PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS
Coração Eucarístico

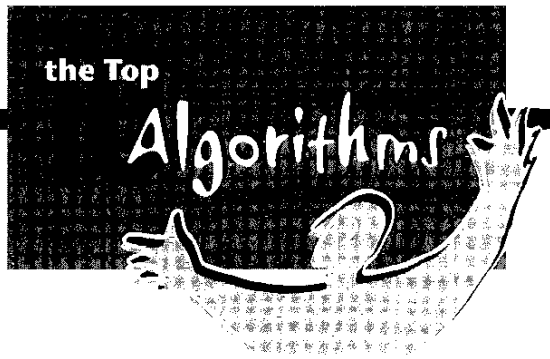
Disciplina	Curso	Turno	Período
Projeto e Análise de Algoritmos	Ciência da Computação	Manhã	5°

Professor
Felipe Cunha (felipe@pucminas.br)

"The question of whether a computer can think is no more interesting than the question of whether a submarine can swim."
Edsger W. Dijkstra

Artigo: A Perspective on Quicksort. Joseph Jaja, Jan/Feb 2000. Volume 2. Issue 1. Pages 43- 49. Computing in Science & Engineering

1. No melhor caso do Quicksort, qual elemento deveria ser selecionado como o pivô? Justifique sua resposta.
2. Descreva as estratégias para selecionar o pivô e o pior caso de cada estratégia.
3. Qual é a configuração de entrada que leva ao pior caso do Quicksort? Dê um exemplo de uma configuração de entrada que leva ao pior caso.
4. Por que o Quicksort é considerado o melhor algoritmo de ordenação mesmo possuindo o pior caso igual a $\Theta(n^2)$?
5. Em quais situações não deveríamos usar o Quicksort?
6. Apresente o algoritmo do Quicksort iterativo.



A PERSPECTIVE ON QUICKSORT

This article introduces the basic Quicksort algorithm and gives a flavor of the richness of its complexity analysis. The author also provides a glimpse of some of its generalizations to parallel algorithms and computational geometry.

Sorting is arguably the most studied problem in computer science, because of both its use in many applications and its intrinsic theoretical importance. The basic sorting problem is the process of rearranging a given collection of items into ascending or descending order. The items can be arbitrary objects with a linear ordering. For example, the items in a typical business data-processing application are records, each containing a special identifier field called a key, and the records must be sorted according to their keys. Sorting can greatly simplify searching for or updating a record. This article focuses only on the case where all the items to be sorted fit in a machine's main memory. Otherwise, the main objective of sorting, referred to as *external sorting*, is to minimize the amount of I/O communication, an issue this article does not address.

Although researchers have developed and analyzed many sorting algorithms, the Quicksort algorithm stands out. This article shows why.

1521-9615/00/$10.00 © 2000 IEEE

JOSEPH JAJA
University of Maryland

Staying power

Tony Hoare presented the original algorithm and several of its variations in 1962,[1] yet Quicksort is still the best-known practical sorting algorithm. This is quite surprising, given the amount of research done to develop faster sorting algorithms during the last 38 years or so. Quicksort remains the sorting routine of choice except when we have more detailed information about the input, in which case other algorithms could outperform Quicksort.

Another special feature of the algorithm is the richness of its complexity analysis and structure, a feature that has inspired the development of many algorithmic techniques for various combinatorial applications. Although the worst-case complexity of Quicksort is poor, we can rigorously prove that its average complexity is quite good. In fact, Quicksort's average complexity is the best possible under certain general complexity models for sorting. More concretely, Quicksort performs poorly for almost sorted inputs but extremely well for almost random inputs.

In retrospective, Quicksort follows the divide-and-conquer strategy (likely one of the oldest military strategies!), one of the most powerful algorithmic paradigms for designing efficient algorithms for combinatorial problems. In particular, a variation of Quicksort, which Hoare also mentioned in his original paper, is based on a randomization step using a random-number generator.

Researchers cultivated this idea years later and significantly enriched the class of randomized algorithms, one of the most interesting areas in theoretical computer science. You could argue that these developments are somewhat independent of Quicksort, but many of the randomized combinatorial algorithms are based on the same randomized divide-and-conquer strategy described in Hoare's original paper. Indeed, many randomized algorithms in computational geometry can be viewed as variations of Quicksort.

