View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Purdue E-Pubs

Purdue University

Purdue e-Pubs

Department of Computer Science Technical Reports

Department of Computer Science

2002

Multi-Dimenstional Traversals for Normalization Constants

Jorge R. Ramos

Vernon J. Rego
Purdue University, rego@cs.purdue.edu

Report Number:
02-031

Ramos, Jorge R. and Rego, Vernon J., "Multi-Dimenstional Traversals for Normalization Constants" (2002). *Department of Computer Science Technical Reports.* Paper 1549.
https://docs.lib.purdue.edu/cstech/1549

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries.
Please contact epubs@purdue.edu for additional information.

MULTIDIMENSIONAL TRAVERSALS FOR NORMALIZATION CONSTANTS

Jorge R. Ramos
Vernon Rego

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

CSD TR #02-031
December 2002

Multidimensional Traversals for Normalization Constants

Jorge R. Ramos and Vernon Rego*
Department of Computer Sciences
Purdue University
W. Lafayette, IN 47907
U.S.A.

Abstract

In many multidimensional systems, including models of computer and network systems and particle systems, the equilibrium distribution of state probabilities and other performance measures hinges on the explicit numerical determination of a normalization constant. When the normalization constant is the a sum of simple, computable functions defined on lattice-points in a constrained space, which is the case for a large class of problems, we show that the normalization constant can be obtained in a time that is directly proportional to the size of the space. We present an algorithm that is a tree-based variant of depth-first search which minimizes the usage of a stack, offering efficient execution for problems in many dimensions. Using simple complexity arguments as well as experimental results, we show that the algorithm offers run-times that are a many-fold improvement over previously proposed multidimensional recurrences.

Categories and subject descriptors: H.4.m. [Information Systems Applications]:Miscellaneous

General terms: Algorithms, Measurement, Performance.

Key Words and Phrases: Normalization constant, multidimensional system, polytope, and depth-first search.

*This work was supported by DoD DAAG55-98-1-0246 and PRF-6903235.

1 Introduction

Normalization constants hold the key to the solution of a host of problems pertaining to the design of systems based on dynamic entities (packets, jobs) which compete for resources (buffers, network links, processors). In the performance analyses of computer and communication systems, these critical constants appear in expressions for state probabilities in closed queueing networks. They are critical because the understanding of system behaviour hinges on state probabilities which, in turn, cannot be evaluated without knowledge of the appropriate normalization constants [1, 2]. Indeed, we motivate the basic problem using an example involving packet-loss probabilities in broadband networks [3], though the solution technique may be exploited in more general situations, i.e., whenever a discrete multidimensional probability needs to be evaluated over a well-defined region in n-dimensional space. Our proposed solution makes no assumptions other than that the contribution of each lattice-point in the normalizing space and the normalizing space both be well-defined.

Broadly speaking, there are two classes of methods that have been developed for the computation of normalization constants, more generally known as partition functions [4, 5]. When dimensionality (e.g., number of packets, queueing chains or resources) is small, the standard method is to exploit recursion [6], such as in the evaluation of convolutions [7], and when dimensionality is large, useful methods based on asymptotic expansions of integral representations have been very successful [8, 9]. A very different, and also highly successful approach, is the numerical inversion of the generating function of the partition function [10]. This technique has its roots in the early transform inversion work of Dubner and Abate [11]. An excellent survey and example of generating function inversion can be found in [12].

The approach we take is direct and algorithmic, without the complexity of numerical inversion or the exploitation of special structure in recursive schemes. In effect, we present an algorithm that views the space as a tree of lattice points and visits each lattice point in turn, without repetition. The advantage is simplicity and robustness. The method is applicable in very general situations, but a marked disadvantage is that its run-time complexity is directly proportional to the size of the space (i.e., number of lattice-points) over which the partition function is enumerated. Even so, the technique is a significant improvement over certain other methods, such as the multidimensional recurrence proposed in [13], and we have found run-times to be small for high-dimensional problems. The computational results serve as a reminder that direct, algorithmic approaches can be highly effective in many problems for which more intricate methods are chosen, perhaps often, as a matter of routine.

In brief, the space of interest is a convex polytope in n-dimensions, and the integration of probabilities over all lattice points in this space requires a time which is proportional to the number of such points in the space. A key point is that as the number of constraints defining the polytope increases, for n fixed, the number of lattice points tends to decrease, thus reducing the algorithm's run-time. Further, it is possible to prune the search so that traversal along any one dimension can stop whenever the contribution of lattice-points to the normalization constant is below some acceptable threshold. This offers a parameter-sensitive method for reducing run-time complexity.

2 Normalization Constants

We follow the conventions established in [3, 14] and consider a broadband network with m links, with link j having a capacity of C_j resource units, for $1 \leq j \leq m$. There are n distinct classes of calls supported by the network, and a call of class k is characterized by an offered load ρ_k and a bandwidth requirement of $r_{j,k}$ on link j, where the latter is zero when link j is not used by a class-k call. Let the vector $\mathbf{r}_j = (r_{j,1}, \cdots, r_{j,n})$ denote the bandwidth requirements of the distinct classes on link j, $1 \leq j \leq m$, and let the state of the system be represented by the vector $\mathbf{X} = (X_1, \ldots, X_n)$, where component X_k captures the number of class-k calls in progress.

