_1$;
    $i\theta S = Select(S, i, s)$;
    determine $A = \{x : x < i\theta X\}, B = \{i\theta S\}$,
              $C = \{x : x > i\theta X\}$,
              $a = |A|$ and $c = |C|$;
    if $(i\theta S = k\theta X)$
      return $(i\theta S)$
    else
      if $(i\theta S \in A)$
        return $(Select(A, k, a))$;
      else
        return $(Select(C, k - a - 1, c))$;
  else
    return $(Find(X, k, n))$; }
```

Figure 4.1. Algorithm *Select*.

```
$BucketSelect(X[\,], k, n)$
{ $B[\,]$;
    if $(n > n_c)$
    { compute $s, \sigma, d, i_1, i_2$;
      if $(i_1 \geq 1)$
        $u$ = $BucketSelect(A, i_1, s)$;
      else
        $u$ = $-\infty$;
      if $(i_2 \leq s)$
        $v$ = $BucketSelect(A + s, i_2, s)$;
      else
        $v = +\infty$;
      determine $B = \{x : u \leq x \leq v\}$,
                $a = |A|$ and $b = |B|$;
      if $(a < k \leq a + b)$
        return $(BucketSelect(B, k - a, b))$;
      else
        return $(Select(X, k, n))$;  }
    else
      return $(Find(X, k, n))$}
```

Figure 4.2. Algorithm *BucketSelect*.

elements. Our method retains the more exact formula for σ and the consequent reduction in sample size from that required to determine the median.

4.4. Analysis of the Algorithms

Figure 4.1 describes the *Select* algorithm as it is given in [13] using the notation introduced earlier. Items are split into the sets A, B and C as defined by (4.1) in two stages. When $k \leq \frac{n}{2}$, the first stage determines $A \bigcup B$ and C, while the second splits $A \bigcup B$ into A and B. The algorithm uses $\frac{(k-1)(m-k)}{(m-1)}$ exchanges on the average to determine each partition, where m is the cardinality of the set being split. This is true because each of the $k-1$ elements to the left of $k\theta X$ has an equal probability of having any rank in X other than k and, therefore, a probability of $\frac{m-k}{m-1}$ of having a rank bigger than k and thus of being swapped.

Figure 4.2 describes the *BucketSelect* algorithm. Considering only the case where $1 \leq i_1 < i_2 \leq s$, $k > d\sigma n/s$ and $n > n_c$, the following recurrence may be written for the running time of both *Select* and *BucketSelect*:

$$\dot{T}(n,k) = \dot{T}_1(n,k) + \dot{T}(s,i_1) + \dot{T}(s,i_2) + \dot{T}(b,k-a),$$

where $\dot{T}_1$ is the time required to do the partitioning, $\dot{T}(s,i_1)$ and $\dot{T}(s,i_2)$ are the times to find the two pivoting elements, and $\dot{T}(n,k-a)$ is the time for locating the desired element in the set B by applying the algorithm recursively. If we replace the parameters in the preceding equation by their expected values and note that $E(k-a) = k - E(a) = {}^1\!/\!_2\bar{b}$, we get

$$\dot{T}(n,k) = \dot{T}_1(n,k) + 2\dot{T}(s, ps - d\sigma) + \dot{T}(\bar{b}, {}^1\!/\!_2\bar{b}).$$

Since $d\sigma$ is small compared to s, we may alternatively express the running time T entirely in terms of n and $p = k/n$ as

$$T(n,p) = T_1(n,p) + 2T(s,p) + T(\bar{b}, {}^1\!/\!_2). \tag{4.9}$$

The primary difference in the running times of the two algorithms is the nature of T_1, although the lower order terms in *BucketSelect* are also smaller due to the less conservative formula for s.

We will now investigate the form of T_1 for each algorithm. Figures 4.3 and 4.4 show the C programming language code for the partitioning steps of *BucketSelect* and *Select* along with the expected number of times that each line is executed. The partitioning loop of *BucketSelect* executes n times with a single data assignment executed $\bar{b}_B = o(n)$ times.[2] Since there are two steps in the

[2] Because $\bar{b}$ is different in *BucketSelect* and *BucketSelect*, we will use the notation $\bar{b}_B$ and $\bar{b}_S$ to distinguish between them.

```
a = 0;                                   1
pA = A;   pC = C;                        n
for ( i = l; i ≤ n; ++i; )               n
   if (*pA > v) ;                        n
   else
      if (*pA ≥ u)                       $k + \frac{b}{2}$
         *(++pC) = *pA                   $b$
      else
         ++a;                            $k - \frac{b}{2}$
b = pC - C;                              1
```

Figure 4.3. Partitioning step of *BucketSelect*.