What makes Quicksort so interesting from the complexity-analysis point of view is its far superior average complexity.

Finally, a generalization of the randomized version of the Quicksort algorithm results in a parallel sample-sorting strategy, the best-known strategy for sorting on parallel machines, both from a theoretical perspective and from extensive experimental studies.[2]

The basic algorithm

The divide-and-conquer strategy has three phases. First, divide the problem into several subproblems (typically two for sequential algorithms and more for parallel algorithms) of almost equal sizes. Second, solve independently the resulting subproblems. Third, merge the solutions of the subproblems into a solution for the original problem. This strategy's efficiency depends on finding efficient procedures to partition the problem during the initial phase and to merge the solutions during the last phase. For example, the fast Fourier transform follows this strategy, as does Quicksort. Here is a high-level description of Quicksort applied to an array $A[0 : n - 1]$:

1. Select an element from $A[0 : n - 1]$ to be the pivot.
2. Rearrange the elements of A to partition A into a left subarray and a right subarray, such that no element in the left subarray is larger than the pivot and no element in the right subarray is smaller than the pivot.
3. Recursively sort the left and the right subarrays.

Ignoring the implementation details for now, it is intuitively clear that the previous algorithm will correctly sort the elements of A. Equally clear is that the algorithm follows the divide-and-conquer strategy. In this case, Steps 1 and 2 correspond to the partitioning phase. Step 3 amounts to solving the induced subproblems independently, which sorts the entire array, making a merging phase unnecessary.

Two crucial implementation issues are missing from our initial description of the Quicksort algorithm. The first concerns the method for selecting the pivot. The second concerns the method for partitioning the input once the pivot is selected.

Selecting the pivot

Because a critical assumption for an efficient divide-and-conquer algorithm is to partition the input into almost equal-size pieces, we should select the pivot so as to induce almost equal-size partitions of the array A. Assuming A contains n distinct elements, the best choice would be to select the median of A as the pivot. Although some good theoretical algorithms can find the median without first sorting the array, they all incur too much overhead to be useful for a practical implementation of Quicksort.

In reality, there are three basic methods to select the pivot. The first and simplest is to select an element from a fixed position of A, typically the first, as the pivot. In general, this choice of the pivot does not work well unless the input is random. If the input is almost sorted, the input will be partitioned extremely unevenly during each iteration, resulting in a very poor performance of the overall algorithm. The second method for selecting the pivot is to try to approximate the median of A by computing the median of a small subset of A.

One commonly used method is to select the pivot to be the median of the first, the middle, and the last elements of A. Such a method works well, even though we may still end up with very uneven partitions during each iteration for certain inputs. Finally, we can randomly select an element from A to be the pivot by using a random-number generator. In this case, we can rigorously prove that each step will result in an almost even partition with very high probability, regardless of the initial input distribution.

Partitioning the input

Because the algorithm is recursive, I describe a simple partitioning procedure for a general subarray $A[l, r]$, where $l < r$. The procedure manipulates two pointers i and j, where i moves from left to right starting at position l, and j moves from right to left starting at position r. The pointer i is continually incremented until an element $A[i]$ larger than the pivot is encountered. Similarly, j is decremented until an element $A[j]$ smaller than the pivot is encountered. At that time, $A[i]$ and $A[j]$ are exchanged, and the process continues until the two pointers cross each other, which indicates the completion of the partitioning phase. More precisely, Figure 1 gives the following pseudocode procedure for partitioning $A[l, r]$, where all the elements of A are assumed to be distinct, and the left-most element is chosen as the pivot.

```
procedure partition(A, l, u)

begin

Step 1. Set i = l; j = r + 1; pivot = A[l];

Step 2. while(true){
          while(A[++i] < pivot);
          while(A[--j] > pivot);
          if i < j then exchange A[i] and A[j];
            else break;
            }
          A[l] = A[j];
          A[j] = pivot;

end procedure
```

Figure 1. The partitioning procedure for $A[l, r]$.

