Rectangular Point Location in d Dimensions with Applications

H. EDELSBRUNNER,* G. HARING* AND D. HILBERT†

Rectangle location search in d dimensions is finding the d-dimensional axis-parallel box of a non-overlapping collection C that contains a query point. A new data structure is proposed that requires optimal space and $O(\log^d |C|)$ time for a search. The significance of this data structure in practical applications is substantiated by empirical examinations of its behaviour.

Received January 1984

1. INTRODUCTION

Given a plane graph G that subdivides some domain of the real plane, the *point location problem* asks for identification of the region that contains a query point. By virtue of a number of applications, point location is one of the best-examined problems in computational geometry.

The first algorithm that solves a special case in optimal – that is $O(\log n)$ – time is given by Dobkin and Lipton.[8] In this case G consists of n straight edges. The $O(n^2)$ space requirements were improved by Preparata[20] to $O(n \log n)$ and by Lee and Preparata[16] to $O(n)$ while sacrificing the optimal query time. Since Kirkpatrick[15] and Edelsbrunner, Guibas and Stolfi,[11] algorithms with optimal $O(n)$ space and optimal $O(\log n)$ query time exist. Generalisations of this special case to curved edges are considered by Edelsbrunner and Maurer,[10] who give a space-optimal solution. In fact, the space- and time-optimal algorithm of Ref. 11 applies to curved edges also settling, in some sense, the issue.

This paper extends the notion of point location to three and higher dimensions while restricting itself to Cartesian products of intervals as regions. The planar case of this version is also studied by Lipski and Preparata,[17] who applied the suboptimal algorithms of Ref. 20. We take a different approach and modify the algorithm of Ref. 10 to obtain a $O(n)$ space and $O(\log^2 n)$ query-time solution for n rectangles in the plane. Although suboptimal, we believe that its simple implementation makes it the proper choice in practical applications. Without accepting much complication, this solution is extended to $d \geq 3$ dimensions where $O(n)$ space and $O(\log^d n)$ query time are required for n regions.

The organisation of the paper is as follows. Section 2 introduces the new data structure along with a worst-case analysis of its behaviour. In Section 3 the results of empirical investigations are presented. They support the accuracy of the theoretical analysis as well as the usefulness of the data structure in practice. Section 4 examines Sections 2 and 3 from a practical viewpoint: Applications of the algorithm in analytic models of computer systems are discussed. Finally, Section 5 reviews the results obtained.

2. THE d-DIMENSIONAL SKEWER TREE

This section gives a thorough description of the problem considered and the data structure proposed for its solution. The concepts are explained for arbitrary positive dimensions and a running example illustrates the concept for the planar case.

We use the term *interval* to denote intervals that are closed to the left and open to the right. A *rectangle in d dimensions* (for some particular positive integer d) is the Cartesian product of d intervals, one on each coordinate axis.

Partially closed rectangles as described above can easily be used to define a *rectangular subdivision* of a rectangular domain, that is, no two rectangles of the subdivision intersect and the union of the rectangles is the domain.

The *rectangle location search problem in d dimensions* can now be stated as follows. Given a set of non-intersecting rectangles in d-dimensions, determine for a specified query point the rectangle in which it is contained.

The set of rectangles is stored in some data structure which allows us to answer queries efficiently. Thus a *solution* consists of a data structure for the set of rectangles, an algorithm that constructs the data structure, and an algorithm that determines for a query point the rectangle in which it is located. Such a solution is measured in terms of the amount of storage required by the data structure (called the *space*), the amount of time required for the construction of the data structure (called the *preprocessing time*), and the amount of time required to answer a query (called *query time*).

Let now S denote a set of n non-intersecting rectangles in d dimensions. Each rectangle is given as a $2d$-tuple of real values defining its corner points. E.g. $r = (r_1^1, r_2^1, \ldots, r_1^d, r_2^d)$ means that r is the Cartesian product of the intervals $[r_1^1, r_2^1), \ldots, [r_1^d, r_2^d)$. Figure 2.1 depicts a set of 13 rectangles in the plane. They are chosen to define a subdivision of a rectangular domain.