The system can be seen to move between states in the set $\mathbf{S} = \{\mathbf{x} \mid \mathbf{x} \geq 0\}$, where each component x_i of the state vector $\mathbf{x}$ belongs to the set of nonnegative integers. If a call is always accepted whenever there is capacity to handle the call, and blocked calls are always cleared, then under appropriate assumptions on the nature of the call-arrival process, the probability that the system is in a given state $\mathbf{x}$ is obtained as

$$P\{\mathbf{X} = \mathbf{x}\} \quad = \quad \prod_{k=1}^{n} e^{-\rho_k} \frac{{\rho_k}^{x_k}}{x_k!} \tag{1}$$

provided that link-capacities are infinite. Let $Z_{j,k} = r_{j,k} X_k$ be a random variable representing the occupancy level of link j, due to demands exercised by calls of type k, so that

$$P\{Z_{j,k} \ = \ z\} = e^{-\rho_k} \frac{{\rho_k}^{z/rj,k}}{(z/r_{j,k})!} \tag{2}$$

whenever $r_{j,k} > 0$ and $z = ir_{j,k}$ for some nonnegative integer i, and the probability is defined as zero otherwise. When link capacities are finite, complications arise because the state-space becomes restricted. That is, the set of permissible states shrinks to

$$\mathbf{R} = \{\mathbf{x} \in \mathbf{S} \mid \forall j : \mathbf{r}_j \cdot \mathbf{x} \leq C_j\} \tag{3}$$

where $\mathbf{r}_j \cdot \mathbf{x} = \sum_k r_{j,k} x_k$ for $1 \leq j \leq m$. The steady-state distribution must now be normalized by conditioning on the probability mass of the constrained space $\mathbf{R}$, so that the probability that the constrained system is in state $\mathbf{x}$ at equilibrium is given by

$$\pi(\mathbf{x}) \quad = \quad \frac{1}{G} \prod_{k=1}^{n} e^{-\rho_k} \frac{{\rho_k}^{x_k}}{x_k!} \quad = \quad \frac{1}{G} \prod_{k=1}^{n} f(x_k, \rho_k) \tag{4}$$

where $f(x_k, \rho_k) = e^{-\rho_k} {\rho_k}^{x_k} / x_k!$, and the value of the normalization constant

$$G \quad = \quad \sum_{\mathbf{x} \in \mathbf{R}} \prod_{k=1}^{n} e^{-\rho_k} \frac{{\rho_k}^{x_k}}{x_k!} \tag{5}$$

is what we are interested in explicitly determining. Once the value of G is known, a number of quantities of interest can be computed — such as, for example, queue-size distributions or blocking probabilities for the system. For example, the set of blocking-states for a class-k call is given by those states in $\mathbf{R}$ that violate capacity, i.e.,

$$\mathbf{S}_k = \{\mathbf{x} \in \mathbf{R} \mid \exists j : \mathbf{r}_j \cdot (\mathbf{x} + \mathbf{u}_k) > C_j\} \tag{6}$$

where $\mathbf{u}_k$ is an n-vector with a 1 in the k-th component and zeros elsewhere. If Y_k is a Bernoulli random variable which takes on the value 1 when a class-k call is blocked and takes on value 0 otherwise, the blocking probability of a class-k call is given by

$$P(Y_k = 1) \quad = \quad \sum_{\mathbf{x} \in \mathbf{S}_k} \pi(\mathbf{x}) \quad = \quad \frac{P\{\mathbf{X} \in \mathbf{S}_k\}}{P\{\mathbf{X} \in \mathbf{R}\}} \tag{7}$$

The set $\mathbf{R}$ defined in Equation (3) is responsible for complicating the space over which the sum of products in Equation (5) is evaluated, for without such a complication, approximation and other convolution schemes may be exploited [7]. In effect, the normalization constant G is a sum of joint probabilities over the restricted space $\mathbf{R}$ in n-dimensions. We now focus on the space of interest, generated by the system of constraints

$$\sum_{k=1}^{n} r_{j,k} \cdot x_k \leq C_j \qquad j = 1, \cdots, m \tag{8}$$

yielding a nonnegative and bounded region because $r_{j,k} \geq 0$, $\forall j, k$. In the remainder of the paper we present a tree-based search scheme that computes normalization constants, such as the one defined in Equation (5), by a systematic traversal of lattice-space. The algorithm

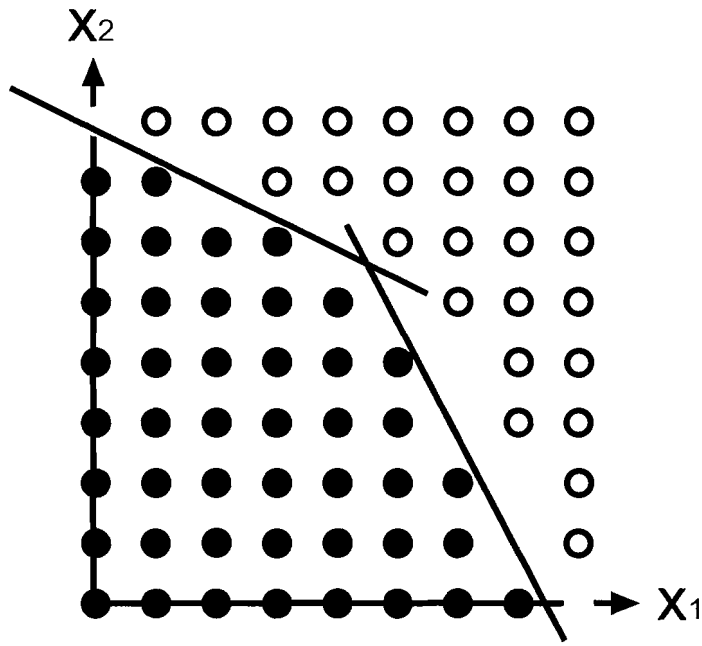

Figure 1: Example of a sample space for two dimensions

is insensitive to the particular form of the normalization, though its run-time depends on the constrained lattice-space in Equation (3). We begin by presenting basic depth-first search schemes and a significant modification that results in an efficient tree-search. We present the algorithm in detail, along with an example, and conclude with a brief report of our experiences with its run-time performance.