The partition procedure starts by initializing the pointers i and j and selecting the left-most element to be the pivot. Note that j is initially set to the value $r + 1$ so that the `while` loop for j in Step 2 will start by examining the subarray's right-most element. The last two assignments in Step 2 ensure that the pivot is placed in its correct position in the final sorted order of A. The partition procedure takes linear time with a very small constant factor. Donald Knuth[3] attributes this particular partitioning procedure to Robert Sedgewick.[4]

Quicksort initially calls the partition procedure with the values $l = 0$ and $r = n - 1$, followed by recursive calls to handle the subarrays $A[l, j - 1]$ and $A[j + 1, r]$. We can eliminate the recursive calls by using a stack to keep track of the partitions yet to be sorted. We run Quicksort until the number of elements in the subarray is small (say, fewer than 30); then we use a simpler sorting procedure, such as insertion sort.

Complexity analysis

We can use several measures to analyze a sorting algorithm's performance. We can count the number of comparisons the algorithm makes, because comparisons seem to be the major operations performed by any general sorting algorithm. Another important parameter is the number of swaps the algorithm incurs. Yet another measure, one that is perhaps more relevant for today's processors and their use of memory hierarchies (typically two levels of cache and main memory), is the data movement and its spatial locality. Here, I concentrate on the number of comparisons, for which there are well-known models (for example, algebraic decision trees[5]) that can derive nonlinear lower bounds for sorting.

Worst-case analysis

In general, an algorithm's most commonly used performance metric is the asymptotic estimate of the maximum amount of resources (for example, the number of operations, the number of memory accesses, and memory size) required by any instance of a problem of size n. In sorting, we determine the asymptotic number of comparisons required in the worst case. Let $T(n)$ be the number of comparisons required by our Quicksort algorithm, using any of the methods just described for selecting the pivot. Then the following recurrence equations express $T(n)$:

$$T(n) = \max_{1 \le i \le n-1} \{T(i) + T(n-i)\} + cn$$

$$T(1) = \Theta(1)$$

where cn is an upper bound on the time required to partition the input. The parameter i is the size of the left partition resulting after the first iteration. Because we are focusing on the worst-case scenario, we take the maximum over all possible values of i. It is intuitively clear that the worst case occurs when $i = 1$ (or equivalently, $i = n - 1$), which can happen for each of our three pivot-selection methods. Therefore, $T(n) = \Theta(n^2)$, which is inferior to the worst-case complexity of several of the other known sorting algorithms. However, what makes Quicksort so interesting from the complexity-analysis point of view is its far superior average complexity.

Average-case analysis

Although performing a worst-case analysis on an algorithm is usually easy, it often results in a very pessimistic performance estimate. Known alternative performance measures seem more useful, but deriving their asymptotic behavior is significantly more difficult. One such measure is to establish the average complexity under a certain probability distribution of the input. Not only is deriving such a bound difficult, but also this approach fails to offer adequate ways for defining the probability distribution of the input in most cases.

An alternative approach is to let the algorithm use a random-number generator (such as our third method for randomly selecting the pivot from the input array). In this case, a probabilistic analysis of the algorithm does not require any assumptions about the input distribution, and hence holds for any input distribution. This approach gives rise to the so-called *randomized algorithms*.

Here, I consider the case when the array's first element is selected as the pivot and assume random input. So, the pivot is equally likely to be of any rank between 1 and n. Therefore, the average complexity is given by the recurrence equations,

$$T(n) = \frac{1}{n}\left(\sum_{i=0}^{n-1}\left(T(i)+T(n-i)\right)\right)+cn$$

$$T(1) = \Theta(1)$$

The first equation is equivalent to

$$T(n) = \frac{2}{n}\left(\sum_{i=0}^{n-1} T(i)\right)+cn\ .$$

Multiplying both sides by n gives

$$nT(n) = 2\left(\sum_{i=0}^{n-1} T(i)\right)+cn^2\ .$$

Subtracting the recurrence equation corresponding to $n-1$ from the one corresponding to n, we get

$$nT(n) - (n-1)T(n-1) = 2T(n-1) + 2cn - c.$$

Rearranging the terms and dropping c (it is asymptotically irrelevant) gives

$$nT(n) = (n+1)T(n-1) + 2cn.$$

Dividing both sides by $n(n+1)$ gives

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2c}{n+1}\ .$$

In particular, we have this sequence of equations:

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2c}{n+1}$$

$$\frac{T(n-1)}{n} = \frac{T(n-2)}{n-1} + \frac{2c}{n}$$

$$\frac{T(n-2)}{n-1} = \frac{T(n-3)}{n-2} \frac{2c}{n-1}$$

M

$$\frac{T(2)}{3} = \frac{T(1)}{2} + \frac{2c}{3}$$

Adding the above equations and considering that

$$\sum_{i=3}^{n+1}\frac{1}{i} = \log_e(n+1)+\gamma-\frac{3}{2}$$

where $\gamma \approx 0.577$ is Euler's constant, we find that $T(n) = O(n \log n)$.