```
i = l;   j = r;                          1
while (i < j)                            $\alpha + 1$
   { swap (A_i, A_j) ;                   $\alpha$
      ++i; --j;
      while (A_i < v)                    $\beta + \alpha$
         ++i;                            $\beta$
      while (A_j > v)                    $\gamma + \alpha$
         --j;  }                         $\gamma$
```

Figure 4.4. Partitioning step of *Select*.

partitioning process of *Select*, the inner *while* loops execute a total of $n + k$ times while the swap, which has a cost almost three times that of the data assignment in *BucketSelect*, is executed $O(n)$ times asymptotically. We will now derive T_1 for the two algorithms.

BucketSelect

The running time of the partitioning step in *BucketSelect* may be described by the equation

$$T_{B1}(n,p) = (e_1 + pc_2)n + c_3\bar{b}_B \qquad (4.10)$$

The constants e_1, c_2 and c_3 can be measured for a particular implementation by performing a series of experiments with various values of n and p. Each experiment is performed twice on the same random data. In one experiment the actual CPU time required to execute the loop is measured, while in the other the actual execution frequencies are measured. A linear regression is then performed to determine the constants.

The algorithm was programmed in C and implemented on a VAXstation 3100 running VMS Version 5.3. For the sake of efficiency, the algorithm was coded without the explicit use of recursion.

Twenty five sets of random data were used for each n and k, with n ranging from 10,000 to 100,000 in increments of 10,000. The following values of p were used: $1/n$, 0.01, 0.02, 0.1, 0.25 and 0.5.

For small values of n and p, the assumptions that give rise to the expected execution frequencies in Figure 4.3 do not hold. However, since actual rather that expected execution counts are used in the regression, these data points do contribute to the accurate determination of the constants. For the same reason, any convenient sample size may be used, so we chose $\kappa = 1$ in (4.7). The constants obtained for this implementation are: $c_1 = 2.72933 \times 10^{-6}$, $c_2 = 1.23664 \times 10^{-6}$, $c_3 = 1.69048 \times 10^{-6}$.

Select

The code in Figure 4.4 is repeated twice. When $k \le n/2$, $l = 1$ for both stages. In the first stage $r = n$ while in the second it has an expected value of $k + {}^1\!/\!_2\bar{b}_S$. Since the time constants for the steps with counts β and γ are the same and since $2\alpha + \beta + \gamma = r$, the running time may be expressed by the equation $T = c_4 r + c_5 \alpha$. Performing a similar set of experiments for an implementation of this algorithm, also coded without the explicit use of recursion, we obtained $c_4 = 3,00348 \times 10^{-6}$ and $c_5 = 4.65353 \times 10^{-6}$. The expected value of α for the first stage is $p(1-p)n$, and for the second stage it is $\dfrac{k\bar{b}_S}{2k + \bar{b}_S} \approx {}^1\!/\!_2\bar{b}_S$ when $k \gg \bar{b}_S$. Combining the two stages we get

$$\begin{aligned} T_{S1} &= c_4\left((1+p)n + {}^1\!/\!_2\bar{b}_S\right) + c_5\left(p(1-p)n + {}^1\!/\!_2\bar{b}_S\right) \\ &= \left(c_4 + (c_4 + c_5)p - c_5 p^2\right) n + \frac{c_4 + c_5}{2}\bar{b}_S. \end{aligned} \tag{4.11}$$

4.5. Timing Results

Substituting the values of the constants into (4.10) and (4.11), the running times for the partitioning steps of the two algorithms may be expressed in microseconds as

$$\begin{aligned} T_{B1} &= (8.949 + 2.669p)n + 5.113\bar{b}_B, \\ T_{S1} &= (10.049 + 26.439p - 16.390p^2)n + 13.220\bar{b}_S. \end{aligned} \tag{4.12}$$

As a result $T_{S1} > T_{B1}$ for all n and p with the maximum difference occuring when $p = {}^1\!/\!_2$.

Assuming $s > n_c$ we may obtain expressions for the running times of the two algorithms which include terms that are $O(s)$ and neglect only those terms that are $o(s)$. Substituting (4.10) into (4.9) we have

$$T_B(n,p) = (c_1 + pc_2)n + c_3\bar{b}_B + 2T_B(s,p) + T_B(\bar{b}_B, 1/2). \tag{4.13}$$

Substituting the expressions in (4.7) and (4.8) for s and $\bar{b}_B$ and ignoring terms that are $o(s)$, we obtain

$$\begin{aligned} T_B(n,p) &= (c_1 + pc_2)n \\ &+ \left((c_1 + 1/2c_2 + c_3)\frac{4}{\sqrt{\kappa}} + 2(c_1 + pc_2)\kappa\right) g_B(n,p), \end{aligned} \tag{4.14}$$

which is maximized by choosing

$$\kappa = \left(\frac{c_1 + 1/2c_2 + c_3}{c_1 + pc_2}\right)^{2/3}.$$

Finally substituting this value into (4.14), we find

$$\begin{aligned} T_B(n,p) &= (c_1 + pc_2)n \\ &+ 6(c_1 + pc_2)^{1/3}(c_1 + 1/2c_2 + c_3)^{2/3} g_B(n,p). \end{aligned} \tag{4.15}$$

If we take $g_S(n) = n^{2/3}(2 + ln(n/n_c))^{1/3}$ and $s = 1/2g_S(n)$ then $\bar{b}_S = 4\sqrt{p(1-p)}g_S(n)$. Replacing $\bar{b}_S$ by this value in (4.11) and substituting the resulting expression in (4.9) we have

$$\begin{aligned} T_S(n,p) &= (c_4(1+p) + c_5p(1-p))\,n \\ &+ ((8c_4 + 3c_5)\sqrt{p(1-p)} \\ &+ c_4(1+p) + c_5p(1-p))g_S(n). \end{aligned} \tag{4.16}$$

Formulas (4.15) and (4.16) apply well when $n \geq 30{,}000$ and p is not too small. The neglected $o(s)$ terms do not vary greatly with p when $p > 0.1$. At $n = 30{,}000$ the neglected terms in (4.15) account for about 16% of the running time. This percentage decreases to about 7% at $n = 250{,}000$. In (4.16) the corresponding percentages are 10% at $n = 30{,}000$ and 4% at $n = 250{,}000$. The asymptotic ratio of the running times of the two algorithms may be calculated by considering only the coefficients of n in (4.15) and (4.16). When $p = 1/2$ the ratio is 0.536. The largest ratio occurs when $p = 1/n$, where the ratio approaches $c_1/c_4 = 0.89$.

n	k	*Bucket Select*	*Select*	%
250,000	125,000	1.006	1.682	59.8
250,000	62,500	0.913	1.371	66.6
250,000	25,000	0.824	1.109	74.3
200,000	100,000	0.814	1.366	59.3
200,000	50,000	0.735	1.122	65.5
200,000	20,000	0.669	0.903	74.1
150,000	75,000	0.621	1.025	60.6
150,000	37,500	0.562	0.847	66.6
150,000	15,000	0.507	0.688	73.7
100,000	50,000	0.430	0.700	61.4
100,000	25,000	0.388	0.574	67.6
100,000	10,000	0.355	0.467	76.0
50,000	25,000	0.232	0.363	63.9
50,000	12,500	0.206	0.304	67.8
50,000	5,000	0.186	0.250	74.4

% = *BucketSelect*/*Select* expressed as a percentage

Table 4.1. Average execution times (in seconds) for *BucketSelect* and *Select*.

We note that the usual asymptotic analysis would neglect all $o(n)$ terms and approximate T by T_1. For large n such as 250,000, T_{B1} accounts for only 83.2% of the running time of *BucketSelect* with about 10% coming from the $O(s)$ terms in (4.15). In *Select*, T_{S1} accounts for 87.8% of the running time with an additional 8% contributed by the $O(s)$ terms in (4.16).

The timing results for the two algorithms are shown in Table 4.1 with each time representing the mean of 25 runs on different data sets. The standard deviation was low for each reported time, indicating that $P(k\theta X \notin B)$ is small, as expected. Although the exact results are system dependent, *BucketSelect* will outperform *Select* on all machines because, although both algorithms make $n + k$ comparisons asymptotically, the data moves in *Select* are $O(n)$ when $k = \Theta(n)$ while those of *BucketSelect* are $o(n)$ for all k. A consequence of using the auxillary array is that some extra storage is required. However the extra storage requirement is $o(n)$. When $n = 250{,}000$, *BucketSelect* requires only about 6% more storage than *Select*. In addition, since the array accesses are sequential, excessive page faults are not generated on a virtual memory system, as is the case when bucketing algorithms are implemented with linked lists.

4.6. Conclusion

In this paper we have presented a unique application of the bucketing principle to the problem of selecting a single ranked item from a dataset. It is unique in the sense that only one bucket is used. Algorithms using multiple buckets to select one item do not compare favorably with the fastest known algorithm, *Select*, because there is too much overhead required to maintain the buckets.

Our algorithm, *BucketSelect*, outperforms *Select* because it eliminates the expense of rearranging the data elements. The data expected to be candidates for the required ranked item are copied to an auxilliary array, or bucket. Since the probability of not placing the desired item in the bucket is very small, the expected performance of *BucketSelect* is governed only by the comparisons required to determine bucket membership and the expense of copying their values. We have shown that the exchanges required by *Select* are $O(n)$ while the copying of data values in *BucketSelect* is $o(n)$. We have also shown that it is possible to use a smaller sample size than used in the implementation of *Select*, thereby reducing the work done by the recursive calls. Finally, we have shown that the usual asymptotic analysis neglects a significant portion of the running time of both algorithms even when n is quite large.

CHAPTER 5

ERROR FREE INCREMENTAL CONSTRUCTION OF VORONOI DIAGRAMS IN THE PLANE

5.1. Introduction

In this chapter we present an efficient, error-free method of constructing the Voronoi diagram in the plane. A Voronoi diagram is defined as follows: Suppose that there exist n points in the plane called generators. The Voronoi polygon P_i of point p_i is defined to be the set of all points that are closer to p_i than to any other generator. The planar graph which consists of all points that are equidistant to two or more generators is called the Voronoi diagram. An example of a Voronoi diagram is given in Figure 5.1. The vertices and edges of the graph are called Voronoi points and Voronoi edges, respectively. The graph is also known in the literature by the names Dirichlet tesselation and Thiessen tessellation. This diagram plays an important role in computational geometry and has applications in several fields [37,34]. An example is the problem of determining the location of public facilities in such a way that the cost of inhabitants gaining access to the nearest facility is minimized [3]. The solution of this problem requires the repeated construction of Voronoi diagrams and is only feasible if this can be done efficiently.

Several algorithms based on the principle of divide and conquer can construct the diagram in $O(n \log n)$ time in the worst case, which is optimal in the decision tree model of computation [37,19]. Bentley, Weide and Yao were the first to show that it is possible to construct an $O(n)$ expected time algorithm [4]. However, their algorithm is quite complex and, to the best of our knowledge, has never been implemented. In 1984, Ohya, Iri and Murota published an $O(n)$ expected time algorithm [32,33] which is based on the incremental method of Green and Sibson [15]. The problem with algorithms based on the incremental method is that they are sensitive to numerical errors which

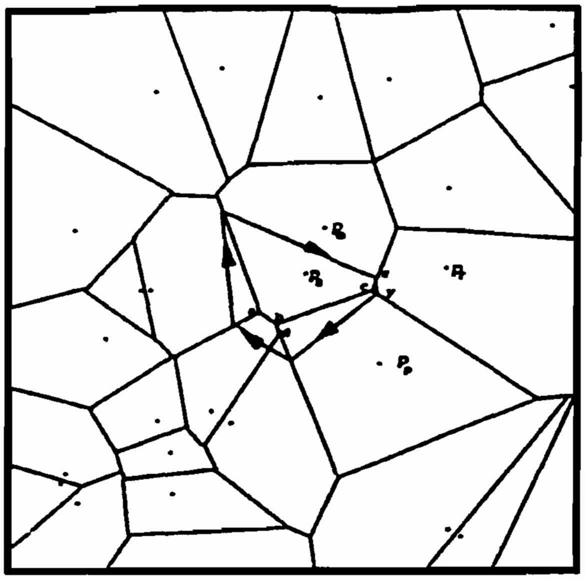


Figure 5.1. The addition of a new generator, p_n, to a Voronoi diagram. Construction starts in polygon P_m and proceeds in the order indicated by the arrows. The generators are indicated by the dots.

may prevent them from terminating despite the fact that they may be otherwise correct.

In 1989, Sugihara and Iri published an implementation of the incremental method which, although guaranteed to terminate regardless of the computational precision, sacrifices certain properties of the diagram [38]. For instance, although their construction insures that there are as many Voronoi polygons as points, it is possible to produce polygons with no generator inside them.

In this chapter we present an algorithm which, by placing restrictions on the allowable values of the generator coordinates, always constructs a correct Voronoi diagram. Correctness is insured by using integer arithmetic, instead of floating point computation which can introduce numerical errors. We will show that our method requires considerably fewer multiplications to add a generator than does Sugihara and Iri's algorithm and also greatly reduces the number of additions and subtractions required.

5.2. The Incremental Method

The incremental method starts with a trivial diagram for three generators. It then processes the remaining generators one at a time, adding and deleting lines to the diagram as each new generator is considered. The process of adding a generator is illustrated in Figure 5.1. First, the nearest neighbor generator, p_m, of the new point p_n is found and the perpendicular bisector of the line between them is erected. The bisector, which is represented by a heavy line in Figure 5.1, is given a direction such that p_n is on the right. The front of the bisector intersects an edge of P_m at x. At this point it enters the neighboring polygon P_r. Next the bisector of the line $\overline{p_n p_r}$ is erected and the intersection y with the front of this bisector and an edge of P_r is located. In this manner, a sequence of bisectors is constructed until a cycle is completed by returning to polygon P_m. Finally the points and edges enclosed by this sequence of bisectors are removed and the cycle just constructed and its interior becomes the polygon of the new generator.

This construction assumes that all neighboring polygons are closed. Green and Sibson introduced a method of dealing with unbounded polygons. However, the complexity of the method far outweighs its benefits. Asano et. al. have developed a simple technique to avoid dealing with open polygons [3]. They start with three pseudo generators that are known to be outside the convex hull of the actual generators. The diagram for these three generators alone consists of one real Voronoi point and three semi-infinite rays. Each additional bisector must intersect either another bisector or one of the three semi-infinite rays. Thus all polygons, except for those of the three pseudo generators, are closed. Figure 5.2 depicts a wider view of the same Voronoi diagram as Figure 5.1 showing the three pseudo generators.

Time Complexity

Bentley, Weide and Yao give the conditions for which the nearest neighbor of a new generator can be found in constant time [4]. These conditions are: 1) the generators be chosen independently from the same distribution over a bounded, convex, open region in the plane, and 2) the probability of a point being assigned to a region of area A lies between c_1A and c_2A, where c_1 and c_2 are