3 Basic depth-first search

Consider an algorithm that covers the space $\mathbf{R}$ by systematically visiting each lattice point in the set. At each point, properties of the point may be considered and its contribution toward the sum making up constant G evaluated. Starting from a well-defined location within $\mathbf{R}$ — the origin, for example — lattice points may be visited in a certain order, at each step verifying that the lattice point being considered is indeed a member of $\mathbf{R}$. This check is simple and entails verifying that the coordinates of the point in question satisfy the constraints in (8). When all points in $\mathbf{R}$ have been visited, the algorithm terminates and returns the computed value of G.

A simple algorithm for traversing the space $\mathbf{R}$ is a recursive depth-first search [15, 16]. The algorithm starts from a point inside the sample space (e.g., the origin) and then recursively visits each of its neighbours, moving in all possible directions. A flag may be

used at each location to ensure that each point is visited only once. The pseudo-code for this algorithm is given in Figure 2, where the number of dimensions n is represented by ndim.

```
DFS (v) {
        Mark vertex v as ``visited'';

        for (direction = 1; direction ≤ ndim; direction++){
             Increment G by normalization-function f(v);
             v' = v + unit_vector(direction);

             if (v' ∈ R and v' is ``not visited'') then
                          DFS(v');

        }
}

main() {
        Initialize all vertices as ``not visited'';
         G = 0.
        DFS(origin);
}
```

Figure 2: Recursive dfs

The nonrecursive variant of this algorithm can be implemented using a stack. Starting from an interior point, the algorithm visits lattice-points along one dimension, repeatedly pushing each point onto a stack, until it arrives at the boundary and has exhausted all points in this direction. Points are then popped off the stack, one by one, and the process is repeated using each point as a starting point, until the entire space has been covered. The algorithm ends when there are no more points to push onto the stack, and the stack is found empty.

Besides avoiding the inefficiencies of recursion, this algorithm has the advantage that each point is approached from only one possible direction, since the stack implicitly keeps track of directions, ensuring that lattice-points are visited in order. If each point is viewed as a node in a graph, and edges define a nearest-neighbour relation, the effect is to reduce

the number of edges between the lattice points and eliminate the need for a "visited" flag. Pseudo-code for the nonrecursive version is given in Figure (3).

```
v = origin
direction = 1
not_exit = TRUE;

while (not_exit) {
        if (v is in R){
                Increment G by normalization-function f(v);
                if (direction < ndim)
                        PUSH(v, direction + 1);
                v = v + unit_vector(direction);
        }
        else {
                if (POP (v, direction) == EMPTY)
                                not_exit = FALSE;
        }
```

Figure 3: Nonrecursive dfs

While both algorithms are simple, they suffer from performance problems caused by their huge reliance on stack space. Indeed, each requires as much stack space as the maximum number of points encountered over all directions traversed. For example, a simple graph consisting of a single line in a single dimension, but with 10^6 lattice points along that line, would generate a stack requirement of 10^6 stack cells. In actuality, both algorithms are variants of classical depth-first search in the sense that both exploit the structure of the search-space. While the classical algorithms do not rely on any sense of "direction" during traversal, the above algorithms start at a special lattice-point and travel only in increasingly positive directions. It is not necessary to mark points that have been visited, though we leave the marking procedure in-place because of its association with classical depth-first search.

Because the space $\mathbf{R}$ is both non-negative and convex, and bounded by coordinate planes in n-dimensions, it is possible to modify the depth-first traversal and make it considerably more efficient. In the following section we propose a search algorithm that makes more

efficient use of the stack.

4 Depth-First Tree Search

We propose a modification to the standard depth-first search algorithm to provision it with certain desirable characteristics. The algorithm must be non-recursive, exploit optimizations to minimize the number of graph-edges traversed, and ultimately reduce usage of the stack to a minimum. The optimization will be achieved by exploiting the special property of the set $\mathbf{R}$ and, for that purpose, we formalize some concepts that were introduced only informally in the previous two algorithms.

Instead of using a generalized graph to represent the neighborhoods of lattice-points in $\mathbf{R}$, we use a tree. We will construct a tree such that each lattice-point in the tree is visited just once. The origin is taken to be both the root of the tree and the starting point of the traversal; movement is always in a positive direction, and one lattice-point is consumed at each step. All edges in the tree will be the unit vectors of the given dimension n, and vertices of the tree will be the lattice-points that are connected by edges. Since only positive values are considered for the unit vectors, all edges are presumed to be directed edges and there are no back-edges. Finally, the tree is constructed in such a way that each point is reachable only via a single edge. That is, each point can be approached from only one direction. This minimizes the number of edges traversed while searching for points not yet visited, consequently eliminating repeated visits to lattice-points.

The tree T representing lattice-points in $\mathbf{R}$ is constructed as follows. Initially, T is defined to be empty. In the very first step the origin in $\mathbf{R}$ is added to T and taken to be the root of the tree. Next, starting from a lattice-point (vertex) v already in the tree (e.g., the origin) one edge E $(v \rightarrow v')$ and one vertex v' are added to the tree T, *if* the traversal allows a move from lattice-point v to lattice-point point v' in $\mathbf{R}$. Repeating this procedure recursively for all lattice-points and each permissible direction, the tree T is eventually completed. In essence, T is a tree because each vertex acts as the root of a new sub-tree. Clearly, the order in which dimensions are traversed, and the allowable directions, ultimately determine how the tree is built — a procedure that depends on both the algorithm as well as $\mathbf{R}$. The permissible directions are made to depend on the vertices, so that the result of the traversal is a tree.

The concept of degree is important for understanding the algorithm, and so we present the following definitions.

- `Degree of a vertex`: The number of edges emanating from a vertex. Recall that we use directed edges. By definition, only one edge reaches into a vertex, though many edges reach out of it.
- `Degree of an edge`: The direction of the edge.
- `Degree of a tree`: The degree of the root of that tree.
- `Degree of a branch`: A branch is a collection of consecutive lattice-points possessing the same degree. The degree of a branch will be the degree of its lattice-points.