Under a fairly general model for sorting that even allows algebraic operations,[5] no algorithm can beat this bound. This analysis formally justifies the superior performance of Quicksort in most practical situations.

Analysis of randomized Quicksort

Consider the case where the pivot is randomly selected from the input array. With high probability—that is, with probability $1 - n^{-c}$ for some positive constant c—the resulting Quicksort algorithm's complexity is $O(n \log n)$, regardless of the initial input distribution.

Our strategy is as follows. We view the Quicksort algorithm as a sequence of iterations such that the first iteration partitions the input into two *buckets*, the second iteration partitions the previous two buckets into four buckets, and so on, until each bucket is small enough (say, ≤ 30 elements). We then use insertion sort to sort these buckets. Another way of looking at this is to unfold the recursion into a tree of partitions, where each level of the tree corresponds to an iteration's partitions. Partitioning the buckets during each iteration takes a deterministic $O(n)$ time. So, our goal is to show that the number of iterations is $O(\log n)$ with high probability.

For any specific element e, the sizes of any two consecutive buckets containing e decrease by a constant factor with a certain probability (Claim 1). Then, with probability $1 - O(n^{-7})$, the bucket containing e will be of size 30 or less after $O(\log n)$ iterations (Claim 2). Using Boole's inequality, we conclude that, with probability $1 - O(n^{-6})$, the bucket of every element has ≤ 30 elements after $O(\log n)$ iterations.[6]

Let e be an arbitrary element of our input array A. Let n_j be the size of the bucket containing e at the end of the jth partitioning step, where $j \geq 1$.

We set $n_0 = n$. Then, this claim holds:

Claim 1. $Pr\{n_{j+1} \geq 7n_j/8\} \leq 1/4$, for any $j \geq 0$.

Proof. An element a partitions the jth bucket into two buckets, one of size at least $7n_j/8$ if and only if $rank(a : B_j) \leq n_j/8$ or $rank(a : B_j) \geq 7n_j/8$. The probability that a random element is among the smallest or largest $n_j/8$ elements of B_j is at most 1/4; hence, Claim 1 follows.

We fix the element e of A, and we consider the sizes of the buckets containing e during various partitioning steps. We call the jth partitioning step successful if $n_j < 7n_{j-1}/8$. Because $n_0 = n$, the size of the bucket containing e after k successful partitioning steps is smaller than $(7/8)^k n$. Therefore, e can participate in at most $c \log(n/30)$ successful partitioning steps, where $c = 1/\log(8/7)$. For the remainder of this proof, all logarithms are to the base 8/7; so the constant c is equal to 1.

Claim 2. Among 20 log n partitioning steps, the probability that an element e goes through 20 log n – log(n/30) unsuccessful partitioning steps is $O(n^{-7})$.

Proof. The random choices made at various partitioning steps are independent; therefore, the events that constitute the successful partitioning steps are independent. So, we can model these events as *Bernoulli trials*. Let X be a random variable denoting the number of unsuccessful partitioning steps among the 20 log n steps. Therefore,

$$Pr\{X > 20\log n - \log(n/30)\} \leq Pr\{X > 19\log n\}$$

$$\leq \sum_{j>19\log n} \binom{20\log n}{j}\left(\frac{1}{4}\right)^j\left(\frac{3}{4}\right)^{20\log n - j}$$

Given that

$$\binom{n}{k} \leq \left(\frac{en}{k}\right)^k$$

we obtain that our probability is

$$\leq \sum_{j>19\log n}\left(\frac{20e\log n}{j}\right)^j\left(\frac{1}{4}\right)^j = \sum_{j>19\log n}\left(\frac{5e\log n}{j}\right)^j$$

$$\leq \sum_{j>19\log n}\left(\frac{5e\log n}{19\log n}\right)^j = \sum_{j>19\log n}\left(\frac{5e}{19}\right)^j = O\left(n^{-7}\right) .$$

Therefore, Claim 2 follows.