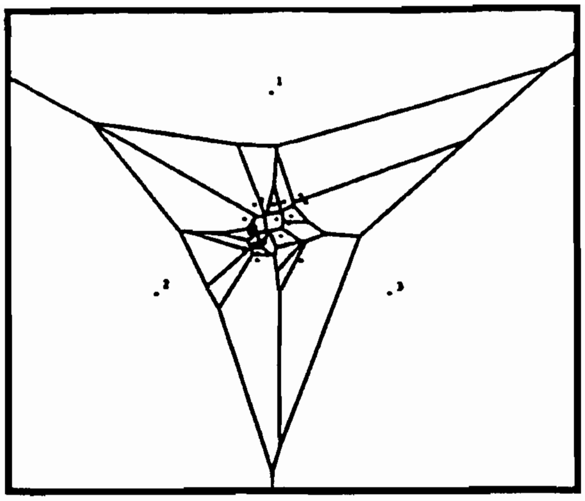


Figure 5.2. A wide view of the Voronoi diagram of Figure 5.1 showing the three pseudo generators p_1, p_2 and p_3.

constants satisfying the relation $0 < c_1 \leq c_2$.

Ohya, Iri and Murota introduced a canonical method of adding the generators. They showed that their algorithm finds the nearest neighbor of the new generator in constant time, provided that the number of edges in any intermediate diagram is bounded above by a constant. This is true for a wide range of distributions for the generator coordinates. They also showed that under these conditions the remaining steps in adding a new generator are $O(1)$ [33]. Intuitively, the generators that are already in the diagram should be distributed approximately uniformly around the new generator. In this paper, we will assume that the data distribution is sufficiently well-behaved that the expected time to add a new generator is $O(1)$ and we will concentrate on reducing the time constant.

Degeneracies

A degeneracy takes place when the perpendicular bisector of the line between the new generator and an old generator intersects a Voronoi point. This can happen when the new generator lies on a circle containing three existing generators. In this case, the bisectors of the four generators meet

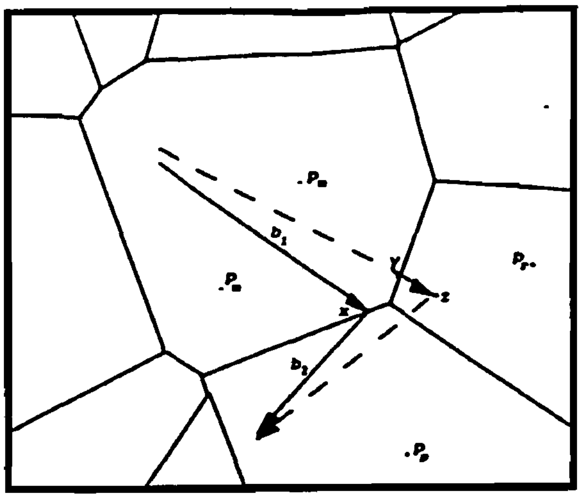


Figure 5.3. The effect of numerical errors on the constuction of the Voronoi diagram. The new vertex is actually located at x but, because of numerical error, may be incorrectly located at y or z.

at a common point which, if it is a Voronoi point, will be degenerate. Numerical errors can cause the incremental algorithm not to terminate when this bisector passes through, or close to, a Voronoi point. In Figure 5.3, the bisector of $\overline{p_m p_n}$ intersects polygon P_m at point x so the construction should continue in polygon P_p. Suppose, however, the intersection point were incorrectly computed as y. Then an intersection of the bisector of $\overline{p_r p_n}$ with polygon P_r, which in this example does not exist, would be sought. The avoidance of such errors is of fundamental importance in the implementation of the incremental method.

5.3. The Algorithm of Sugihara and Iri

A pseudocoded version of the algorithm used by Sugihara and Iri [38] to construct the Voronoi diagram is given in Appendix B. First the nearest existing generator, p_m, of the new generator p_n is found. A quarternary tree data structure enables this to be done in constant time [33]. Next, each vertex of the polygon P_m associated with p_m is checked to determine if it lies inside or outside the new polygon P_n. In Figure 5.1, the Voronoi points labeled a, b and c would be determined to be inside P_n. This judgement is made by evaluating the determinant

$$H(p_i,p_j,p_k,p_n) = \begin{vmatrix} 1 & x_i & y_i & (x_i^2+y_i^2)/2 \\ 1 & x_j & y_j & (x_j^2+y_j^2)/2 \\ 1 & x_k & y_k & (x_k^2+y_k^2)/2 \\ 1 & x_n & y_n & (x_n^2+y_n^2)/2 \end{vmatrix}, \tag{5.1}$$

where p_i, p_j and p_k are the old generators that surround the vertex in counterclockwise order. For example, for vertex c in Figure 5.1 we could take $i = p, j = r$ and $k = m$ (any permutation of p, r and m which maintains the counterclockwise order would yield the same result). If $H < 0$ (resp. $H > 0$), the vertex is judged to be inside (resp. outside) P_n. If $H = 0$, an edge of P_n passes through a vertex of P_m and a degeneracy occurs. The Sugihara and Iri algorithm does not check for $H = 0$, so in this case a vertex would be judged to be outside P_n. These authors denote by T the set of vertices of P_m that are inside P_n. Then they augment T by first calculating H at each vertex which is on the opposite end of an edge connecting an element of T with a vertex not in T. Then, if the computed H is negative, the new vertex is added to T. In Figure 5.1, vertex d is added to T by this process. Next, the new Voronoi points, which are the vertices of P_n, are located on the edges which connect a vertex of T with one that is not in T. This computation is done as follows: Let $H^{(s)}(i,j,k), s = 2,3,4$, be the matrix obtained by deleting the fourth row and the sth column of H and denote by $q(i,j,k)$ the Voronoi point surrounded in counterclockwise order by polygons P_i, P_j and P_k. Then the coordinates of $q(i,j,k)$ are given by

$$q_x = -H^{(2)}/H^{(4)} \quad \text{and} \quad q_y = H^{(3)}/H^{(4)}. \tag{5.2}$$

By subtracting the first row of H from the second and third rows, $H^{(2)}, H^{(3)}$ and $H^{(4)}$ may be written in the following form:

$$H^{(2)}(i,j,k) = \begin{vmatrix} y_j - y_i & ((x_j+x_i)(x_j-x_i)+(y_j+y_i)(y_j-y_i))/2 \\ y_k - y_i & ((x_k+x_i)(x_k-x_i)+(y_k+y_i)(y_k-y_i))/2 \end{vmatrix} \tag{5.3a}$$

$$H^{(3)}(i,j,k) = \begin{vmatrix} x_j - x_i & ((x_j+x_i)(x_j-x_i)+(y_j+y_i)(y_j-y_i))/2 \\ x_k - x_i & ((x_k+x_i)(x_k-x_i)+(y_k+y_i)(y_k-y_i))/2 \end{vmatrix} \tag{5.3b}$$

$$H^{(4)}(i,j,k) = \begin{vmatrix} x_j - x_i & y_j - y_i \\ x_k - x_i & y_k - y_i \end{vmatrix} \tag{5.3c}$$

The final step is to construct P_n by connecting the vertices calculated by (5.3) and (5.2) and then to remove all edges and vertices in the interior of P_n. Sugihara and Iri observe that if numerical errors occur, it is possible that the sign of H may be determined incorrectly. By making checks for topological consistency which do not involve floating point arithmetic each time a vertex is added to T, they claim that the following topological properties of the diagram are satisfied after the addition of each generator:

P1. There are as many polygons as generators.

P2. Two polygons do not share more than one edge as part of their common boundary.

There is a problem, however, with this approach. The algorithm requires that a new vertex be generated on an edge connecting a vertex in T with one not in T. However, if a numerical error results when applying Equations (5.3) and (5.2), the computed vertex may not lie on this edge. Moreover, if this vertex is close to an existing one, it is possible that the new polygon may enclose a Voronoi point which is not in T. In this case, property *P2* may be violated. In Figure 5.3, suppose that x is a true vertex of polygon P_n but that it is incorrectly located at z. Then the boundaries of P_n and P_r will have two edges in common and $q(p, m, r)$, which is not in T, will be enclosed by P_n.

If there are no numerical errors, the algorithm will construct the Voronoi diagram correctly. When a degeneracy occurs, *Algorithm Incremental1* will construct an extra edge but the length of this edge will approach zero with increasing precision. However, as we will show, the computational expense of this method is considerably more than that of an algorithm using a straightforward geometrical construction that we will introduce.

5.4. A Straightforward Implementation of the Incremental Method

In this section we present a method that produces an exact diagram by requiring that the generator coordinates be even integers. Such an approach is realistic since only a finite subset of real numbers can be represented on a computer. This algorithm requires that the range of generators be bounded and implicitly constructs an equally spaced grid on which the generators must lie. The use of floating point numbers also imposes a grid, albeit one with logarithmically spaced points.

We observe from Equations (5.3) that, if the inputs are even integers, the computation of the coordinates of the Voronoi points can be done using integer arithmetic. (The fact that the integers are even allows for division by two with no remainder.) Therefore, we assume that the generator coordinates belong to a finite subset of the even integers. Equations (5.2) and (5.3) give the intersection of the bisector $\vec{b}$ of generators p_i and p_j with the bisector $\vec{l}$ of generators p_j and p_k. These same relations can also be obtained by equating the equations of the lines $\vec{b}$ and $\vec{l}$. Since all Voronoi points are determined by intersections of this type and since the generators are even integers, each Voronoi point may be represented as an integer triple giving the numerators of each coordinate and their common denominator. If p bits are used to represent the generators, it can be seen from Equations (5.3) that $3p+2$ bits are needed to store each numerator and $2p+3$ bits to store the denominator. This is a consequence of the fact that the product of two p bit quantities requires $2p$ bits while their sum or difference requires $p+1$ bits to represent.

Since the bisector $\vec{l}$ is already in the diagram, it must also be determined if the intersection of $\vec{b}$ and $\vec{l}$ lies between the endpoints of $\vec{l}$ or, in the case of a degeneracy, at one of the endpoints. To do this, Voronoi points must be compared. Since the numerators and denominators are cross multiplied, two temporaries with at least $5p+5$ bits are required to compare two Voronoi coordinates.

A pseudocoded rendition of our implementation of the incremental method is given in Appendix B. We call it *Algorithm Incremental2*. The computations are performed by the procedures *Bisect* and *Intersect*. *Bisect* computes a point on the perpendicular bisector $\vec{b}$ of the new generator p_n and an existing generator p_i which lies on the line $\overleftrightarrow{p_n p_i}$. *Intersect* computes the intersection of $\vec{b}$ and the extension of a line $\vec{l}$ in an adjacent polygon. It then determines whether the intersection lies between the endpoints of $\vec{l}$ or coincides with one of its endpoints. The construction starts in the nearest neighbor polygon P_m. Thus, initially $i = m$. After an intersection is found, the procedure is repeated in the other polygon adjacent to the intersected edge. This process is repeated until an intersection with rear of the bisector erected in P_m is reached, which completes the cycle.

We give $\vec{b}$ an orientation such that p_n lies to the right of the bisector. The edges of the polygon

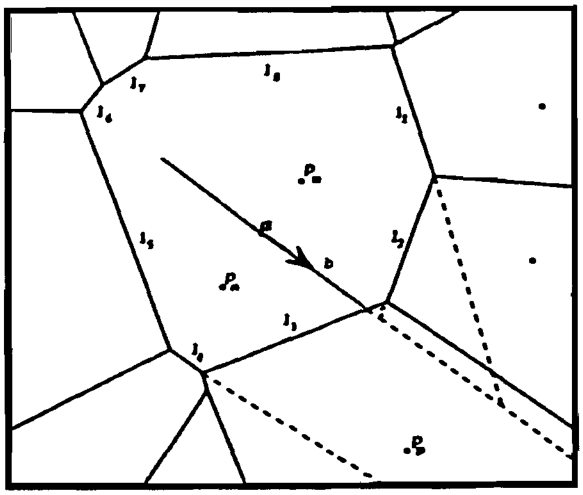


Figure 5.4. Constructing an edge in the new polygon, P_n. Point α, as well as the slope of $\vec{b}$, are stored. The extensions of the lines $\vec{l_1}$ through $\vec{l_4}$ intersect the front of $\vec{b}$. However, only line $\vec{l_3}$ is intersected between its endpoints. Since the starting edge is arbitrary, the expected number of intersection computations in this polygon is 2.5.

have a counterclockwise orientation but are added to the diagram in clockwise order. We refer to the left endpoint of an edge as that which is clockwise adjacent to the other endpoint. Finally, we say that a point or edge of a polygon lies to the left (resp. right) of another point or edge in the polygon if it is encountered first (resp. last) in traversing the polygon from the starting point in counterclockwise order. Figure 5.4 shows the bisector $\vec{b}$ of $\overline{p_n p_m}$ along with α, the point computed on $\vec{b}$.

The following features are included to minimize computations:

1. The coordinates of the point on the perpendicular bisectors that are computed by *Bisect* are stored. The numerator and denominator of the slope of each bisector, which are computed by taking the difference two generator coordinates, are also stored. These sums and differences of generator coordinates are computed only once for each line. This avoids repeating the additions and subtractions in Equations (5.3) several times.
2. From the orientation of p_n and p_i and the relative slopes of $\vec{b}$ and $\vec{l}$, it is possible to determine

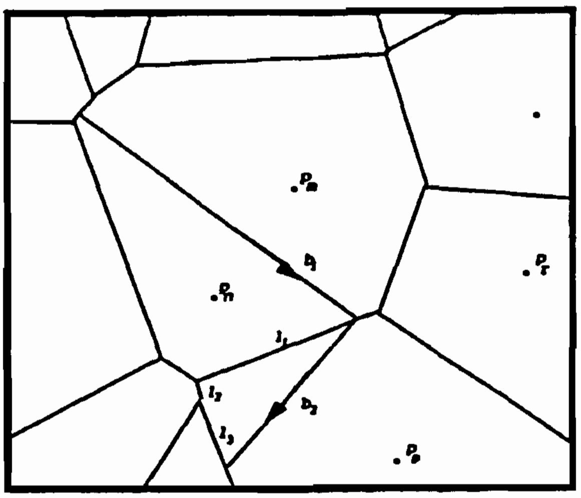


Figure 5.5. Inspection of edges in polygon P_p starts with line $\vec{l}_2$ since an intersection with $\vec{l}_1$ is impossible. In determining that there is no intersection with $\vec{l}_2$, it is also determined that there is no intersection to the right of $\vec{l}_3$. Hence, only the right endpoint of $\vec{l}_3$ has to be checked.

if the front of $\vec{b}$ intersects $\vec{l}$ extended. On the average, half of the edges in the adjacent polygon are eliminated as candidates for intersection without having to compute a possible intersection point. This is true since, in order for an edge to intersect the front of $\vec{b}$, its front endpoint with respect to the direction of $\vec{b}$ must at least as close to $\vec{b}$ as its rear endpoint. If we assume that each line orientation is equally likely, then this occurs with probability one half. In Figure 5.4, the extensions lines $\vec{l}_1$ through $\vec{l}_4$ intersect the front of $\vec{b}$ and those of the rest of the lines in P_m do not.

3. With the exception of the nearest neighbor polygon P_m, in which the starting point is arbitrary, the search for an intersection starts with an edge such that the number of intersection checks is minimal. We see in Figure 5.5 that since $\vec{b}_2$ turns clockwise with respect to $\vec{b}_1$, a counterclockwise search will result in fewer edges being inspected. Because another intersection with $\vec{l}_1$ is impossible, the best edge to start with is $\vec{l}_2$. In the example depicted two edges, $\vec{l}_2$ and $\vec{l}_3$, have to be inspected in order to locate the next intersection. Experiments reveal that an average 2.0 intersection checks are needed in P_m and only 1.4 in each of the other polygons adjacent to P_n.

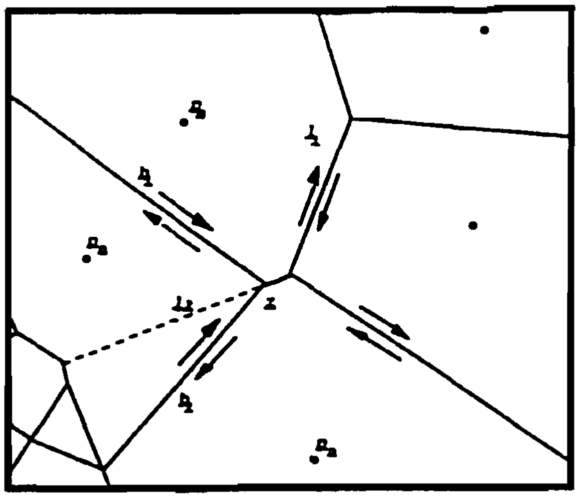


Figure 5.6. A non-degenerate intersection. Voronoi point x and lines $\vec{b_1}$ and $\vec{b_2}$ are added. The dotted line indicates the portion of $\vec{l}$ that is removed. The arrows indicate the orientation of the edges in the polygons.

4. Once it has been determined that $\vec{b}$ intersects $\vec{l}$ extended, it must then be decided if this intersection lies between the endpoints of $\vec{l}$ inclusive. Due to the order in which the edges are checked, it is necessary to check both endpoints only in P_m. In the other polygons, if bisector $\vec{b}$ passes to the left of an edge it must intersect a previously inspected edge. This follows since the inspection starts to the right of $\vec{b}$.

Handling Degeneracies