The Algorithm

When the algorithm runs, it implicitly builds tree T. It utilizes the following rules to assign each lattice-point a degree when it visits the point:

- `Degree of root` = ndim.
- `Degree of a vertex`: The direction of the edge reaching the vertex. Observe that this edge is unique. This also defines the number of edges emanating from the vertex.
- `Degree of the edges`: The edges emanate in up to K distinct directions, $1, 2, \ldots, K$, where K is the degree of the starting vertex. The degree simply the direction.

An illustration of vertex and edge degrees in three dimensions is given in Figure 4, based on the definition of degree and the rules presented above. The resulting tree is of degree 3. Observe that there are unseen edges present at *boundary vertices*, which are vertices lying outside but are neighbours of vertices in $\mathbf{R}$. Boundary vertices have 3 edges. This also applies for vertices of degree 1 and 2 at the boundary.

During traversal, when the algorithm encounters a vertex of degree K, $K > 1$, it proceeds to traverse the edges emanating from the vertex in the natural order $1, 2, \ldots, K$. While the degree of a vertex is the number of edges emanating from the vertex, it also the label of the direction that must be traversed from that vertex. A stack is used to guarantee that traversal occurs in this specific order. Upon finding lattice-points of degree greater than one, along traversed edges, the algorithm invokes the above procedure recursively, storing points that have already been encountered in the stack alongside the additional information detailed below. The algorithm repeatedly compares the degree of a point to the direction being traversed, stopping only when the K-th direction has finally been traversed. Direction

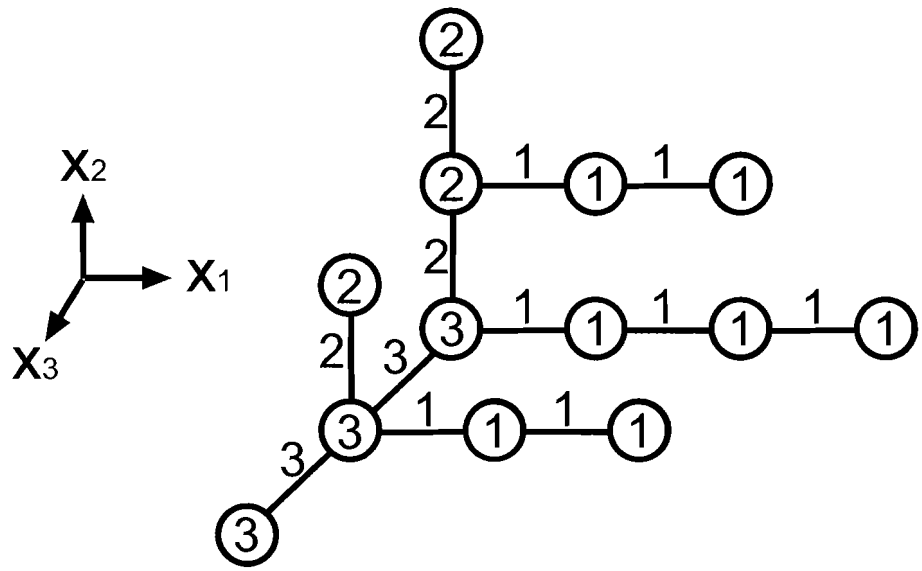

Figure 4: Degree of root, edges, and vertices

1, a special case, is traversed without pushing any point onto the stack. By following these rules for traversal, utilization of the stack is kept to a minimum.

The stack, which enables the traversal to be done nonrecursively, is used to store records containing the following items:

- The coordinates of a lattice-point.
- The direction, or edge, that is going to be traversed, after popping a lattice-point off the stack.
- The degree of the lattice-point.

The stack stores a lattice-point with multiple edges (i.e., degree greater than one) and the next direction to be traversed by the algorithm when this point is popped off the stack. Whenever a boundary of $\mathbf{R}$ is met, a lattice-point is popped off the stack and the next edge of that lattice-point is considered for traversal. This procedure is repeated until all edges have been traversed. The degree of the lattice-point is stored on the stack to enable simple detection of a termination condition. The root is a special case that is handled at the very start of the algorithm — by pushing it onto the stack, if the tree possesses a degree greater than one.

As an illustration, Figure 5 shows the implicit tree that is obtained when the sample space of Figure 1 is traversed. Although the traversal algorithm doesn't explicitly construct

and return a tree, the implicit but transient tree built during the search is critical for visiting lattice-points efficiently. Thus, the tree is merely a series of edges and vertices, where leaves are lattice-points outside **R** but neighbors of points in **R**. In Figure (5), these are the grey vertices lying immediately beyond the boundary of **R**, as seen from within the set. Although this is not shown, recall that the permissible directions are the unit vectors of $x1$ and $x2$. The pseudo-code for the tree-based algorithm is given in Figure 6.