By Boole's inequality, the probability that one or more elements of A go through 20 log n – log(n/30) unsuccessful steps is at most $O(n \times n^{-7}) = O(n^{-6})$. Thus, with probability $1 - O(n^{-6})$, the algorithm terminates within 20 log n iterations. So, we have proven that the algorithm takes $O(n \log n)$ time with high probability.

Extension to parallel processing

For our purposes, a parallel machine is simply a collection of processors interconnected to allow the coordination of their activities and the exchange of data. Two types of parallel machines currently dominate. The first is symmetric multiprocessors (SMPs), which are the main choice for the high-end server market, and soon will be on most desktop computers. The second type clusters high-end processors through proprietary interconnect (for example, the IBM SP High-Performance Switch) or through off-the-shelf interconnect (the ATM switch, gigabit Ethernet, and so on).

One way to choose the pivots is by randomly sampling the input elements—hence, the name sample sort.

A parallel sorting algorithm tries to exploit the various resources on the parallel machine to speed up the execution time. In addition to distributing the load almost evenly among the various processors, a good parallel algorithm should minimize the communication and coordination among the different processors. Parallel algorithms often perform poorly (and sometimes execution time increases with the number of processors) primarily because of the amount of communication and coordination required between the different processors.

Quicksort's strategy lends itself well to parallel machines, resulting in *sample sorting*:

1. Select $p - 1$ pivots, where p is the number of processors available.
2. Partition the input array into p subarrays, such that every element in the ith subarray is smaller than each element in the $(i + 1)$th subarray.
3. Solve the problem by getting the ith processor to sort the ith subarray.

One way to choose the pivots is by randomly sampling the input elements—hence, the name *sample sort*. (A generalization of the bucket-sorting method, also called sample sort, involves sampling as well.[7]) As in the sequential case, the algorithm's efficiency depends on how the pivots are selected and on the method used to par-

tition the input into the p partitions.

Before addressing these issues, let's define our problem more precisely. We have n distinct elements that are distributed evenly among the p processors of a parallel machine, where we assume that p divides n evenly. The purpose of sorting is to rearrange the elements so that the smallest n/p elements appear in processor P_1 in sorted order, the second smallest n/p elements appear in processor P_2 in sorted order, and so on.

A simple way to realize the sample sorting strategy is to have each processor choose s samples from its n/p input elements (In the next section, I'll show you one way to do this.), route the ps samples into a single processor, sort the samples on that processor, and select every sth element as a pivot.[8] Each processor partitions its input elements into p groups using the pivots. Then, the first group in each processor is sent to P_1, the second to P_2, and so on. Finally, we sort the overall input by sorting the elements in each processor separately.

The first difficulty with this approach is the work involved in gathering and sorting the samples. A larger value of s improves load balancing but increases overhead. The second difficulty is that the communication required for routing the elements to the appropriate processors can be extremely inefficient because of large variations in the number of elements destined for different processors.

Consider a variation that scales optimally with high probability and has performed extremely well on several distributed-memory parallel machines using various benchmarks.[2] Here are the algorithm's steps:

1. *Randomization.* Each processor P_i $(1 \le i \le p)$ randomly assigns each of its n/p elements to one of p buckets. With high probability, no bucket will receive more than $c_1(n/p^2)$ elements for some constant c_1. Each processor then routes the contents of bucket j to P_j.
2. *Local sorting.* Each processor sorts the elements received in Step 1. The first processor then selects $p - 1$ pivots that partition its sorted sublist evenly and broadcasts the pivots to the other $p - 1$ processors.
3. *Local partitioning.* Each processor uses binary search on its local sorted list to partition it into p subsequences using the pivots received from Step 2. Then the jth subsequence is sent to P_j.
4. *Local merging.* Each processor merges the p sorted subsequences received to produce the ith column of the sorted array.

Our randomized sampling algorithm runs in $O((n \log n)/p)$ with high probability, using only two rounds of balanced communication and a broadcast of $p - 1$ elements from one processor to the remaining processors.[2]

Applications to computational geometry

Computational geometry is the study of designing efficient algorithms for computational problems dealing with objects in Euclidean space. This rich class of problems arises in many applications, such as computer graphics, computer-aided design, robotics, pattern recognition, and statistics. Randomization techniques play an important role in computational geometry, and most of these techniques can be viewed as higher-dimensional generalizations of Quicksort.[9] We'll look at the simplest example of such techniques, which is also related to the sample sorting procedure I just described.