Figure 5.6 shows an example of a nondegenerate intersection. First, the intersection x of $\vec{b_1}$ and $\vec{l}$ is determined. Then the portion of $\vec{l}$ on the same side of $\vec{b_1}$ as p_n is removed and x becomes an endpoint of $\vec{b_1}$ and a new endpoint of $\vec{l}$. The three polygons affected by the addition of the new Voronoi point are the new polygon P_n, its nearest neighbor P_m, and polygon P_p which is on the opposite side of $\vec{l}$ from P_m. When the cycle is complete, $\vec{b_2}$ (which is constructed in P_p) is connected to $\vec{b_1}$ in P_n, $\vec{b_1}$ is connected to $\vec{l}$ in P_m, and $\vec{l}$ is connected to $\vec{b_2}$ in P_p. The connections turn counterclockwise around generators p_n, p_m and p_p, respectively.

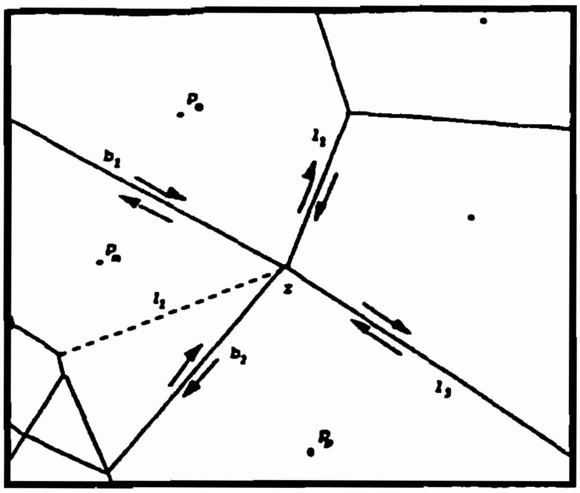


Figure 5.7. A degenerate intersection. Lines $\overrightarrow{b_1}$ and $\overrightarrow{b_2}$ are added and $\overrightarrow{l}$ is deleted. The degree of vertex z increases from 3 to 4.

Figure 5.7 illustrates the degenerate case. Before point p_n is added, vertex z has degree three. Since $\overrightarrow{b_1}$ passes through z, a right turn must be made around $\overrightarrow{l_2}$ into its other adjacent polygon P_p. The convention used is that a bisector intersects only right endpoints. Thus in the figure, $\overrightarrow{b_1}$ is considered to intersect the endpoint of $\overrightarrow{l_2}$ rather than that of $\overrightarrow{l_1}$. The next polygon is, therefore, P_p. Bisector $\overrightarrow{b_2}$ is then erected in P_p. At the end of the cycle, $\overrightarrow{b_2}$ is connected to $\overrightarrow{b_1}$ in P_n, $\overrightarrow{b_1}$ is connected to $\overrightarrow{l_1}$ (the counterclockwise successor of $\overrightarrow{l_2}$ in P_m) and $\overrightarrow{l_3}$ (the clockwise successor of $\overrightarrow{l_2}$) is connected to $\overrightarrow{b_2}$ in P_p. Just as in the non-degenerate case, the connections turn counterclockwise around generators p_n, p_m and p_p respectively. Line $\overrightarrow{l_2}$ becomes disconnected by this process and is marked as deleted. After the addition of the generator p_n, vertex z has degree four. Regardless of the initial degree of vertex z, it is increased by one by this process.

5.5. Efficiency Considerations

As we have stated in Section 5.2, the incremental method takes constant time to add a point for a broad class of distributions of generators. We have concentrated on reducing the time constant as much as possible in developing *Incremental2*. In this section we will compare the number of

	Incremental1	*Incremental2*
Multiplications	$180n$	$112n$
Additions/Subtractions	$360n$	$57n$
Divisions by two	$36n$	$12n$

Table 5.1. Average computations performed by each algorithm.

arithmetic operations done by *Incremental2* with those done by *Incremental1*.

According to Sugihara and Iri, *Incremental1* requires about 30×10^7 computations of H to construct a diagram for one million generators, or about 30 computations per generator. However, an efficient implementation of the algorithm can significantly reduce this total. It is well known that the number of edges in a Voronoi polygon approaches six from below as the number of generators increases [38]. A reasonable estimate of the number of computations of H required to add a generator may be obtained as follows: First, one computation is required for each vertex (edge) of the nearest neighbor polygon P_m. We recall from Section 5.2 that T is defined as the set of vertices to be removed from the diagram when generator p_n is added. One computation is required for each element of T that is a vertex of P_m since two of its three adjacent vertices are in P_m and have thus already been checked. Finally, two computations are required for every other element of T since one of its adjacent vertices has already been added to T and the other two have not.

In order to obtain a comparison of the number of arithmetic operations performed by the two algorithms, we will assume that both the new polygon, P_n, and polygon P_m have six edges, which is the asymptotic average for individual polygons. Then T will contain four vertices since, in the absence of degeneracies, the total number of Voronoi points always increases by two when a new generator is added. Let us also assume that two elements of T are vertices of P_m.

Table 5.1 summarizes the average number of arithmetic operations performed by each algorithm under these assumptions. The total number of computations of H required to add generator p_n is twelve. An efficient computation of H requires 15 multiplications, 30 additions/subtractions and 3 divisions by two. Thus, for the case where polygons P_m and P_n each have six edges and two elements of T are vertices of P_m, 180 multiplications, 360 addition/subtractions and 36 divisions by two are required to add the generator.

We consider now the computation required by *Incremental2* to add the same generator. Since P_n has six edges, there are six calls to *Bisect* which require a total of 12 additions/subtractions and 12 divisions by two. There are two calls to *Intersect* on the average when processing polygon P_m. This is so because only edges having potential intersections with the front of the bisector must be checked and there are three such intersections, on the average, for a polygon of size six. In each of the other polygons *Intersect* is called slightly more than once on the average because of the order in which the edges are inspected. Experiments reveal that for a polygon with six edges, there are an average of seven calls to *Intersect* for the remaining polygons adjacent to P_n. As observed previously, both endpoints have to be checked when *Intersect* is called in P_m and only one when it is called in the other polygons. Thus, on the average, there are a total of nine calls to *Intersect* and eleven endpoint checks required to add a polygon with six edges.

Each call to *Intersect* requires 10 multiplications and 5 additions/subtractions and each endpoint check requires 2 multiplications. Therefore, adding the new generator requires a total of 112 multiplications, 57 additions/subtractions and 12 divisions by two.

As a result we see that, on the average, *Incremental2* uses over one third fewer multiplications, one sixth as many additions, and one third the number of divisions by two as *Incremental1*.

Space Complexity

The space required to implement *Incremental2* is governed by the value selected for p, the number of bits used to represent the generator coordinates, and by the size of the pointers. In the following we assume that 16, 32 and 64 bit word sizes are available. The largest datum is the numerator of a Voronoi point coordinate, which we take as close to 64 bits as possible. The largest p consistent with this choice is 20 bits, which allows for about 500,000 different values for each generator coordinate. If we choose, we can reduce p to 16 bits in order to save space if 32,000 coordinate values are sufficient for the application. We can select two byte pointers for small diagrams ($n \leq 64K$) and

	Number	Unit Size (bytes)	Total Size (bytes)
Generators	n	4	$4n$
Voronoi Points	$2n$	24	$48n$
Lines			
endpoint pointers	$6n$	4	$24n$
pointers to adjacent polygons	$6n$	4	$24n$
points/slopes	$3n$	16	$48n$
Polygon pointers	$6n$	4	$24n$
Polygon heads	n	4	$4n$
Total			$180n$

Table 5.2. Space requirements of *Incremental2*.

four byte pointers for larger diagrams.

To obtain a realistic comparison with *Incremental1*, which was tested for one million generators, we will assume that four byte pointers are used and that $p = 20$. Table 5.2 shows the details of the space requirements for *Incremental1* under these assumptions.

Sugihara and Iri report that their algorithm uses 180 bytes per generator [38]. Included in this total is the space occupied by the data structure that they use to determine the order in which to add the generators. This, however, is a relatively small proportion of the overall space overhead, about $15n$ bytes, assuming that four byte pointers are used. Thus, the storage requirements of the two algorithms are comparable.

We note from Table 5.2 that storing the points and slopes, which makes the computation of intersections efficient, contributes a significant amount to the space requirement ($48n$ bytes). Of course, this overhead could be eliminated if one is willing to compute these quantities each time they are needed. We also observe that over half of the space ($100n$ bytes) is consumed by pointers. For small diagrams this total could be cut in half by using 16 bit pointers.

5.6. Conclusion

We have presented an efficient implementation of the incremental method for constructing the Voronoi diagram in the plane. We assume that the possible sites for generators are evenly spaced and that, therefore, their coordinates may be represented by even integers. This allows the construction of an error-free diagram. We have compared our algorithm that of Sugihara and Iri, which is the best known method of constructing the diagram in $O(n)$ expected time. In doing so we have shown that our technique makes about one third fewer multiplications and far fewer of the other types of arithmetic operations. At the same time, it does not sacrifice any of the properties of the diagram since there are no numerical errors.

Although we have concentrated on time efficiency in our implementation, the space requirements for large diagrams are comparable to those of Sugihara and Iri's algorithm. In addition, adjustments can be made in cases where space is the dominant concern and the number of *generators*, n, is not too large. Since over half of the space overhead is occupied by pointers, a considerable amount of space can be saved by reducing the size of the pointers for small n. Also, the required space may be reduced by over 25% if one is willing to compute the sums and differences of generator coordinates each time they are used. In concluding we observe that our algorithm is especially well suited to situations where the probability of degeneracies occuring is significant.

CHAPTER 6

AN EFFICIENT CLOSEST PAIR ALGORITHM

6.1. Introduction

The closest pair problem may be stated as follows: Given n points in d dimensional Euclidean space ($\Re^d$), find the two points that are the closest. A brute force solution is to find the minimum of all pairwise distances between points using $O(n^2)$ comparisons. Optimal decision tree algorithms usually employ a divide and conquer approach. The idea is to first divide the space into two subsets of points. Next, the minimum inter-point distance in each subset is found by applying the algorithm recursively. The results are then combined by processing a slice, which can be shown to contain an expected sublinear number of points, at the junction of the subsets. Bentley, Weide and Yao [4] have given $O(n)$ expected time algorithms for solving a number of closest point problems in d dimensional space. For example, they show how under certain conditions on the probability density function that the all nearest neighbors problem can be solved in linear expected time. Since one of the pairs of nearest neighbors is the closest pair, it follows immediately that the closest pair problem can also be solved in $O(n)$ expected time. It should be possible, however, to obtain a more efficient closest pair algorithm since finding all nearest neighbors is not a prerequisite for solving the problem. Of course, such an algorithm would be still $O(n)$, but with a smaller constant of proportionality.

We present here an algorithm for the closest pair problem whose expected running time is asymptotically proportional to dn, where d is the dimension of the space and n is the number of points. This is an important achievement since, as Bentley et. al. [4] report, the constant of linearity of most bucketing algorithms increases exponentially with d. We show that for $d \leq 5$, the algorithm performs well for all n, but that when d is large, n must also be large for the algorithm to work efficiently. It is believed that this is also true for all closest pair algorithms [4].

The key idea for this algorithm comes from a theorem of Yuval [42], which he used to construct an $O(n \log n)$ worst-case algorithm with the technique of divide and conquer. This result is as follows: Construct a partition of d dimensional Euclidean space by subdividing the space into hypercubes of size $(d+1)\delta$ on a side, where δ is a small positive real number. Then construct d additional partitions by shifting the origin of the new partition $-\delta$ in all coordinate directions with respect to the previous partition. If there exist two points that are closer than δ, they must lie in the same cell in at least one partition. Using this theorem, a closest pair algorithm will be developed here that runs in linear expected time.

In 1976 Rabin [35] published a probabilistic closest pair algorithm based on Yuval's result which has linear expected time performance. His method is to use a large number of buckets whose size is determined by the closest pairwise distance among points in a random sample. The algorithm requires computing and sorting d integer sequences, whose distribution is highly non-uniform. Moreover, it may be possible that the range of the sequence may exceed the range of integers available on many computers. The advantage is that, if it can be implemented, its expected performance is independent of the data distribution of the point coordinates.

Hinrichs, Nievergelt and Schorn [16] have published a two dimensional plane sweep algorithm that runs in $O(n)$ expected time, provided the sorting phase can be assumed to be linear. Golin [14] has shown that this algorithm can be simplified in that it is unnecessary to use complicated data structures. Unfortunately, the algorithm does not retain its linear expected time property when generalized to higher dimensions.

We will show that the closest pair problem in d dimensional space can be solved efficiently using a bucketing approach. In Section 6.2 we present our closest pair algorithm. An expression is derived in Section 6.3 for $E(\delta_0)$, the expected value of the closest interpoint distance within the buckets. This permits the derivation of an expression for $E(D)$, the expected value of the number of distance computations performed by the algorithm. This quantity is the dominant factor controlling the algorithm's running time. In Section 6.4 we present empirical data showing that values obtained