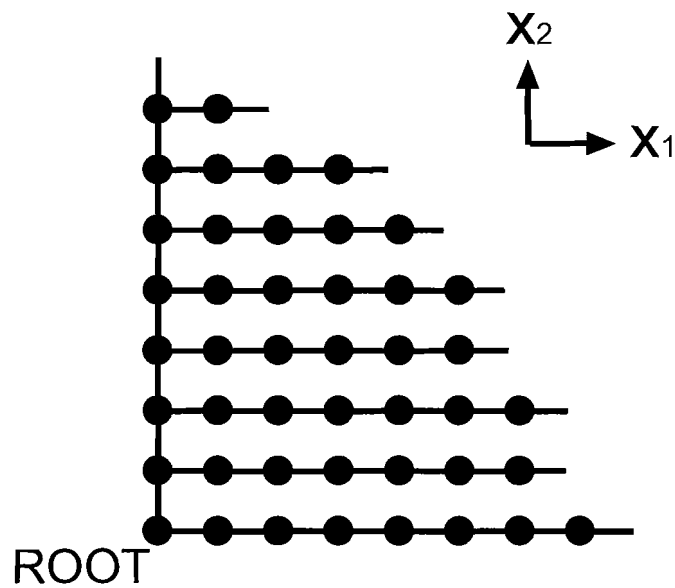

Figure 5: Tree defined by the space shown in Figure 1

4.1 A Sample Traversal

Consider the setup shown in Figure 5. The algorithm begins with `curr_dim = 1`, `ndim = 2`, and v = (0,0). The vertex/edge and degree record [(0,0), 2, 2] is pushed onto the stack, and lattice-points (1,0) through (8,0) are traversed, one by one. Lattice-point (8,0) lies immediately outside the set specified by the constraints, and so lattice-point (0,0) is popped off the stack, along with `curr_dim = 2` and `degree = 2`. Because `next_dim = 3`, which is greater than `degree = 2`, the point (0,0) is not pushed onto the stack again.

At the next step, lattice-point (0,1) is obtained and found to lie inside **R**. Because `curr_dim = 2`, which is greater than 1, the record [(0,1), 2, 2] is pushed onto the stack. Next, `curr_dim` is set to 1, and lattice-points (1,1) through (7,1) are traversed. Because (7,1) lies outside the constrained set, lattice-point (0,1) is popped off the stack, with `curr_dim = 2` , `degree = 2` and `next_dim = 3`. Point (0,1) is not pushed again onto the stack.

```
v = origin;
not_exit = TRUE;
degree = ndim;

curr_dim = 1;

        G = normalization-function f(v);
        // this pushes the root point in the stack if necessary
        if (ndim > 1)
                PUSH (v, 2, degree);

        while (not_exit){
        // go in direction curr_dim -- edges and vertexes have degree curr_dim
                v = v + unit_vector(curr_dim);

                if (v is in R){
                        Increment G by normalization-function f(v);
                        // always force a start with direction 1 (next_dim = 2)
                        if (curr_dim > 1){
                                PUSH(v, 2, curr_dim);
                                curr_dim = 1;
                        }
                }
                else {
                        if (POP (v, curr_dim, degree) == EMPTY)
                                                        not_exit=FALSE;
                        else {
                                next_dim = curr_dim + 1;

                                // push on stack for later direction
                                if (next_dim ≤ degree)
                                        PUSH(point, next_dim, degree);
                        }
                }
        }
```

Figure 6: Tree-based lattice traversal

Lattice points (0,2) through (0,7) are processed in a similar manner, so that neighboring points of degree one are always traversed. Lattice-point (0,8) is determined to be at the boundary, since it lies outside the constrained set. At this point the stack is empty, and the algorithm terminates.

Stack Size

Consider the building of a tree of degree D. In the worse case, the algorithm pushes a sequence of lattice-points of degree D, $D-1$, $D-2, \ldots, 3, 2$ onto the stack. Direction 1 is not pushed, but traversed. That means that the size of the stack thus generated is bounded from above by $D-1$, i.e., at most the number of dimensions minus one.

Proof of Correctness

Up to this point, we focused on minimizing the processing of lattice-points in terms of edge-traversals and stack-space. We now need to prove that the algorithm touches all lattice-points in the set $\mathbf{R}$. For `ndim = 1`, it is clear that the entire constrained space is traversed, because the algorithm systematically moves from the origin, along direction 1, up until the boundary of the graph. The result is a tree of degree 1.

Now assume that the algorithm covers all of the lattice-points given by a tree of degree D. If we build a tree of degree $(D+1)$, the algorithm proceeds by constructing a branch of degree $(D+1)$ in which all vertices are roots of trees in dimension D. The branch of degree $(D+1)$ starts at the origin and goes in direction $(D+1)$, up until the boundary for dimension $(D+1)$. This effectively enables all the lattice-points in that direction to be covered. Since those points are the roots of trees of degree D, and they cover all points of dimension D, all points in the set $\mathbf{R}$ are covered.

Upon effectively building a tree of degree $(D+1)$, the algorithm adds one branch that ties together trees of degree D. This idea is illustrated in Figure 7, for a tree of degree 3. In turn, trees of degree 2 are constructed by a branch that ties together trees of degree one.

Complexity

Let $\mathbf{R}^*$ be the set of lattice-points $\mathbf{x} \in \mathbf{R}$ for which equality is attained for at least one constraint in the system given in (8), and define the boundary $b(\mathbf{R})$ to be the set of lattice-points that are not in $\mathbf{R}$ but are neighbours of points in $\mathbf{R}^*$. Let d be the number of lattice-points in $\mathbf{R} \cup b(\mathbf{R})$. During the traversal of any set $\mathbf{R}$, the algorithm must touch points in $b(\mathbf{R})$ while querying lattice-points for membership in $\mathbf{R}$. At the boundary it is

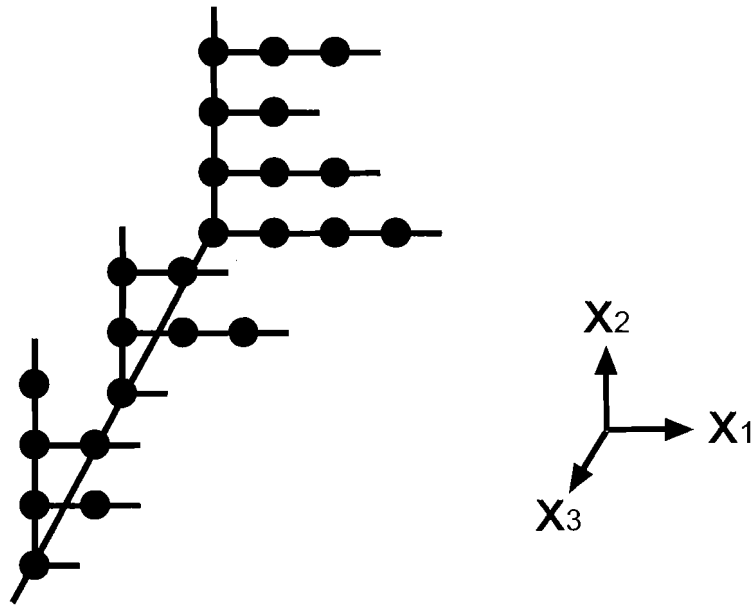

Figure 7: Recursive definition of a degree-3 tree

able to verify that it has indeed arrived at a limit in some direction. Considering stack processing, each lattice-point is visited a number of times that is equal to its vertex degree, and is also equal to the number of edges emanating from that vertex. This includes edges reaching outside the constrained set for boundary points. Thus, the run-time $g(d)$ of the algorithm is given by