We now present a description of the so-called skewer tree for the set S of rectangles in d dimensions. The

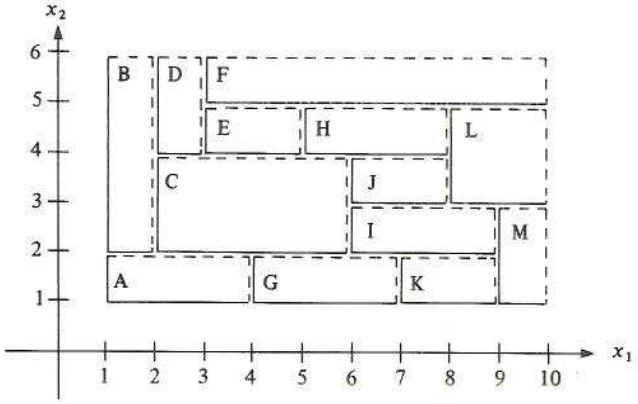

Figure 2.1. Rectangular subdivision in the plane.

* Institutes for Information Processing, Technical University of Graz, Schießstattgasse 4a, A-8010 Graz/Austria. (Address for correspondence.)

† Institute for Applied Mathematics, Technical University of Graz, Steyrergasse 17, A-8010 Graz/Austria.

algorithms for constructing the skewer tree and for answering a rectangle location query follow.

The skewer tree (of any dimension) for an empty set of rectangles is the empty tree. The *one-dimensional skewer tree* for a set of non-intersecting intervals on a line is the sorted list of the intervals (in sequential storage).

If $d > 1$ then we rename the endpoints of the rectangles' dth intervals as $e_1, e_2, \ldots, e_{2n}$ such that $e_i \leqq e_j$ if $i < j$. Let $m = (e_n + e_{n+1})/2$, and let S_1 contain the rectangles in S which lie to the left of the $(d-1)$-dimensional hyperplane $x_d = m$. Similarly, S_2 contains the rectangles in S which lie to the right of this hyperplane. Let S_3 contain all rectangles of S that intersect the hyperplane. Since each rectangle of S_3 intersects $x_d = m$, the projections of any two rectangles in S_3 on to $x_d = 0$ do not intersect. The root R of the *d-dimensional skewer tree for S* contains the value m which defines the dividing hyperplane. Its left subtree is the d-dimensional skewer tree for S_1 and its right subtree is the d-dimensional skewer tree for S_2. The middle subtree for R is the $(d-1)$-dimensional skewer tree for the orthogonal projection on to $x_d = 0$ of the rectangles in S_3.

Figure 2.2 shows the two-dimensional skewer tree for the 13 rectangles depicted in Figure 2.1. The upper-case letters A, B, ..., M stand for the pointers to the respective rectangles stored. A node of a skewer tree contains an integer which defines the dimensionality of the tree rooted at this node, a real value which determines the hyperplane, and three pointers to the left, middle, and right subtree. A one dimensional skewer tree stores one pointer per element.

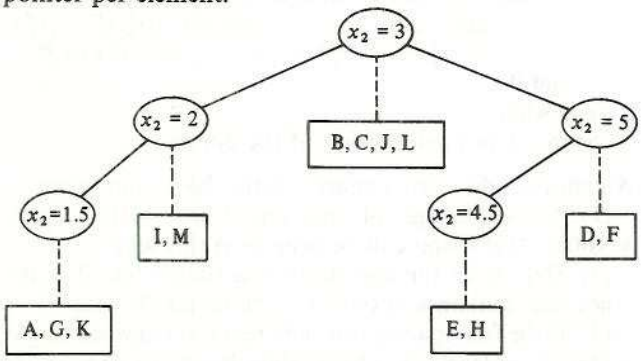

Figure 2.2. Two-dimensional skewer tree for 13 rectangles.

Lemma 2.1

The skewer tree for n non-intersecting rectangles in d dimensions requires $0(n)$ space.

Proof

Note first that each rectangle is stored exactly once in a one-dimensional skewer tree. Thus the space needed for those lists is $0(n)$. Each node of a k-dimensional skewer tree, for $2 \leqq k \leqq d$, which has no left or no right son has a non-empty $(k-1)$-dimensional skewer tree for its middle subtree. Since at least half of the nodes of a binary tree have no left or right son, we conclude that the space needed for the nodes with dimensionality k is at most proportional to the number of non-empty $(k-1)$-dimensional skewer trees involved. As all $(k-1)$-dimensional trees are disjoint, that is, no rectangle is stored in two of them, $0(n)$ space suffices for the nodes with dimensionality k, which completes the argument.

Lemma 2.2

The skewer tree for n non-intersecting rectangles in d dimensions requires $0(n \log n)$ time for its construction.