```
Algorithm CPair
Input: n points, $X_{i,j}, i = 1, 2, \ldots, n; j = 1, 2, \ldots, d$ in $[0, 1]^d$.
Output: The closest pair, $X_p, X_q$ and closest distance, $\delta$.
1. Find $M_j = \max_i X_{i,j}, m_j = \min_i X_{i,j}, j = 1, 2, \ldots, d$ and construct a grid with at most $n - 1$
   rectangular buckets having minimum edge length $\epsilon$.
2. Distribute the points into the buckets and find the closest pair and closest distance, $\delta_0$, within
   any of the buckets by comparing all pairwise distances between points in the same bucket.
3. If $\delta_0 > \frac{\epsilon}{(d+1)}$ then set $\epsilon = \min\{1, (d+1)\delta_0\}$, redistribute, and find a new closest pair and $\delta_0$
   within the buckets.
4. if $\epsilon < 1$ then for $l := 1$ to $d$ do
     a) Shift the grid by $-\delta_0$ in each coordinate direction.
        If $X_i$ is closer that $(l+1)\delta_0$ to at least $l$ bucket boundaries, then distribute $X_i$ into a bucket.
     b) Find the closest pair and the minimum distance, $\delta_1$ between all items placed into buckets
        in step 4a.
     c) Set $\delta = min\{\delta_0, \delta_1\}$ and save the pair of $X_i$ associated with the new $\delta$.
   end for
5. Return the current $\delta$ and closest pair.
```

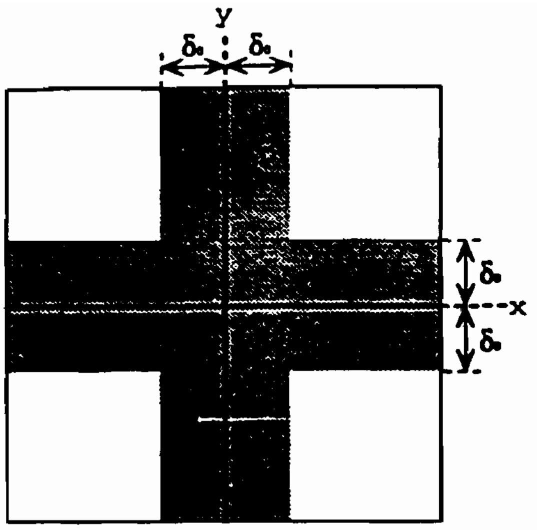
Figure 6.1. Algorithm *CPair*.

from the expressions for $E(D)$ and $E(\delta_0)$ agree well with the average of these quantities determined from experiments. We also show that the running time of this algorithm is much less than that of several others for the case when $d = 2$.

6.2. Our Closest Pair Algorithm

Our closest pair algorithm is given in Figure 6.1. The inputs are assumed to be random vectors from $[0, 1]^d$. We first construct a grid with at most $n - 1$ rectangular buckets having minimum edge length ϵ. Then we distribute the points into the buckets and determine δ_0, the closest interpoint distance within any of the buckets. In the event that $\frac{\epsilon}{(d+1)} < \delta_0$, we set $\epsilon = \min\{1, (d+1)\delta_0\}$ and redistribute. Provided $\epsilon < 1$, we shift the grid by $-\delta_0$ in each coordinate direction, since Yuval's result holds if the cells in the partition have edge lengths greater than $(d+1)\delta_0$. We then find a new δ_0 which is the same or smaller than the current one. The shifting process is repeated d times. It turns out that in most cases only a fraction of the points considered at one pass need to

a) In the first pass only points within δ_0 of either the x or y bucket boundaries remain closest pair candidates.



b) In the second pass, only points closer than δ_0 to either boundaries x and y' or y and x' remain closest pair candidates. These points are contained in a region whose points are less than $3\delta_0$ above x'' and y''.

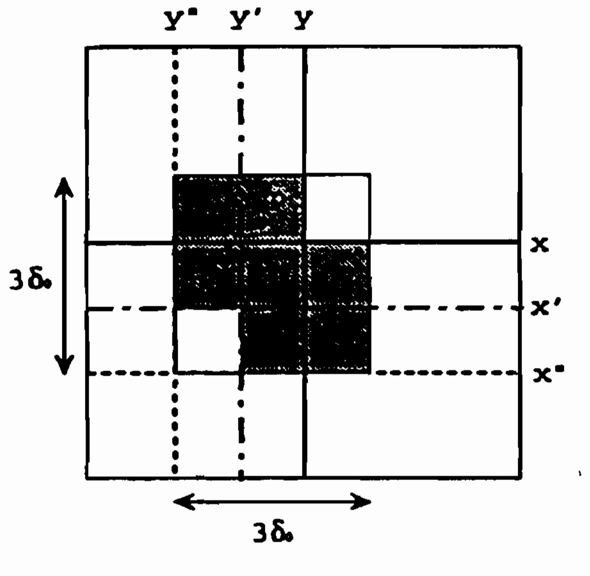


Figure 6.2. Points that remain closest pair candidates after the first two passes ($d = 2$).

be considered at the next. This is because for a point to remain a candidate for closest pair in the lth pass, it must be closer than δ_0 to at least $l-1$ perpendicular bucket boundaries, one from each of the passes to date. Consequently, at the lth pass, all closest pair candidates are points that are closer than $l\delta_0$ to at least $l-1$ of their lower bucket boundaries.

Figure 2 illustrates the case where $d=2$. In Figure 2a the heavy lines represent the boundaries of four contiguous buckets. All points in the shaded areas remain closest pair candidates for pass 2 because they are closer than δ_0 to at least one of two perpendicular bucket boundaries, and can thus be closer than δ_0 to a point lying across one of the boundaries. In Figure 2b the lines x and y represent bucket boundaries for pass 1, which are shifted for pass 2 to x' and y' respectively. For pass 3 these boundaries are shifted again to x'' and y''. After the second pass (Figure 6.2b), the points in the shaded region remain candidates for pass 3 because they are closer than δ_0 to two perpendicular boundaries, one each from passes 1 and 2. Therefore, given a point the shaded region, say p_1, it may be possible to find another point p_2 that lies across two boundaries (x and y') and is still closer than δ_0 to p_1. We see that the square of size $3\delta_0$ on a side contains these points. Membership in these square regions can thus be determined efficiently using d (in this case, $d=2$) comparisons per point.

6.3. Analysis of the Algorithm

In order to analyze the performance of our algorithm, we assume the points to be independent random vectors from $\Re^d$ with a common density f having compact support. To simplify the analysis we assume, without loss of generality, that the support is $[0,1]^d$. Devroye [7] has shown that, under these conditions, an algorithm using $O(n)$ buckets has expected time complexity $O(n)$ if the amount of time, $g(n)$, required to process points in the buckets satisfies the relation $\int_{-\infty}^{\infty} g(f(x))dx < \infty$. In algorithm *CPair* $g(i) = \frac{i(i-1)}{2}$, where i is the number of points in a particular bucket. If $\int_{-\infty}^{\infty} f^2(x)dx < \infty$, *Cpair* is clearly $O(n)$ by Devroye's result. Whenever $\delta_0 \leq \frac{t}{(d+1)}$, $n-1$ buckets are used in each of the $d+1$ passes. Intuitively, δ_0 must decrease as n increases so that it must be less than $\frac{t}{(d+1)}$ when n is larger than some n_0. In fact since $\delta_0 \sim 0$, the number of distance computations is asymptotically the same as the number of comparisons performed by *Hybridsort* (see Chapter 3). However for values of n encountered in practice, $\delta_0 > \frac{1}{(d+1)}$ with significant probability when d is large.

Expected Number of Distance Computations

First we derive a lower bound for the probability that δ_0 is less than some x under the assumption that the data density is uniform. This is a lower bound for all other densities as well.

Lemma. Given n points uniformly distributed in $[0,1]^d$, partition the space into n buckets of size $\frac{1}{n^{1/d}}$ on a side. Let δ_0 be the closest interpoint distance within any of the buckets. Then $P(\delta_0 < x) \geq 1 - \epsilon$, where $\epsilon = \prod_{i=1}^{n-1} \frac{m-i}{m}$ and $m = \frac{\sqrt{d}^d}{x^d}$.

Proof: Construct a refinement of the current partition whose grid size is as close to $\frac{x}{\sqrt{d}}$ as possible. Since the expected value of the range in each direction is extremely close to one, this grid contains $m = \frac{\sqrt{d}^d}{x^d}$ cells. Therefore, ϵ is the probability that each cell contains at most one point since, if the first i points lie in different buckets, the probability that the $i+1$st point lies in an unoccupied bucket is $\frac{m-i}{m}$. This implies that at least one cell contains no fewer than two points with probability $1-\epsilon$ and thus $\delta_0 < x$. Since δ_0 may also be less than x when each cell contains at most one point, $1-\epsilon$ represents a lower bound on this probability.

For computational purposes, ϵ may be approximated well as follows. Using Stirling's approximation for factorials we may write

$$m! = \sqrt{2\pi m}\, m^m e^{-m}\left(1 + O\left(\frac{1}{m}\right)\right).$$

Taking $1 + O(\frac{1}{m}) \approx 1$, we may approximate ϵ closely by

$$\begin{aligned}\epsilon &= \prod_{i=1}^{n-1} \frac{m-i}{m} = \frac{1}{m^{n-1}} \frac{m!}{(m-n)!} \\ &\approx \frac{1}{m^n} \frac{m^{m+1/2}}{(m-n)^{m-n+1/2}} e^{-n} \\ &= \left(\frac{m}{m-n}\right)^{m+1/2} e^{-n}.\end{aligned}$$

Setting $t = m/n$ (thus $m - n = n(t-1)$) and since $m >> 1/2$, we have

$$\begin{aligned} c &\approx \left(\frac{t}{t-1}\right)^{n(t-1)} e^{-n} \\ &\approx \exp\left(-n\left((t-1)\ln(\frac{t}{t-1}) - 1\right)\right). \end{aligned}$$

Since the preceding relation is a very close approximation for c and $t > 1$ we may write using series expansions,

$$\begin{aligned} c &= \exp\left(-n\left((t-1)\ln(\frac{t}{t-1}) - 1\right)\right) \\ &\leq \exp\left(-n\left((t-1)\ln(1+\frac{1}{t}) - 1\right)\right) \\ &\leq \exp\left(-n\left((t-1)(\frac{1}{t} - \frac{1}{2t^2}) - 1\right)\right) \\ &\leq \exp\left(-\frac{3n}{2t}\right) = \exp\left(-\frac{3n^2}{2}\frac{1}{\sqrt{d}^d}x^d\right). \end{aligned}$$

Thus
$$P(\delta_0 < x) \geq 1 - \exp\left(-\frac{n^2}{2}\frac{3}{\sqrt{d}^d}x^d\right). \qquad \square$$

Golin [14, p. 32] has shown that, neglecting lower order terms,

$$P\left(\delta < x\right) \leq 1 - \exp\left(-\frac{n^2}{2}\omega_d x^d\right),$$

where ω_d is the volume of the unit d dimensional hypersphere. Since $\delta_0 \geq \delta$, this is also an upper bound for $P(\delta_0 < x)$. Thus we may write

$$P\left(\delta_0 < x\right) = 1 - \exp\left(-\frac{n^2}{2}\beta_d x^d\right), \quad \frac{3}{\sqrt{d}^d} \leq \beta_d \leq \omega_d. \qquad (6.1)$$

We observe that, except for very large d, the lower bound in (6.1) is tighter than the value $\omega_d/4^d$ derived by Golin [14, p. 33] for $P(\delta < x)$.

We may estimate the expected number of distance computations, $E(D)$, performed in a particular pass as follows. With $\alpha = 1$ and $n - 1 \approx n$, the volume of each cell may be taken as $1/n$. Let the expected proportion of each bucket containing points that enter into distance computations in a given pass be w. The probability that a region of volume $\frac{w}{n}$ contains i points is given in [26] by

$$\binom{n}{i}\left(\frac{w}{n}\right)^i\left(1 - \frac{w}{n}\right)^{n-i} \sim \frac{w^i}{i!}e^{-w}.$$

The number of distance computations in such a region is $\frac{i(i-1)}{2}$, the number of point pairs. Since the probability density is the same for all n regions, the expected number of distance computations for each pass is

$$E(D_w) = n\sum_{i=0}^{n} \frac{i(i-1)}{2i!} w^i e^{-w} \sim \frac{nw^2}{2}\sum_{i=2}^{\infty} \frac{w^{i-2}}{(i-2)!} e^{-w} = \frac{nw^2}{2}.$$

$E(D)$ is, therefore, asymptotic to ${}^{n}\!/_{2}$ times the sum of the squares of w for each pass.

Assuming $\delta_0 < \frac{\epsilon}{(d+1)}$ and letting $r = \frac{\delta_0}{\epsilon} = n^{1/d}\,\delta_0$, we obtain for $d = 2$,

$$E(D) = {}^{n}\!/_{2} E\Big(1 + 4\big(2r - (2r)^2\big)^2 + \big(3r\big)^4\Big).$$

We see from Figure 6.2 that, for uniformly distributed points, w for each pass is the area of the shaded regions in each bucket divided by the area of the bucket.

For $d = 3$ we may calculate

$$E(D) = {}^{n}\!/_{2} E\Big(1 + 9\big(2r - (2r)^2\big)^2 + 9\big((3r)^2 - (3r)^3\big)^2 + \big((4r)^3\big)^2\Big),$$

and, in general, we have

$$E(D) = {}^{n}\!/_{2} E\big(1 + 4d^2r^2 + O(r^4)\big). \tag{6.2}$$

Since $E(D)$ is determined by terms of the form r^k, we wish to obtain an expression for $E(\delta_0^k)$, and thus for $E(r^k)$. We can do this by using (6.1). It has been shown in [14, p. 31] that $x_{max} = sn^{-1/d}$, where s is a constant that depends on d. Thus

$$\begin{aligned} E(\delta_0^k) &= \int_0^{sn^{-1/d}} x^k P'(\delta_0 < x)dx \\ &= \frac{n^2\beta_d d}{2}\int_0^{sn^{-1/d}} x^{d+k-1}\exp\Big(-\frac{n^2\beta_d x^d}{2}\Big)dx. \end{aligned} \tag{6.3a}$$

Substituting $v = \frac{n^2\beta_d\, x^d}{2}$, (6.3a) may be written as

$$E(\delta_0^k) = n^{-2\,k/d}\left(\frac{2}{\beta_d}\right)^{k/d}\int_0^{s^d\,{}^{n}\!/_{2}\beta_d} v^{k/d}e^{-v}dv. \tag{6.3b}$$

Since Golin has further shown [14, p. 33] that

$$\int_0^{s^d\,{}^{n}\!/_{2}\beta_d} v^{k/d}e^{-v}dv = \Gamma({}^{k}\!/d + 1) + O\Big(\frac{1}{n}\Big),$$

we have that

$$E(r^k) = n^{k/d}\, E(\delta_0^k) = n^{-k/d} \left(\frac{2}{\beta_d}\right)^{k/d} \left(\Gamma(k/d+1) + O\left(\frac{1}{n}\right)\right). \tag{6.3c}$$

Setting $k = 2$ in (6.3c) and substituting into (6.2), we obtain

$$E(D) = n/2\left(1 + 4d^3\, n^{-2/d} \left(\frac{2}{\beta_d}\right)^{2/d} \Gamma(2/d+1) + O(n^{-4/d})\right). \tag{6.4}$$

We note that since $E(D)$ varies inversely with β_d, setting $\beta_d = \omega_d$ determines a lower bound for δ_0 and $E(D)$, whereas $\beta_d = \frac{3}{\sqrt{d}}$ provides an upper bound.

It can be seen from (6.4) that $E(D) \sim {}^n\!/_2$. Since the number of arithmetic operations in each distance computation is proportional to d, the expected running time is asymptotically proportional to dn.

6.4. Experimental Results

In order to test the validity of (6.3c) and (6.4), and to determine the most likely value of β_d, we tested an implementation of *CPair* in which the number of distance computations and the value of δ_0 were computed after each pass. The results revealed that the values of δ_0 and δ were very close to the lower bound, therefore we chose $\beta_d = \omega_d$. By setting $k = 1$ and $E(r) = \frac{1}{d+1}$, we can calculate the value n_0 of n such that $E(r) = \frac{1}{d+1}$ for a given d. Thus, for all $n > n_0, E(r) < \frac{1}{d+1}$. These results are shown in Table 6.1. Table 6.2 shows values for $E(\delta_0)$ obtained from (6.3c) with $\beta_d = \omega_d$ and neglecting the $O(\frac{1}{n})$ term, along with the average δ_0 obtained by ten runs of *Cpair* for each n and d. Also shown are the average values of δ obtained experimentally, when different from δ_0. Table 6.3 shows a comparison of the number of distance computations made by *CPair* in the second and subsequent passes (Step 4b) with the value of the second term of (6.4).

Comparison with Other Algorithms

We have implemented this algorithm in Pascal on a VAXStation 3100, along with a classical divide and conquer algorithm [34], Bentley, Weide and Yao's linear expected time algorithm [4] and an algorithm based on the plane sweep method [16]. Because these algorithms are either restricted

	d					
	2	3	4	5	6	7
n_0	4	21	170	1,927	29,030	556,466

Table 6.1. Values of n_0 so that $E(r) < \frac{1}{(d+1)}$ when $n \geq n_0$.

	$E(\delta_0)$ from equation (3c) with $\beta_d = \omega_d$		Average δ_0 from experiments	
d	$n = 10{,}000$	$n = 100{,}000$	$n = 10{,}000$	$n = 100{,}000$
2	0.707109×10^{-4}	0.707109×10^{-5}	0.641467×10^{-4}	0.659593×10^{-5}
3	0.150370×10^{-2}	0.323963×10^{-3}	0.158803×10^{-2}	0.359990×10^{-3}
4	0.723203×10^{-2}	0.228697×10^{-2}	0.659719×10^{-2}	0.174118×10^{-2}
5	0.190051×10^{-1}	0.756606×10^{-2}	0.189496×10^{-1}	0.830968×10^{-2}

	Average δ (if different from δ_0)	
d	$n = 10{,}000$	$n = 100{,}000$
2		
3		
4	0.650909×10^{-3}	0.166607×10^{-2}
5	0.180288×10^{-2}	0.802772×10^{-2}

Table 6.2. Expected value of δ_0.

	Experimental Average		Second term of Equation (4)	
d	$n = 10{,}000$	$n = 100{,}000$	$n = 10{,}000$	$n = 100{,}000$