$$g(d) \in O(\sum v_i E(v_i)) = O(E) \tag{9}$$

from where, by associating each lattice-point with the edge used to reach that point, we can conclude that $d = E - 1$. This is because each lattice-point is approached only once, so that every point has one approaching edge, except for the origin which is the start point and has no approaching edge.

Thus, the run-time complexity is given by $g(d) = d$, implying that the execution time is directly proportional to the number of points in the constrained set $\mathbf{R}$. Indeed, this appears to be a marked improvement over the generating function-based recursion proposed in [13] which offers a run-time complexity of $O(n \prod_{j=1}^{m} C_j\ )$ and a space complexity of $O(\prod_{j=1}^{m} C_j)$. In contrast, observe that the stack-space requirement of the tree-search algorithm is bounded from above by $n - 1$.

Direct application of the classical depth-first search algorithm is impractical because it

indiscriminately pushes new points onto the search stack. This is a serious limitation because it can cause the stack to grow rapidly, bounded only by d. The tree-based search, on the other hand, consistently moves in the same direction until the set boundary is reached, without pushing points onto the stack. When the algorithm traverses any direction other than direction 1, it moves at most one step in that direction, pushes the corresponding lattice-point onto the stack and then immediately begins a traversal of directions 1 through K. In this way, the algorithm forces the traversal to start at direction 1 each time such a "new" vertex is found, drastically reducing usage of the stack. Further, whenever the algorithm considers a vertex of degree D, the edges that will be traversed before popping that vertex off the stack will each have degree at most $D-1$. After the vertex is popped off the stack, the algorithm moves in the direction D. This guarantees the upper bound on stack requirements, as claimed above.

5 Experimental Results

While the complexity of the algorithm suggests that the procedure may not be feasible when the size of the constrained set $\mathbf{R}$ is prohibitively large, in terms of the number of lattice-points it contains, the procedure is feasible when (a) d is of reasonable size, (b) the number of constraints m is moderate or large, relative to the number of dimensions n, and (c) the parameters (i.e., ρ_k) along each dimension are such that the tree-building procedure enables run-time pruning, so that traversals along certain directions can be terminated when lattice-points yield meager contributions to the total sum G. In this section we present the results of a set of experiments showing how the size of an average constrained set varies with the dimension of the problem space. In each case the exact number of points in the lattice-space was determined explicitly, without any use of pruning.

Experiment 1.

We ran the traversal algorithm on randomly constructed constrained regions in n-dimensional space, where $2 \leq n \leq 100$. A total of 840 such runs were made using uniformly randomly generated constraints and coefficients, and run-times were measured and graphed against the total number of points. The constant G was computed using all the points in each space, and the number of points traversed ranged from 10 to 25×10^6, with the number of points within the randomly generated sets $\mathbf{R}$ ranging from 6 to 17×10^6.

In Figure 8 is shown the result of this run-time measurement. In applying a linear regression to the measured points, a correlation coefficient of 0.95 was obtained, suggesting