Proof

Since the algorithm for constructing the d-dimensional skewer tree works very much along the lines of the definition of the skewer tree given above, we omit its description. For the determination of the skewing hyperplanes and the splitting of the current set into three disjoint subsets, S_1, S_2 and S_3 so defined, we employ a median algorithm which requires $0(n)$ time for the determination of the median among n values (see e.g. Ref. 2).

Let v denote an arbitrary inner node of the skewer tree. Let n_v denote the number of rectangles stored in the subtree rooted at v. Those n_v rectangles are placed in $0(n_v)$ time into three disjoint sets S_1, S_2 and S_3 satisfying $|S_1| + |S_2| + |S_3| = n$ and $|S_1| \leqq n/2$ and $|S_2| \leqq n/2$. We define the *level* of a node as the distance of this node from the root of the skewer tree.

Each rectangle in the whole set is in at most one subtree rooted at a node with level 1. Hence the construction of all nodes with level 1 costs $0(n)$ time. Since a path from the root to a one-dimensional skewer tree contains at most $d-1$ pointers to a middle subtree, the height of the skewer tree is $0(\log n + d)$. Thus, considering d as a constant, $0(n \log n)$ time suffices for the construction of the inner nodes of a skewer tree. $0(n \log n)$ time suffices also for the construction of the one-dimensional skewer trees which completes the argument.

Lemma 2.3

A rectangle location query in a skewer tree for n non-intersecting rectangles in d dimensions can be answered in time $0(\log^d n)$ in the worst case.

Proof

The algorithm answers a d-dimensional rectangle location query by answering $0(\log n)$ $(d-1)$-dimensional queries. More specifically, it works as follows.

The algorithm starts at the root of the tree which discriminates w.r.t. the dth coordinate.

Let v denote the current node which discriminates w.r.t. the kth coordinate, where $2 \leqq k \leqq d$. The value stored in v is denoted by val (v) and defines the $(k-1)$-dimensional hyperplane $x_k = \text{val}(v)$.

Step 1

A $(k-1)$-dimensional query (considering the $k-1$ first coordinates of the query point only) is carried out in the middle subtree of v. This query yields a rectangle r (if any) whose projection on to the $k-1$ first coordinates contains the projection of the query point.

Step 2

Three cases, depending on the relative position of r and the query point $p = (p^1, \ldots, p^d)$, are distinguished.

Case 1

r contains q or r does not exist and $p^k = \text{val}(v)$. In this case the solution is found.

Case 2

p^k lies to the left of the kth interval of r, or if r does not exist, $p^k < \text{val}(v)$. Then the procedure is repeated with the left son of v as the current node (if it exists).

Case 3

p^k lies to the right of the kth interval of r, or if r does not exist, $p^k > \text{val}(v)$. Then the procedure is repeated with the right son of v as the current node (if it exists).

For the sake of completeness we note that a one-dimensional rectangle location query is answered simply by binary search. Let $Q_k(n)$ denote the time required for answering a k-dimensional rectangle location query in a skewer tree that accommodates n k-dimensional rectangles. Then $Q_1(n) = 0(\log n)$. For higher dimensions we have $Q_d(n) = 0(\log n)\,Q_{d-1}(n) = 0(\log^d n)$ (as the tree has height $0(\log n)$), which completes the argument.

We have now obtained the main result of this section, namely a solution for the d-dimensional rectangle location search problem.

Theorem 2.4

For a set of n non-intersecting rectangles in d dimensions there exists a data structure that requires $0(n)$ space and $0(n \log n)$ time for its construction such that $0(\log^d n)$ time sufficies to answer a rectangle location query. The assertion is an immediate consequence of Lemmas 2.1, 2.2 and 2.3.

In Section 3, it is argued and substantiated that the constants involved in the time and space requirements for the skewer tree are rather small. This fact and the conceptual simplicity of the structure make the skewer tree an interesting possibility in practical environments. It is worthwhile to note that the query time of the skewer tree can be improved to $0(\log^{d-1} n)$ in the worst case by application of the techniques described in Kirkpatrick[15] or, more recently, in Edelsbrunner, Guibas and Stolfi:[11] each two-dimensional middle subtree is replaced by an instance of the structure in Ref. 15 or Ref. 11. This improvement, however, sacrifices the conceptual simplicity of the structure and is, therefore, omitted in this study.

3. EXPERIMENTAL INVESTIGATIONS

This section describes experimental experiences in implementing the algorithms outlined in Section 2. We are particularly interested in examining the query time of the rectangle location search algorithm. Our intention is to test the algorithm on 'random subdivisions' of a given rectangular domain.