2	5	4	5	5
3	206	490	214	461
4	1,191	3,235	1,805	5,709
5	5,423	26,421	7,567	30,124

Table 6.3. $E(D)$ in step 4b of *CPair*.

to the plane or are difficult to implement in higher dimensions, we have restricted the comparisons to the case where the number of dimensions $d = 2$. The results, shown in Table 6.4, clearly indicate

Algorithm	Space (bytes)	Execution Time (sec.)		
		$n = 1{,}000$	$n = 10{,}000$	$n = 50{,}000$
CPair	$24n$	0.13	1.18	5.88
Divide and Conquer (iterative improvement)	$32n$	0.68	8.93	47.65
Algorithm of Bentley, Weide and Yao	$24n$	1.11	11.30	57.75
Plane Sweep	$29n$	0.50	5.37	28.35

Table 6.4. Execution times (in sec.) for various closest Pair Algorithms.

that our algorithm outperforms the others by wide margins.

CHAPTER 7

DATA TRANSFORMATIONS

In this chapter we will consider methods for dealing with non-uniform data distributions. Although we will confine our discussion to sorting, these methods can also be applied to geometric problems in higher dimensions. In 1980, Meijer and Akl [29] observed that if the cumulative distribution function, $F(x)$, of the data is known and can be computed in constant time then Hybridsort is $O(n)$. However, the requirement for *a priori* knowledge of the data distribution is a serious impediment to the successful application of bucketing algorithms. A worthwhile goal is to apply an $O(n)$ order preserving transformation to data from an unknown distribution such that the transformed data is close to uniform.

7.1. General Techniques

Several techniques have been employed to obtain an approximation to $F(x)$ if it is not known. One approach is to subdivide a range of data values into equal length intervals, or "cells," and to obtain estimates, $\hat{F}(x)$, of $F(x)$ at the endpoints of each interval. This can be done by constructing a histogram based on the values of a random sample of the data. Various curve fitting techniques have been tried [24,31] but it is generally agreed that piecewise linear approximation works best. Since the width of each interval is the same, the cell that contains a given datum can be determined in constant time and thus the required linear interpolation can be done rapidly. Janus and Lamagna [25] employed this technique, which they called the "CDF adaptation," using a fixed number of cells and a fixed sample size. They subdivided the *entire* range of the data set into cells. Noga and Allison [31] used a linear number of samples and cells ($0.01n$ cells and about $0.2n$ samples). They split the *sample* range into equal width cells and used two additional cells for outliers.

When the density has compact support, $\hat{F}(x)$ provides a good estimate of $F(x)$ for fairly small samples [25]. Thus, there is no advantage to using a non-constant number of cells or samples. Moreover, the more costly estimate of the range provided by the data set minimum and maximum is not justified. On the other hand, when the density does not have compact support, there may be an advantage to using a linear number of cells. This is because the sample maximum and minimum divide the range into three regions – one containing the points between them and two outlier regions containing the points above and below them. The expected proportion of outliers is $\frac{2}{(s+1)}$, where s is the sample size [30]. Therefore, when s is linear in the number of data items, the expected number of outliers is constant. However, since the sample range is unbounded, it is unclear under what conditions the time required to sort the interior cells is $O(n)$. Noga and Allison [31] claim that linearity holds for densities with "either exponentially vanishing tails or compact support (without strong peaks)" but do not provide sufficient proof of their claim. Even when the density has compact support, there is a problem in using fixed width cells. If $F(x)$ changes dramatically from one cell to the next, the linear approximation of F fails to smooth the data.

Another approach is to subdivide the range of F into approximately equal intervals by sorting a small sample of the data of size s. The ith smallest sample gives a good estimate of $F^{-1}(\frac{i}{s+1})$ for moderate sized samples [30]. This technique subdivides the range of F into $s + 1$ cells and insures an even distribution of data points to those cells. The problem is that the work required to determine cell membership for each data point grows with the number of samples. Therefore, s must be independent of n or the sort will be non-linear. Janus [24] tested an implementation of this method, which he called the "Ranking Method," and found that the overhead was costly. However, the distributions he tested had either compact support or exponentially vanishing tails, and therefore his CDF adaptation worked better. Devroye [7] implies that the Ranking technique may be good for "thick tailed" densities such as the Cauchy. We have tested this method and have found that it outperforms the CDF adaptation for practical input sizes when applied to Cauchy data even though the algorithm is asymptotically non-linear. Since the number of cells must be fixed, there are a linear number of outliers. If Quicksort is used to sort the outliers, the overall complexity is $T(n) = c_1 n \log n + O(n)$. By proper choice of sample size, c_1 can be made sufficiently small so that the sort appears to be linear for reasonable values of n. We present experimental results to support this claim in Section 7.2.

Another way of treating densities with non-compact support is to apply an order-preserving transformation, such as $h(x) = \frac{1}{1+|x|}$ which maps $[-\infty,\infty]$ onto $[-1,1]$. Devroye [7] points out that if the data density has exponentially dominated tails then the density of the transformed data is bounded.

None of these methods work well with densities that have strong peaks. It has been shown [6] that Hybridsort is $O(n)$ for densities, $f(x)$, with compact support if $\int g(f(x))dx < \infty$, where g is the complexity of the secondary sort. Unfortunately, it is possible for the integral, and thus the time constant, to be quite large albeit finite. On the other hand, the integral may be infinite while the sort is only marginally super-linear.

The ideal would be a probabilistic algorithm that does not depend on the data distribution. The Ranking Method comes closest because it insures, with high probability, that the cell proportions are essentially equal. However, the data density still affects the bucket populations within the cells. Using Rabin's probabilistic approach, the problem of sorting real numbers can be transformed to that of sorting integers. As we observed in Chapter 1, however, a uniformly distributed set of reals would generate a highly non-uniform integer sequence. Of course, radix sort could be used, in which case the performance would be $O(n)$ and not affected by the data distribution. The time constant, however, would not be favorable.

It follows from the preceding discussion that empirical testing is required to determine which transformation method(s) work best in practice.

7.2. Experimental Results on Cauchy Data

The sorting method we used was as follows. We first selected the 0.5 and 99.5 percentiles of the dataset in $O(n)$ time. Since n is large, these values provided good estimates of the corresponding

n (1000 items)	*Distribution Phase*	*InsSort*	*QSort*	*Total*
40	12.30	0.97	0.08	11.35
50	14.10	1.20	0.09	14.39
60	15.90	1.44	0.11	17.41
70	18.63	1.67	0.13	20.43
80	21.44	1.90	0.15	23.49
90	24.26	2.13	0.17	26.56
100	27.07	2.36	0.18	29.61
110	29.88	2.60	0.20	32.68
120	32.69	2.84	0.22	35.75

Table 7.1. Running times (in sec.) for *Hybridsort* on Cauchy data.

percentiles of the population. We then distributed all of the points that fell between these percentiles into buckets using the Ranking method, and sorted the buckets using *Insertionsort*. Since the probability density of points restricted to this range is bounded and has compact support, the time to sort these points is $O(n)$. The outliers were sorted using Quicksort. We see from the results in Table 7.1 that, despite the fact that the Quicksort phase is non-linear, it accounts for only a small fraction of the running time for n as high as 120,000.

We tested an implementation of the CDF adaptation on this same data and found that the running time was over ten times slower than the Ranking method for the same values of n. Incidentally, the running times of this algorithm are very close to that of Sedgewick's Quicksort, which of course does not depend on the data distribution. Quicksort was slightly better for smaller values of n and the Ranking method was a little better for the larger values of n tested. This is, of course, due to the $n \log n$ term in the running time of Quicksort.

REFERENCES

[1] A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley (1974).

[2] D. C. S. Allison and M. T. Noga, "Selection by Distributive Partitioning," *Information Processing Letters* 11 (1980), 7-8.

[3] T. Asano, M. Edahiro, H. Imai and M. Iri, "Practical Use of Bucketing Techniques in Computational Geometry," in *Computational Geometry*, G. T. Toussaint (editor), Elsevier, North Holland (1985).

[4] J. L. Bentley, B. W. Weide and A. C. Yao, "Optimal Expected-Time Algorithms for Closest Point Problems," *ACM Transactions on Mathematical Software* 6 (1980), 563-580.

[5] P. J. Denning, "Virtual memory," *Computing Surveys* 2 (1970), 153-189.

[6] L. Devroye and T. Klincsek, "Average Time Behavior of Distributed Sorting Algorithms," *Computing* 26 (1981), 1-7.

[7] L. Devroye, *Lecture Notes on Bucket Algorithms*, Birkhäuser (1986).

[8] Digital Equipment Corporation, *VAX/VMS System Management* (1980).

[9] Digital Equipment Corporation, *VAX Hardware Handbook* (1982).

[10] W. Dobosiewicz, "Sorting by Distributive Partitioning," *Information Processing Letters* 7 (1978), 1-6.

[11] W. Dobosiewicz, "The Practical Significance of D. P. Sort Revisited," *Information Processing Letters* 8 (1979), 170-172.

[12] R. W. Floyd and R. L. Rivest, "Expected Time Bounds for Selection," *Communications of the ACM* 18 (1975), 165-172.

[13] R. W. Floyd and R. L. Rivest, "Algorithm 489 (Select)," *Communications of the ACM* 18 (1975), 173.

[14] M. J. Golin, "Probabilistic Analysis of Geometric Algorithms," Technical Report CS-TR-266-90, Princeton University (1990).

[15] P. J. Green and R. Sibson, "Computing Dirichlet Tessellations in the Plane," *The Computer Journal* 21 (1978), 168-173.

[16] K. Hinrichs, J. Nievergelt and P. Schorn, "Plane Sweep Solves the Closest Pair Problem Elegantly," *Information Processing Letters* 26 (1988), 255-261.

[17] C. A. R. Hoare, "Algorithm 63 (Partition)," and "Algorithm 65 (Find)," *Communications of the ACM* 4 (1961), 321.

[18] C. A. R. Hoare, "Quicksort," *Computer Journal* 5 (1962), 10-15.

[19] R. N. Horspool, "Constructing the Voronoi Diagram in the Plane," Technical Report SOCS-79.12, McGill University, Montreal, Canada, (1979).

[20] M. Huits and V. Kumar, "The Practical Significance of Distributive Partitioning Sort," *Information Processing Letters* 8 (1979), 168-169.

[21] G. A. Hyslop and E. A. Lamagna, "Performance of Distributive Partioned Sorting in a Demand Paging Environment," *Information Processing Letters* 35 (1987), 61-64.

[22] G. A. Hyslop and E. A. Lamanga, "An Application of the Method of Buckets to the Selection Problem," in H. Berghel, E. Deaton, G. Hendrick, D. roach and R. Wainwright (eds.), *Applied Computing: Technological Challenges of the 1990'S*, ACM Press (1992), 231-236.

[23] G. A. Hyslop and E. A. Lamagna, "Error Free Incremental Construction of Voronoi Diagrams in the Plane," in E. Deaton, K. M. George, H. Berghel, G. Hendrick (eds.), *Applied Computing: States of the Art and Practice - 1993*, ACM Press (1993), 388-396.

[24] P. J. Janus, "Adaptive Methods for Unknown Distributions in Disributive Partitioning Sorting," M. S. Thesis, The University of Rhode Island (1981).

[25] P. J. Janus and E. A. Lamagna, "An Adaptive Method for Unknown Distributions in Distributive Partitoned Sorting," *IEEE Transactions on Computers* C-34 (1985), 367-372.

[26] O. Kempthorne and L. Folks, *Probability, Statistics and Data Analysis*, Iowa State University Press (1971).

[27] D. E. Knuth, *The Art of Computer Programming*, Vol. 1: *Fundamental Algorithms* (second edition), Addison-Wesley (1973).

[28] D. E. Knuth, *The Art of Computer Programming*, Vol. 3: *Sorting and Searching*, Addison-Wesley (1973).

[29] H. Meijer and S. G. Akl, "The Design and Analysis of a New Hybrid Sorting Algorithm," *Information Processing Letters* 10 (1980), 213-218.

[30] F. Mosteller, *Sturdy Statistics: Nonparametrics and Order Statistics*, Addison-Wesley (1973).

[31] M. T. Noga and D. C. S. Allison, "Sorting in Linear Expected Time," *BIT* 25 (1985), 451-465.

[32] T. Ohya, M. iri and K. Murota, "A Fast Voronoi-Diagram Algorithm with Quaternary Tree Bucketing," *Information Processing Letters* 18 (1984), 227-231.

[33] T. Ohya, M. Iri and K. Murota, "Improvements of the Incremental Method for the Voronoi Diagram with Computational Comparison of Various Algorithms," *Journal of the Operations Research Society of Japan* 27 (1984), 306-337.

[34] F. P. Preparata and M. I. Shamos, *Computational Geometry: An Introduction*, Springer-Verlag (1985).

[35] M. O. Rabin, "Probabilistic Algorithms," in *Algorithms and Complexity*, J. F. Traub (editor), Academic Press (1976), 21-39.

[36] R. Sedgewick, "Implementing Quicksort Programs," *Communications of the ACM* 21 (1978), 847-856.

[37] M. I. Shamos, "Computational Geometry," Ph.D. Thesis, Department of Computer Science, Yale University (1978).

[38] K. Sugihara and M. Iri, "Construction of the Voronoi Diagram for over 10^5 Generators in Single Precision Arithmetic," *First Canadaian Conference on Computational Geometry*, Montreal, Canada (1989).

[39] A. S. Tanenbaum, *Structured Computer Organization* (third edition), Prentice-Hall (1990).

[40] A. J. Thomasian, *The Structure of Probability Theory with Appplications*, McGraw Hill (1969).

[41] M. van der Nat, "A Fast Sorting Algorithm, A Hybrid of Distributive and Merge Sorting." *Information Processing Letters* 10 (1979), 163-167.

[42] G. Yuval, "Finding Nearest Neighbors," *Information Processing Letters* 5 (1976), 63-65.

APPENDIX A

THE BUCKETSORT ALGORITHM