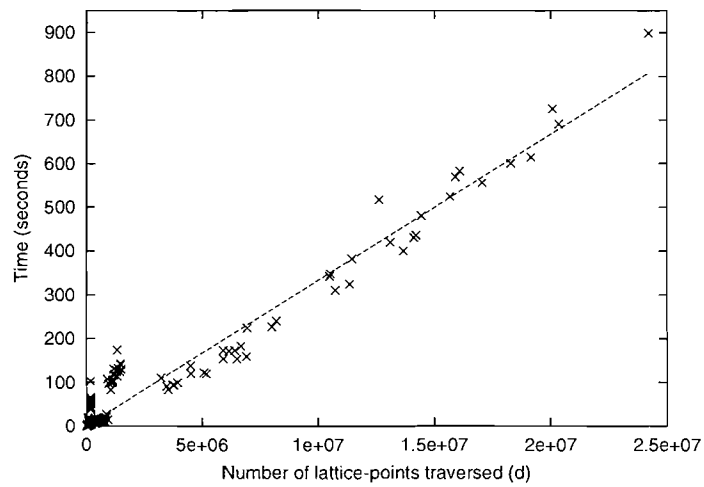

Figure 8: Run-time vs. average set-size $\overline{d}$

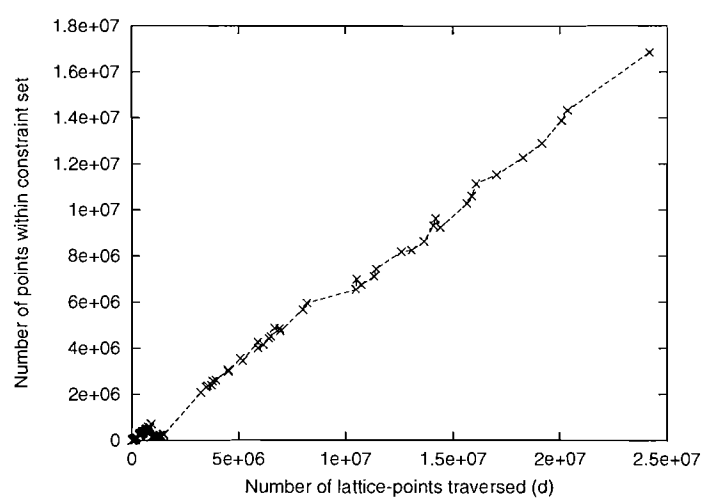

Figure 9: Correlation of d with $(d - |b(\mathbf{R})|)$

a run-time directly proportional to d. The result of an experiment showing the correlation between the total number d of points traversed and the subset of points lying strictly within the set $\mathbf{R}$ (i.e., $d - |b(\mathbf{R})|$), is shown in Figure 9. Again, this suggests that, on average, the number of points on the boundary is directly proportional to the number of points within the constrained set, with the constant of proportionality being approximately 1/2. While the ratio $(d - |b(\mathbf{R})|)/d$ can be surprisingly large in particular cases, the ratio seems to be about 1/2 on average, and at present we have no explanation for this phenomenon.

The determination of the number of points d within an arbitrary set $\mathbf{R}$ is a nontrivial problem. Since the run-time complexity of the algorithm is $O(d)$, it is instructive to get a handle on how d varies with the parameters defining set $\mathbf{R}$, at least in some "average" sense. We accomplished this experimentally in the following manner. First we fixed $C_j = C$, for $1 \leq j \leq m$, without suffering any loss in generality as far as measures of run-time are concerned. Second, values of the constraint coefficients $r_{j,k}$ were generated uniformly randomly from integers in the set $\{1, \ldots, 10\}$, for $1 \leq j \leq m$, and $1 \leq k \leq n$. A single application of the depth-first tree algorithm on a randomly generated "problem-space" (i.e., one complete set of randomly generated constraint coefficients $r_{j,k}, \forall(j,k)$) offered a single observation of the algorithm's run-time, for fixed n and C. The independent generation of 30 such problem-spaces, with associated tree traversals, offered 30 observations from which an average, maximum, minimum and standard-error were obtained, for fixed n and C. The next step was to repeat the procedure for different values of n and C, in order to study the effect of dimensionality on run-time.

Experiment 2.

The objective here was to examine the relationship between dimensionality n and the number of lattice-points d the algorithm must traverse, on average. Runs were made for $n = 2$, 3, 5, 7, 8, 9, 11 and 12, with $C = 100$, and $m = n$, with average, maximum, minimum and standard deviation obtained over 30 independent runs on uniformly randomly generated constrained spaces, as before. The results of this experiment can be seen in Figure 10 and also in in Table 1, which presents the minimum (d_{min}) and maximum (d_{max}) sizes of the constrained sets, along with average size $\overline{d}$ and standard-error (σ_d) over 30 independent runs, for each value of n.

The last last-two columns of the table present relative efficiency ratios $\overline{\eta} = \overline{d}/nC^n$ and $\eta_{max} = d_{max}/nC^n$ for the run-times given by the worst case and average case complexity, respectively, of our tree-based search algorithm; here, nC^n is the actual number of operations

required by the generating-function based multidimensional recurrence presented in [13]. It is interesting to note that both ratios converge rapidly to 0 for relatively small but increasing n, with the average case converging slightly faster. The multidimensional scheme developed in [13] recurses up to C_j times, $1 \leq j \leq m$, where each C_j is a component in an n-vector; the tree-algorithm is less sensitive to these inequality bounds. Also, compare the stack-space complexity n of the tree-algorithm to the space complexity C^n of the recurrence in [13], which is a direct result of the need to utilize recurrence history in computing new values. We use C in the efficiency ratios because $C_j = C \ \forall j$, to simplify our experiments.

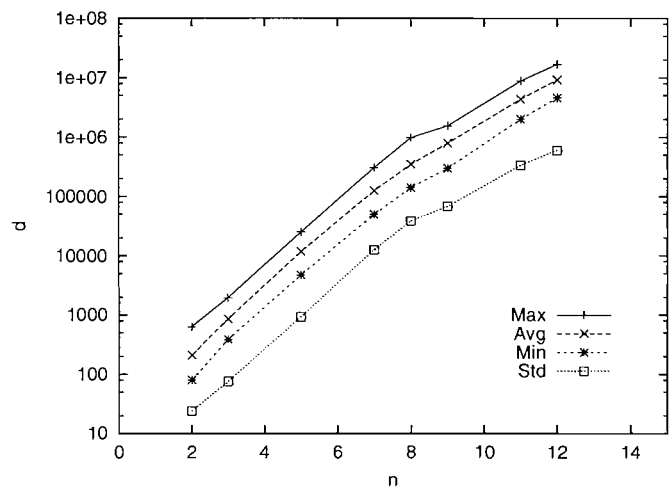

Figure 10: Number of points traversed vs. dimension n

n	d_{min}	d_{max}	$\overline{d}$	σ_d	$\overline{\eta}$	η_{max}
2	80	630	210.97	24.06	0.010548	0.0315
3	384	1953	847.23	76.21	0.000282	0.000651
5	4714	25271	11755.10	933.58	2.35e-07	5.05e-07
7	49813	305678	124870.77	12644.34	1.78e-10	4.37e-10
8	141599	985526	348653.33	38856.70	4.36e-12	1.23e-11
9	298454	1549614	792181.23	67874.05	8.8e-14	1.72e-13
11	2012261	8824529	4366967.33	336553.11	3.97e-17	8.02e-17
12	4599583	16895489	9223994.53	601605.51	7.69e-19	1.41e-18

Table 1: Statistics on n versus $\overline{d}$

Experiment 3.