The generation of such a random subdivision is not straightforward. Therefore, we first give the methods used to generate the rectangular subdivisions for our experimental investigations. One objective criterion a random subdivision has to fulfil is the uniform distribution of the areas of the subdividing rectangles in a given interval.

A subdivision is generated by successively adding randomly selected rectangles to a set L of rectangles, until the whole domain is covered (subdivided). Each rectangle which is added to L starts at a so-called starting point. P is a set which contains the possible starting points. For a rectangle $r = (r_1^1, r_2^1, \ldots, r_1^d, r_2^d)$ its starting point is defined by $(r_1^1, \ldots, r_1^d)$. If we examine a rectangular subdivision in two dimensions as in Fig. 2.1, we see that the lower left corner (the starting point) of each rectangle is the upper left or lower right corner of another rectangle which already belongs to L. The only exception is the rectangle in the lower left corner of the domain. Generally if we look at a new rectangle $r = (r_1^1, r_2^1, \ldots, r_1^d, r_2^d)$ which is added to L, the d points, adjacent to the starting point of this rectangle $(r_2^1, r_1^2, \ldots, r_1^d)$, $(r_1^1, r_2^2, r_1^3, \ldots, r_1^d) \ldots (r_1^1, \ldots, r_1^{d-1}, r_2^d)$ are possible starting points for other rectangles of our subdivision, unless they lie on the boundary of the domain. Therefore the procedure for the generation of a random subdivision of a given domain has the following general structure. It starts with P containing only the lower left corner of the domain and L empty.

Basic algorithm

S_1 **While** P not empty **do**
S_2 Choose randomly a point $p \in P$ and remove it
S_3 **If** a new rectangle can start at p **then**
S_4 Choose a random rectangle with p as starting point;
S_5 **If** it stretches out of the domain **or** intersects a rectangle in L **then**
S_6 modify it
end if
S_7 Add the new rectangle $(r_1^1, r_2^1, \ldots, r_1^d, r_2^d)$ to L and add its d corners $(r_2^1, r_1^2, \ldots, r_1^d)$, $(r_1^1, r_2^2, r_1^3, \ldots, r_1^d)$, $\ldots$, $(r_1^1, \ldots, r_1^{d-1}, r_2^d)$ to P
end if
end while
S_8 **Stop** – L is a subdivision of the domain.

We should add some remarks to this basic algorithm.

(1) An advantage of this algorithm is that every possible subdivision can be generated this way.

(2) Step S_3 of the algorithm is a simple test if p lies inside the domain and outside each rectangle in L.

(3) If the free space is not sufficient for a new rectangle in step S_5 at least one of the edges has to be shortened. If possible the other edge is lengthened to avoid getting too many small rectangles. Fortunately for two dimensions the free space is always a rectangle as in Fig. 3.1 (*a*). The situation in Fig. 3.1 (*b*) is not possible, because the starting point of r cannot occur in P.

(4) To avoid having a new rectangle slightly smaller than the free space and therefore being forced to fill the

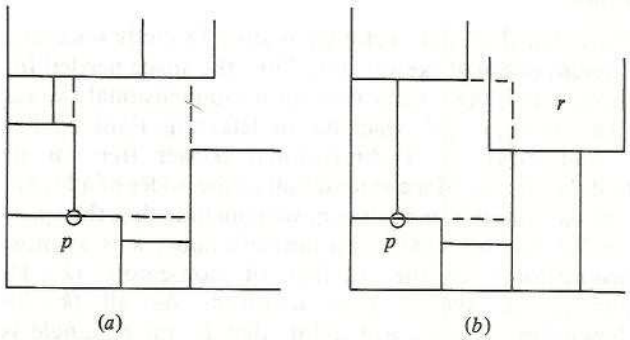

Figure 3.1. Free space for a new rectangle (*a*) in the plane; (*b*) in higher dimensions only.

remaining gaps with small and elongated rectangles, it is necessary to modify the algorithm to guarantee a minimal edge-length of the rectangles. For this purpose it is necessary to alter the rectangles in L in several cases. We skip the details of this modification.

(5) In the case $d > 2$ the difficulties arise where a rectangle is too big and has to be modified. Look at Fig. 3.1 (*b*). It is now interpreted as a plane through the starting point p parallel to two of the coordinate axes. The lower left corner of the rectangle r is only the projection of its starting-point, the point itself not lying in this plane. So we cannot exclude this case any longer. The approach mentioned in Remark (3) would be too expensive for more than two dimensions. Therefore to trim a new rectangle with edge-lengths $e^1, \ldots, e^d$ the following simpler method is used ($f^1, \ldots, f^d$ denote the adjusted edge-lengths).