```
                                                  Expected
                                                  Exec. Freq.
    { initialize }
 1   if n is odd then                             1
       max := min := A[1]
     else
       max := maximum (A[1],A[2]);
       min := minimum (A[1],A[2])
     endif;
     { main loop }
     for each pair of elements
       (A[i],A[i+1]) remaining do
 2      if A[i] > A[i+1] then                     $n/2$
 3         if A[i] > max then                     $n/4$
 4            max := A[i];                        $H_n$
 5         if A[i+1] < min then                   $n/4$
 6            min := A[i+1]                       $H_n$
        else
 7         if A[i] < min then                     $n/4$
 8            min := A[i];                        $H_n$
 9         if A[i+1] > max then                   $n/4$
10            max := A[i+1]                       $H_n$
        endif;
```

Step 2. Determining the Maximum and Minimum.

```
                                                  Expected
                                                  Exec. Freq.
 1   RangeInv := $\alpha n$/(max-min);            1
     for i := 1 to n do
        begin
 2         bucket := $\lfloor$(A[i]-min) * RangeInv+1$\rfloor$;   $n$
 3         S[i] := L[bucket];                     $n$
 4         L[bucket] := i                         $n$
        end;
```

Step 3. Distribution of Items into Buckets.

```
                                                        Expected
                                                        Exec. Freq.
    { initialize }
1   i := 1;                                             1
2   link := 0;                                          1
3   S[0] := 0;                                          1
    { loop for each bucket }
4   while i ≤ αn do                                     $\alpha n + 1$
      begin
      { skip over empty buckets }
5     while L[i] = 0 do                                 $\alpha n$
6       i := i+1;                                       $(\alpha - B(\alpha))n$
      { find tail of previous bucket }
7     while S[link] ≠ 0 do                              $(1 + B(\alpha) - \epsilon)n$
8       link := S[link];                                $(1 - \epsilon)n$
       { attach $i$th bucket to $i$−1st bucket }
9     S[link] := L[i];                                  $B(\alpha)n$
      { go to next bucket }
10     i := i+1                                         $B(\alpha)n$
    end; { while }
```

Note: The value of *max* in step 2 is adjusted slightly so that the largest item is placed in the last bucket, thereby avoiding an extra comparison at each iteration of line 4 of this step.

Step 4. Bucket Chaining.