Let N be a set of points on the real line R such that $|N| = n$. Sorting the n points amounts to partitioning R into $n + 1$ regions, each defined by an (open) interval. Let S be a random point of N that divides R into two halves. Let N_1 and N_2 be the subsets of N contained in these halves. Then, we expect the sizes of N_1 and N_2 to be roughly equal. As in the Quicksort algorithm, we can sort N_1 and N_2 recursively.

As a generalization, let S be a random sample of N of size s. One way to construct S is to choose the first element of S randomly from N and delete it from N. Continue the process, each time selecting a random element from the remaining set N and deleting it from N, until we have s elements in our set S. Such a procedure is called *sampling without replacement.*

Does S divide the real line R into regions of roughly equal size? As I stated earlier, the partition is defined by the set of open intervals. The *conflict size* of any such interval I is the number of points of N and I, but not in S. We would like to determine whether each interval's conflict size is $O(n/s)$ with high probability. Unfortunately, we can only show a weaker result—namely, that with probability greater than 1/2, each interval's conflict size is $O((n \log s)/s)$.

We can apply the same random sampling technique to the case where N is a set of lines in the plane. A random sample S generates an arrangement (that is, the convex regions in the plane induced by the lines in S). The conflict size is the number of lines in N, but not in S, that intersect a region of the arrangement of S. If we refine the

arrangement of S so that each region has a bounded number of sides, then with high probability, the conflict size of every region is indeed $O((n \log s)/s)$.

These two simple generalizations of the randomized Quicksort can be used to develop efficient algorithms for various problems in computational geometry.[9]

The basic strategy of Quicksort—randomized divide-and-conquer—represents a fundamental approach for designing efficient combinatorial algorithms that will remain a source of inspiration for researchers for many years to come. In particular, the full potential of the technique in handling higher-dimensional problems in computational geometry and in developing parallel algorithms for combinatorial problems is yet to be fully exploited. In the future, we can anticipate vigorous research progress along these directions.

References

1. C.A.R. Hoare, "Quicksort," *The Computer J.*, Vol. 5, No. 1, Apr. 1962, pp. 10–15.
2. D. Helman, D. Bader, and J. JaJa, "A Randomized Parallel Sorting Algorithm with an Experimental Study," *J. Parallel and Distributed Computing*, Vol. 52, No. 1, 10 July 1998, pp.1–23.
3. D.E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, Addison-Wesley, Reading, Mass., 1973.
4. R. Sedgewick, "Implementing Quicksort Programs," *Comm. ACM*, Vol. 21, No. 10, Oct. 1978, pp. 847–857.
5. M. Ben-Or, "Lower Bounds for Algebraic Computation Trees," *Proc. 15th Ann. ACM Symp. Theory of Computing*, ACM Press, New York, 1983, pp. 80–86.
6. P. Raghaven, *Lecture Notes on Randomized Algorithms*, tech. report, IBM Research Division, Yorktown Heights, N.Y., 1990.
7. W. Frazer and A. Mckellar, "Samplesort: A Sampling Approach to Minimal Storage Tree Sorting," *J. ACM*, Vol. 17, No. 3, July 1970, pp. 496–507.
8. G. Blelloch et al., "A Comparison of Sorting Algorithms for the Connection Machine CM-2," *Proc. ACM Symp. Parallel Algorithms and Architectures*, ACM Press, New York, 1991, pp. 3–16.
9. K. Mulmuley, *Computational Geometry: An Introduction through Randomized Algorithms*, Prentice Hall, Upper Saddle River, N.J., 1994.

Joseph JaJa is the director of the University of Maryland Institute of Advanced Computed Studies and a professor of electrical engineering at the university. His research interests are in parallel and distributed computing, combinatorial optimization, and earth-science applications. He received his MS and PhD in applied mathematics from Harvard University. He is a fellow of the IEEE. Contact him at the Inst. for Advanced Computer Studies, A.V. Williams Bldg., Univ. of Maryland, College Park, MD 20742; joseph@umiacs.umd.edu; www.umiacs.umd.edu/~joseph.