For $i = 1$ **to** d **do**

T_1 Determine m^i, the maximal edge-length in x_i-direction of a rectangle with edges $f^1, \ldots, f^{i-1}$.

T_2 $f^i = \min(e^i, m^i)$

This loop replaces the steps S_5 and S_6 of the basic algorithm. The first two steps are illustrated in Fig. 3.2.

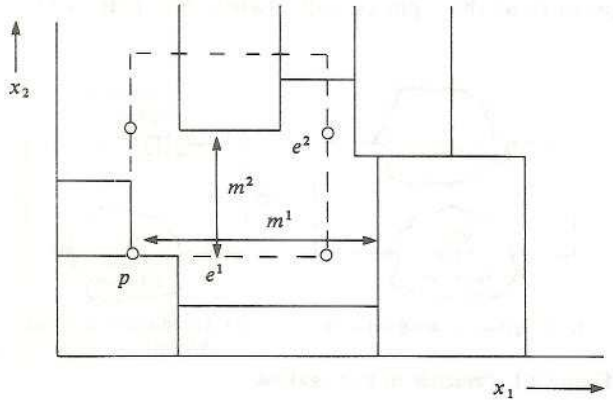

Figure 3.2. Modification of a rectangle – first two steps of the *d*-dimensional case. – – –, Initial, ○, modified rectangle.

As can be seen from Fig. 3.3, the results for two-dimensional subdivisions obtained by this algorithm do not differ from those obtained by the previous one. So we believe that the results are representative for higher dimensions too, and this justifies the simplification of our algorithm to solve the problems in higher than two dimensions.

Now we are ready to present some experimental results for the query time of the rectangle location search algorithm. We decide to use the number of comparisons between coordinate values as measure for the query time.

Thus we use the following test procedure:

R_1 **for** $i = 1$ **to** $n$sub **do**

(*a*) generate a subdivision with a given set of parameters

(*b*) build the skewer tree for this subdivision and determine the values

n: number of rectangles

k: number of nodes in the skewer tree

$C_{max}, C_{av.}$: maximum and average number of comparisons to find the rectangle

R_2 compute the average values of n, k, C_{max} and $C_{av.}$ for further evaluation.

The number of subdivisions $n$sub was limited by our computer resources. It varied between 5 and 100 depending on the number of rectangles in a subdivision.

To determine $C_{av.}$ we adopt the assumption that the query points are uniformly distributed over the whole domain. For an exact evaluation, the rectangular subdivision is refined to rectangles of invariant search-path in the skewer tree. Then one point out of each rectangle is examined and the number of comparisons is weighted with the measure of the rectangle. Obviously, one of the chosen points gives rise to the maximum number of comparisons which yields C_{max}.

Numerous computations for two, three and four dimensions have been performed while varying the shape of the domain and the way of choosing the random rectangles. For two dimensions we compared both algorithms to adjust a rectangle as described above. The results concerning the number of comparisons are presented in Figs 3.3, 3.4 and 3.5. As suggested by the theory, the results are depicted against $\log^d n$. The linear behaviour of C_{max} and $C_{av.}$ is well supported, the deviations due to various parameter combinations being very small.

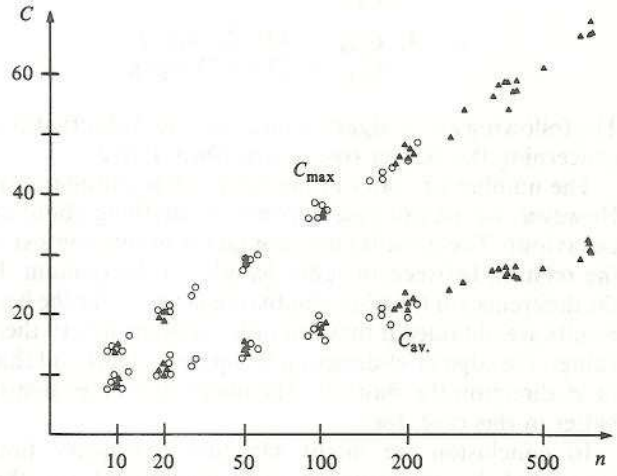

Figure 3.3. *C*-values for two dimensions against $\log^2 n$. △, Special algorithm for two dimensions; ○, general algorithm.