```
                                                        Expected
                                                        Exec. Freq.
    { initialize }
1   link := S[0];                                       1
2   i := 1;                                             1
    { main loop }
3   while link > 0 do                                   $n + 1$
       { equivalently, $i \leq n$ }
       begin
         { chase pointers to find item }
4        while link ≤ i do                              $2n - H_n$
5          link := S[link];                             $n - H_n$
6        Swap (A[i], A[link]);                          $n$
         { set pointers so that the
          A[i] may be found in turn }
7        tempptr := S[link];                            $n$
8        S[link] := S[i];                               $n$
9        S[i] := link;                                  $n$
10       link := temptr;                                $n$
         { go to next position
          in data array }
11       i := i+1                                       $n$
       end; { while }
```

Step 5. The MacLaren Routine.

```
                                                  Expected
                                                  Exec. Freq.
1 A[n+1] := maxint;                               1
2 for i := n-1 downto 1 do                        $n-1$
3   if A[i] > A[i+1] then                         $n-1$
4      temp := A[i];                              $D(\alpha)\alpha n$
5      j := i+1;                                  $D(\alpha)\alpha n$
6      while A[j] ¡ tempdo                        $E(\alpha)\alpha n$
         begin
7          A[j-1] := A[j];                        $(E(\alpha) - D(\alpha))\ \alpha n$
8          j := j+1                               $(E(\alpha) - D(\alpha))\ \alpha n$
         end; { while }
9      A[j-1] := temp                             $D(\alpha)\alpha n$
    endif
```

Step 6. Insertion sort.

APPENDIX B

INCREMENTAL ALGORITHMS FOR VORONOI DIAGRAMS

Algorithm Incremental1

```
// Let $G_n$ be the Voronoi diagram for the first $n$ generators //
1. Construct a triangle large enough to contain all the generators and consider the vertices of the
   triangle as the initial generators $p_1$, $p_2$ and $p_3$.
   // Thus, Voronoi polygons 4, 5, ... are closed. //
2. For $n = 4, 5, \ldots n_{max}$ do
   2.1 Select a subset, $T$, of the vertex set $G_{n-1}$ as follows:
    2.1.1 Find the Voronoi polygon containing $p_n$, and, among the Voronoi points $q(i,j,k)$ on the
          boundary of this polygon, find the one that gives the smallest value of $H(p_i, p_j, p_k, p_n)$.
          Initialize $T$ to the singleton consisting of this point.
    2.1.2 Repeat until $T$ can no longer be augmented:
            For each Voronoi point $q(i,j,k)$ that is connected by a Voronoi edge to an element of
            $T$, add this point to $T$ if $H(p_i, p_j, p_k, p_m) < 0$ and if the resultant $T$ satisfies topological
            conditions $P1$ and $P2$ of Section 3.
   2.2 For every edge connecting a vertex in $T$ to a vertex not in $T$, generate a new vertex on it and
       divide the edge into two edges.
   2.3 Construct new edges connecting the vertices generated in step 2.1 in such a way that they
       form a cycle that encloses only the vertices of $T$.
   2.4 Remove the vertices of $T$ and the edges incident to them.
// The interior of the cycle is the Voronoi region of $p_n$ and the resulting embedded graph is $G_n$. //
```

Algorithm Incremental2

```
// Implementation:
$p$ is the array of generators.
The array $L$ contains lines that have been added to the diagram.
Removed lines are marked for deletion.
Actual deletion takes place after the new point has been added.
Each polygon, $n$, is represented by a circular edge list with an arbitrary edge, $P[n]$, as the head.
The links point to the counterclockwise successor.
Left($e$) denotes the counterclockwise adjacent edge of $e$.
Right(e) denotes its clockwise adjacent edge.
Since it is only rarely used (only in the case of degeneracies), Right(e) is implemented by traversing
the circular list counterclockwise which avoids maintaining pointers in both directions.
Since each line is contained in two polygons, it corresponds to two edges.
Lines and edges have the following relation:
The $i$th line corresponds to edges $2i - 1$ and $2i$.
Thus Line(e) $= L[(e+1)$ div $2)]$. //
```

```
procedure AddPoint($p_n, n, L, P$);
// Input: $p_n, n$, the new generator its polygon
          $L, P$, the current set of lines and polygons
  Output: $L, P$, The updated set of lines and polygons //
// local variables //
  var $p_m, p_k$: point; $m, k$: polygon; e*, $e_0$: edge;
  begin
    NearestNeighbor($p_n, p_m, P_m$);
// returns $p_m$, the nearest neighbor of $p_n$ and its polygon, $P_m$ //
      $k := P_m$;//process the nearest neighbor polygon first//
      $p_k := p_m$;
      $e_0$ := head of $P_m$;
// start the search for intersection at the head of $P_m$ //
    repeat
      ProcessPolygon($p_n, p_k, e_0, b, e, z, degen$);
// Erects the perpendicular bisector, $b$, of the line $\overline{p_n p_k}$ and returns the edge, $e$, in polygon $k$
intersected by the front of $b$ and the point of intersection, $z$. If the intersection is degenerate, $degen$
is true, otherwise it is false. //
      add $b$ to $L$;
      $k_0 := k$; // save the current polygon //
      NextPolygon ($e, e*, k$);
// Input is the current $e$ and current polygon, $k$.
Output is the corresponding edge in the adjacent polygon, e*, and the adjacent polygon, $k$ //
      $p_k := p[k]$; // get generator of adjacent polygon //
// store the edges and current polygon for connection later, see Section 4 for details on how degen-
eracies are handled //
      if $degen$ then
        EnqueAddEdges (Right($e*$),Left($e$),$k_0$)
      else
        EnqueAddEdges ($e*, e, k_0$);
// start the search for next intersection at the counterclockwise adjacent edge of e* //
      $e_0$ := Left($e*$);
    until $k = m$;
    for each polygon, $k$, processed do
      DequeueAdjustEndpoints
        ($degen_{prev}, degen, p_k, z_{prev}, z, b, l_{prev}, l$);
      AdjustEndpoints($b, p_n, p_k, l_{prev}, degen_{prev}$);
      AdjustEndpoints($b, p_n, p_k, l, degen$);
      DequeueAddEdges($e_{prev}, e, k$);
      AddEdges($e_{prev}, e, k, n, P$)
    end for;
  end // AddPoint //

procedure ProcessPolygon($p_n, p_k, k, e_0, b, e, z, degen$):
// Input: $p_n$, the new generator $p_k, k$, the generator and the polygon being processed, $e_0$, the edge
at which the search for an intersection starts.
  Output: $b$, the bisector of the line $\overline{p_n, p_k}$
  $e$, the edge in polygon $k$ intersected by the front of $b$
  $z$, the point of intersection of $b$ with this edge
 $degen$, true if the intersection is degenerate and false otherwise //
```

```
  begin
    ErectBisector($\overline{p_n, p_k}$, $b$);
    $e := e_0$;
    repeat
      if Line($e$) extended intersects the front end of $b$ then
// Intersect returns the intersection of $b$ and $l$ (extended), $z$, and true if $z$ is between the end points
of $l$ ($degen$ = false). Intersect also returns true if $b$ intersects the endpoint of $l$ that is located
counterclockwise from the other endpoint (degen = true). Otherwise, Intersect returns false. //
        if Intersect($p_k, l, b, z, degen$) then
          EnqueAdjustEndpoints($degen, p_k, z, b, l$);
          return
        endif;
      $e :=$ Left($e$)
    until $e = e_0$
  end; // ProcessPolygon //

procedure AdjustEndpoints($degen, p_n, p_k, z, b, l$);
// Input: $degen$, true if the intersection is degenerate, false
           otherwise.
           $p_n$, the new generator
           $p_k$, the generator of the polygon being processed
           $b$, the bisector of the line $\overline{p_n p_k}$
           $l$, the line in polygon $k$ intersected by $b$
           $z$, the point of intersection of $b$ and $l$.
  Output: $b$ and $l$ with their endpoints adjusted //
  begin
    Using the endpoints of $l$, which have already been determined, and the location of $p_n$ and $p_k$,
    set one of the endpoints of $b$ to $z$;
    if not degen and this is the first call with this $b$ then
      reset one of the endpoints of $l$ to $z$
  end;

procedure AddEdges($e_1, e_2, k, n, P$);
// Input:  $e_1, e_2$, the edges to be connected with new edges.
$k, n$, the polygon being processed and the new polygon, respectively. $P$, the set of polygons
  Output: $P$, the set of polygons with the new edges added //
  begin
    Create an edge connecting $e_1$ and $e_2$ and prepend it to polygon $P[n]$;
    Create an edge connecting $e_2$ and $e_1$ in polygon $P[k]$, marking disconnected lines for deletion;
  end;

  begin // Algorithm Incremental2 //
// Initialize $P$ to the three pseudo points and $L$ to the three semi-infinite rays //
    Initialize($L, P$);
    for $n := 4, 5, \ldots n_{max}$ do
      AddPoint ($p_n, n, L, P$);
      remove the lines marked for deletion;
    end for
  end. // Algorithm Incremental2 //
```

BIBLIOGRAPHY

Aho, A. V., Hopcroft, J. E., and Ullman J. D.,, *The Design and Analysis of Computer Algorithms*, Addison-Wesley (1974).

Allison, D. D. S., and Noga, M. T., "Selection by Distributive Partitioning," *Information Processing Letters* 11 (1980), 7-8.

Asano, T., Edahiro, M., Imai, H., and Iri, M., "Practical Use of Bucketing Techniques in Computational Geometry," in *Computational Geometry*, G. T. Toussaint (editor), Elsevier, North Holland (1985).

Bentley, J. L., Weide, B. W., and Yao, A. C.,"Optimal Expected-Time Algorithms for Closest Point Problems," *ACM Transactions on Mathematical Software* 6 (1980), 563-580.

Denning, P. J., "Virtual memory," *Computing Surveys* 2 (1970), 153-189.

Devroye, L., and Klinczek, T., "Average Time Behavior of Distributed Sorting Algorithms," *Computing* 26 (1981), 1-7.

Devroye, L., *Lecture Notes on Bucket Algorithms*, Birkhäuser (1986).

Digital Equipment Corporation, *VAX/VMS System Management* (1980).

Digital Equipment Corporation, *VAX Hardware hangbook* (1982).

Dobosiewicz, W., "Sorting by Distributive Partitioning," *Information Processing Letters* 7 (1978), 1-6.

________"The Practical Significance of D. P. Sort Revisited," *Information Processing Letters* 8 (1979), 170-172.

Floyd, R. W., and Rivest, R. L., "Expected Time Bounds for Selection," *Communications of the ACM* 18 (1975), 165-172.

________"Algorithm 489 (Select)," *Communications of the ACM* 18 (1975), 173.

Golin, M. J.,"Probabilistic Analysis of Geometric Algorithms," Technical Report CS-TR-266-90, Princeton University (1990).

Green, P. J. and Sibson, R.,"Computing Dirichlet Tessellations in the Plane," *The Computer Journal* 21 (1978), 168-173.

Hinrichs, K., Nievergelt. J., and Schorn, P., "Plane Sweep Solves the Closest Pair Problem Elegantly," *Information Processing Letters* 26 (1988), 255-261.

Hoare, C. A. R., "Algorithm 63 (Partition)," and "Algorithm 65 (Find)," *Communications of the ACM* 4 (1961), 321.

________"Quicksort," *Computer Journal* 5 (1962), 10-15.

Horspool, R. N., "Constructing the Voronoi Diagram in the Plane," Technical Report SOCS-79.12, McGill University, Montreal, Canada, (1979).

Huits, M., and Kumar, V., "The Practical Significance of Distributive Partitioning Sort," *Information Processing Letters* 8 (1979), 168-169.

Hyslop. G. A., and Lamagna, E. A., "Performance of Distributive Partioned Sorting in a Demand Paging Environment," *Information Processing Letters* 35 (1987), 61-64.

________"An Application of the Method of Buckets to the Selection Problem," in H. Berghel, E. Deaton, G. Hendrick, D. roach and R. Wainwright (eds.), *Applied Computing: Technological Challenges of the 1990'S*, ACM Press (1992), 231-236.

________"Error Free Incremental Construction of Voronoi Diagrams in the Plane," in E. Deaton, K. M. George, H. Berghel, G. Hendrick (eds.), *Applied Computing: States of the Art and Practice - 1993*, ACM Press (1993), 388-396.

Janus, P. J., "Adaptive Methods for Unknown Distributions in Disributive Partitioning Sorting,"M. S. Thesis, The University of Rhode Island (1981).

Janus, P. J., and Lamagna, E. A., "An Adaptive Method for Unknown Distributions in Distributive Partitoned Sorting," *IEEE Transactions on Computers* C-34 (1985), 367-372.

Kempthorne O., and Folks, L., *Probability, Statistics and Data Analysis,* Iowa State University Press (1971).

Knuth, D. E., *The Art of Computer Programming,* Vols. 1-3: Addison-Wesley (1973).

Meijer H., and Akl, S. G., "The Design and Analysis of a New Hybrid Sorting Algorithm," *Information Processing Letters* 10 (1980), 213-218.

Mosteller, F., *Sturdy Statistics: Nonparametrics and Order Statistics,* Addison-Wesley (1973).

Noga, M. T., and Allison, D. C. S., "Sorting in Linear Expected Time," *BIT* 25 (1985), 451-465.

Ohya, T., Iri, M., and Murota, K., "A Fast Voronoi-Diagram Algorithm with Quaternary Tree Bucketing," *Information Processing Letters* 18 (1984), 227-231.

________"Improvements of the Incremental Method for the Voronoi Diagram with Computational Comparison of Various Algorithms," *Journal of the Operations Research Society of Japan 27* (1984), 306-337.

Preparata F. P., and Shamos, M., I. *Computational Geometry: An Introduction*, Springer-Verlag (1985).

Rabin, M. O., "Probabilistic Algorithms," in *Algorithms and Complexity*, J. F. Traub (editor), Academic Press (1976), 21-39.

Sedgewick, R., "Implementing Quicksort Programs," *Communications of the ACM* 21 (1978), 847-856.

Shamos, M. I.,"Computational Geometry," Ph.D. Thesis, Department of Computer Science, Yale University (1978).

Sugihara K., and Iri, M., "Construction of the Voronoi Diagram for over 10^5 Generators in Single Precision Arithmetic," *First Canadaian Conferenece on Computational Geometry*, Montreal, Canada (1989).

Tanenbaum, A. S., *Structured Computer Organization* (third edition), Prentice-Hall (1990).

Thomasian, A. J., *The Structure of Probability Theory with Appplications,* McGraw Hill (1969).

van der Nat, M., "A Fast Sorting Algorithm, A Hybrid of Distributive and Merge Sorting." *Information Processing Letters* 10 (1979), 163-167.

Yuval, G., "Finding Nearest Neighbors," *Information Processing Letters* 5 (1976), 63-65.