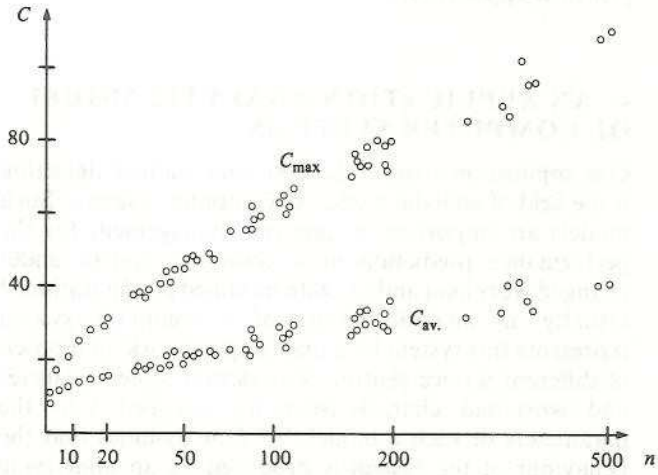

Figure 3.4. *C*-values for three dimensions against $\log^3 n$.

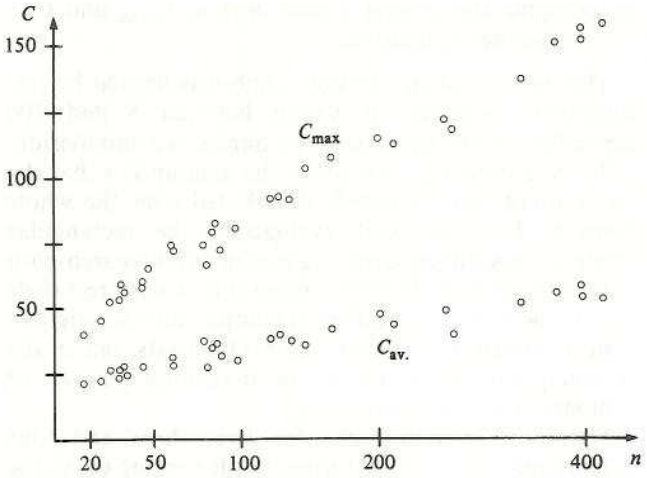

Figure 3.5. C-values for four dimensions against $\log^4 n$.

The following regression lines belonging to Figs 3.3–3.5 have been computed using common logarithms:

$$d=2:\ C_{max}=7+7.5\log^2 n$$
$$C_{av.}=5+3.2\log^2 n$$
$$d=3:\ C_{max}=21+4.8\log^3 n$$
$$C_{av.}=13+1.5\log^3 n$$
$$d=4:\ C_{max}=42+2.4\log^4 n$$
$$C_{av.}=22+0.72\log^4 n$$

The following are straightforward empirical observations concerning the skewer tree as described above.

The number of nodes in the trees varies considerably. However, we are not able to conclude anything about its behaviour. The most influencing factor in this context is the relation between the edge-lengths of the domain. If the differences of the edge-lengths are large, by far the best results are obtained if they are ordered according to their values, the edge in x^1-direction being the longest and that in x^d-direction the shortest. The query times are slightly better in this case, too.

In conclusion we might say that the query time (expressed by the number of comparisons) for this algorithm is rather low even for small values of n. The query algorithm is not complicated – if programmed efficiently (without recursion) it should be interesting for practical applications.

4. AN APPLICATION-ANALYTIC MODEL OF COMPUTER SYSTEMS

One application area for the above-mentioned algorithm is the field of analytic models of computer systems.[7] Such models are important in capacity management for the performance prediction of a computer system under changed workload and/or with modified configuration.[19] Usually an analytic model of a computer system represents this system by a queueing network, composed of different service centres. All relevant system features and workload characteristics are mapped into the parameters of such a model.[14,22] One assumes that the behaviour of the system is produced by an underlying Markov process. An important class of queueing networks is that for which the steady-state probability of the Markov process has product form

$$P(S_1,\ldots,S_K)=\frac{1}{G}\prod_{i=1}^{K}P_i(S_i)$$

$P(S_1,\ldots S_K)$ is the probability of a feasible network state $(S_1,\ldots,S_K)$ inqueueing network with K service centres. G is a normalizing constant, and $P_i(S_i)$ is a factor reflecting the probability that the service centre i is in state S_i. The question whether or not a network has product form solution can be decided on the nature of the queueing discipline and the service time distribution at the service centres of the network.[26]

Many realistic computer systems violate the assumptions for product form solution. If, in addition, the number of states is too big, and therefore a direct solution is intractable, simulation or approximation techniques must be used.[27] The most important approximation approach is the technique of aggregation (decomposition) based on Norton's theorem for electrical circuits.[4] With this technique a subnet of the total queueing network is replaced by a single service centre, a so-called composite queue with identical steady-state probability. One assumes that the composite queue behaves identically in the interaction with the rest of the network as the replaced subnetwork does (Fig. 4.1).

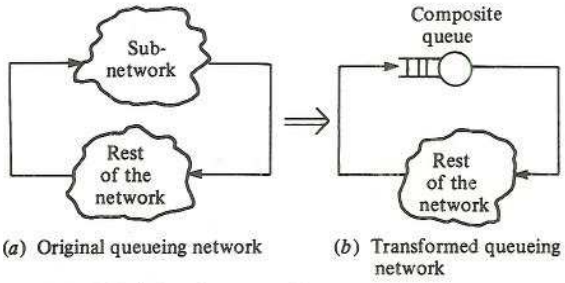

Figure 4.1. Principle of aggregation.

The composite queue with R different classes of jobs is defined by an R-dimensional, positive matrix H, the so-called rate matrix. The states of the composite queue in isolation are R-tuples $(n_1,\ldots,n_R)$, where n_r is the number of class r jobs in the composite service centre. For the further evaluation of the model,[3,21] the rate at which class r jobs are served in the state $S=(n_1,\ldots,n_R)$ is of interest, recognising that each class of jobs is receiving service simultaneously.[23]

$$\mu_r(S)=\frac{H(n_1,\ldots,n_r-1,\ldots,n_R)}{H(S)}$$

This aggregation technique is exact for product form networks. But furthermore, it can be used for many approximations where a heuristic approach is given (e.g. Ref. 5). It can be used for parametric analysis[4] where the parameters in the subnet are fixed, whereas those in the rest of the network are varied. Another important application is hierarchical modelling, where we have several levels of models with different degrees of accuracy, and the results of one level are used as queue characteristics in the model at the next higher level. This technique is often used to represent I/O-subsystems adequately.[12] The above-mentioned approach is also very

important for networks with passive resources, like central memory or peripheral processors, where the job flow through a subnetwork is limited by the availability of physical resources.[24] Very often parts of the system are not tractable analytically. Therefore, simulation or measurement techniques must be used to evaluate the rate function of a composite queue in an analytical model – leading to hybrid modelling techniques.[25] Further practical situations where approximation techniques must be used are multiple resource possession, blocking, parallelism between central and peripheral servers, distributions and queueing disciplines violating product form solution, state-dependent routeing, etc. (for further details see Refs 1, 6 and 13).

In all these situations, multidimensional rate functions of different structure in different regions of the R-dimensional domain are used to characterise the composite queue. The dimension d of the domain in the rectangle location search problem corresponds to the number of job classes R in the queueing network. The domain itself is defined by the range of the number of jobs in each job class. Usually this extends from zero to the maximum number of jobs N_r in each job class r. Therefore, the domain is given by $(0, \ldots, 0)$ and $(N_1, \ldots, N_R)$. The coordinate values are integers, namely the number of jobs in a specified job class. The rate functions themselves can be obtained either by analytic evaluation, simulation experiments, or general considerations and error-bound techniques as used in Refs 9, 18 and 28, where rational motivations can be used for analytic forms of the rate functions. The rectangular subdivision of the domain corresponds to regions in the domain where the rate function is of a different kind or was evaluated using different techniques. In the evaluation of the total network this rate function must be used repeatedly,[3, 21] delivering the rate for different subnetwork, populations $(n_1, \ldots, n_R)$. The rectangle location search problem is to find for a given subnetwork population $(n_1, \ldots, n_R)$ the corresponding rate function and to reference its value or evaluate the function in sequence. This search problem can be efficiently solved by the above-explained skewer-tree algorithm.

5. DISCUSSION

A new data structure, the so-called skewer tree, is introduced to solve the rectangle location search problem in d dimensions efficiently: find the axis-parallel box (or d-dimensional rectangle) of a non-overlapping collection that contains a query point. While extending the classical notion of point location from computational geometry to three and higher dimensions, the type of region is more restrictive. For a collection of n rectangles in d dimensions, the skewer tree takes $0(n)$ space and $0(\log^d n)$ time.

Empirical experiments support this theoretical analysis and also indicate that the constants involved are rather small. This together with the conceptual simplicity of the skewer tree makes us believe that it is the proper choice to solve rectangle location search in practice. The existence of this problem in practice is also supported by discussing applications in analytic modelling of computer systems.

REFERENCES

1. J. R. Agre, *Approximate Solutions to Queueing Networks with State-dependent Parameters*. Air Force Office of Scientific Research, Techn. Rep. TR-1092 (1981).
2. A. V. Aho, J. E. Hopcroft and J. D. Ullmann, *Data Structures and Algorithms*. Addison-Wesley, New York (1983).
3. S. C. Bruell and G. Balbo, *Computational Algorithms for Closed Queueing Networks*. Elsevier North–Holland, New York (1980).
4. K. M. Chandy, U. Herzog and L. Woo, Parametric analysis of queueing networks. *IBM Journal of Research and Development* **19** (1) pp. 36–42 (Jan. 1975).
5. K. M. Chandy, U. Herzog and L. Woo, Approximate analysis of general queueing networks. *IBM Journal of Research and Development* **19** (1) pp. 43–49 (Jan. 1975).
6. K. M. Chandy and C. H. Sauer, Approximate methods for analyzing queueing network models of computing systems. *ACM Computer Survey* **10** (3) 281–317 (1978).
7. P. J. Denning and J. P. Buzen, The operational analysis of queueing network models. *ACM Computer Survey* **10** (3) 225–261 (1978).
8. D. P. Dobkin and R. J. Lipton, Multidimensional searching problems. *SIAM Journal of Computing* **5** (1976), 181–186.
9. D. L. Eager and K. C. Sevcik, Performance bound hierarchies for queueing networks, Proc. SIGMETRICS Conf. ACM. *Performance Evaluation Review* **11** (4) 213–214 (1982).
10. H. Edelsbrunner and H. A. Maurer, A space-optimal solution of general region location. *Theoretical Computer Science* **16** (1981), 329–336.
11. H. Edelsbrunner, L. J. Guibas and J. Stolfi, Optimal point location in monotone subdivision (submitted for publication).
12. G. Haring and H. Schelch, On modelling RPS-disc systems (to appear in Computer Systems Science and Engineering).
13. P. Heidelberger and K. S. Trivedi, *Queueing Network models for Parallel Processing with Asynchronous Tasks*. IBM Research Report 9102 (1981).
14. M. G. Kienzle, *Measurements of computer systems for queueing network models*, University of Toronto Technical Report CSRG-86 (1977).
15. D. G. Kirkpatrick, Optimal search in planar subdivisions. *SIAM Journal of Computing* **12** 28–35, (1983).
16. D. T. Lee and F. P. Preparata, Location of a point in a planar subdivision and its applications. *SIAM Journal of Computing* **6** 596–606, (1977).
17. W. Lipski, Jr and F. P. Preparata, Segments, rectangles, contours. *Journal of Algorithms* **2** 63–76, (1981).
18. L. Lipsky, C.-M. H. Lieu, A. Tehranipour and A. van de Liefvoort, On the asymptotic behavior of time-sharing systems. *Comm. ACM*, 25, (10) 707–714, (1982).
19. T. L. Lo, Computer capacity planning using queueing network models. Proceedings, Performance '80 Conference, Toronto pp. 145–152, (May 1980).
20. F. P. Preparata, A new approach to planar point location. *SIAM Journal on Computing* **10** 542–545, (1981).
21. M. Reiser, Mean-value analysis and convolution method for queue-dependent servers in closed queueing networks, *Performance Evaluation* **1** 7–81, (1981).
22. C. A. Rose, A measurement procedure for queueing models of computer systems, *ACM Computer Survey* **10** (3), 263–280 (Sept. 1978).
23. C. H. Sauer and K. M. Chandy, *Computer Systems Performance Modeling*. Prentice-Hall, Englewood Cliffs (1981).
24. C. Sauer, Approximate solution of queueing networks with

simultaneous resource possession. *IBM Journal of Research and Development* **25** (6) 894–903 (Nov. 1981).
25. H. D. Schwetman, Hybrid simulation models of computer systems. *Communications of the ACM*, **21** (9) 718–723 (Sept. 1978).
26. K. S. Trivedi, *Probability and Statistics with Reliability, Queueing, and Computer Science Applications*. Prentice-Hall, Englewood Cliffs (1982).
27. J. Zahorjan, *The Approximate Solution of Large Queueing Network Models*. University of Toronto Technical Report CSRG-122 (1980).
28. J. Zahorjan, K. C. Sevcik, D. L. Eager and B. I. Galler, Balanced job bound analysis of queueing networks. *Communications of the ACM*, **25** (2) 134–141 (Feb. 1982).