The goal of this experiment was to examine the relationship between the average number of available resource units (C), the average size r of the constraint coefficients $r_{j,k}$, and the number of points in the constrained set, on average. We proceed by graphing the ratio $\gamma = C/r$ versus the number of points in the constrained set, for $\gamma = 5$, 10, 15, 20, 30, 40, and 50, $r = (1 + 10)/2 = 5.5$, and $m = n$. The results of this experiment can be seen in Figure 11, where the average number of points in a constrained set is graphed against dimension n.

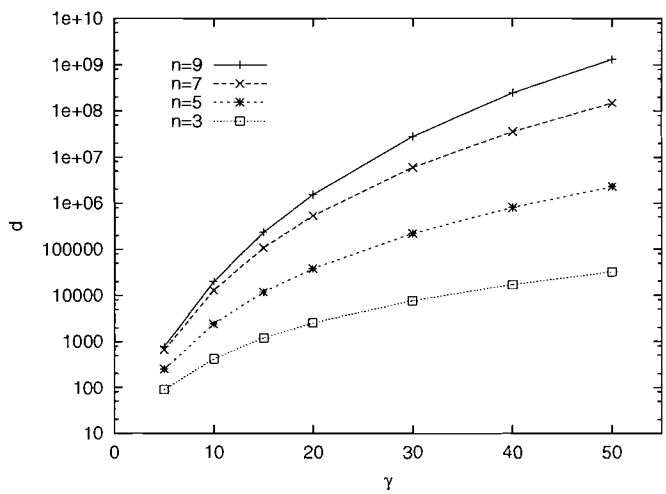

Figure 11: Number of points traversed vs. γ

Experiment 4.

The goal of this experiment was to examine the effect of an increasing number of constraints on run-time, for a fixed dimensionality n. That is, we measured the relationship between $\delta = m/n$ and the number of points in the constrained set, on average. Runs were made for a fixed value of $n = 8$, with γ ranging from 10 to 25 in steps of 5. The ratio δ took on the values 1/8, 1/4, 1/2, 1, 2, 3, and 4. The results of this experiment are shown in Figure 12, where the average number of points in a constrained set is graphed against selected values of γ, for increasing δ. Observe that beyond some point, an increasing number of constraints m reduces run-time complexity, but with diminishing returns.

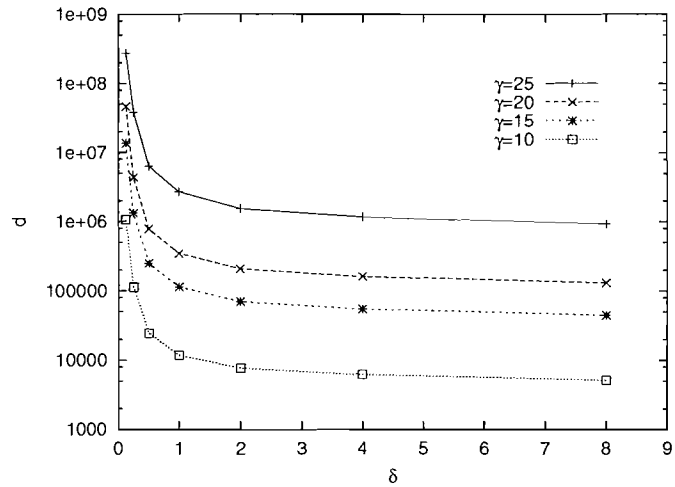

Figure 12: Number of points traversed vs. δ

6 Conclusions

We have found the proposed algorithmic procedure to be effective in computing normalization constants over fairly large spaces, on average, as indicated by our experiments. For example, computing the normalization constant over a space of about $d = 10 \times 10^6$ lattice-points takes roughly five minutes on a 336 MHz Sun enterprise-server. The advantage of using such a simple and direct procedure should not be overlooked, since the algorithm is robust and only needs the specification of function(s) to be evaluated, and the definition of the space over which a sum is to be computed.

Though the run-times offered by the tree-based algorithm are much smaller than those given by algorithms based on previously proposed multidimensional recurrences, we do not wish to overstate our case, since it is possible for some problems to offer truly large constrained spaces. When the number of lattice-points d in such spaces is prohibitively large, direct application of the algorithm may not be feasible. We are currently investigating pruning-based approximation schemes for such situations, and also techniques for better estimating run-time complexity of the algorithm for a given problem.

References

[1] J. J. Gordon. The evaluation of normalizing constants in closed queueing networksprogram transformations in a denotational setting. *Opns. Res.*, 38:863–869, 1990.

[2] P.Harrison. On normalizing constants in queueing networks. *Opns. Res.*, 33:464–468, 1985.

[3] K.W.Ross. *Multiserver Loss Models for Broadband Telecommunication Networks.* Springer-Verlag, London, 1995.

[4] F. Reif. *Fundamentals of Statistical and Thermal Physics.* McGraw-Hill, New York, 1965.

[5] Jr. D. McLachlan. *Statistical Mechanical Analogies.* Prentice-Hall Series in Materials Science, New Jersey, 1968.

[6] A.E.Conway. Recal: A new efficient algorithm for the exact analysis of multiple-chain closed queueing networks. *J.ACM*, 33:768–791, 1986.

[7] S. S. Lam and Y. L. Lien. A tree convolution algorithm for the solution of queueing networks. *Commun. ACM*, 26:203–215, 1986.

[8] J. McKenna and D. Mitra. Integral representations and asymptotic expansions for closed markovian queueing networks: normal usage. *Bell System Tech. J.*, 61:661–683, 1982.

[9] C. Knessl and C. Tier. Asymptotic expansions for large closed queueing networks with multiple job classes. *IEEE Trans. Computers*, 41:480–488, 1992.

[10] G. L. Choudhury, Ward Whitt, and D.M.Lucantoni. Numerical transform inversion to analyze teletraffic models. In *Proc. of the 14th Intl. Teletraffic Congress. The Fundamental Role of Teletraffic in the Evolution of Telecommunication Networks*, volume 1b, pages 1043–1052, Amsterdam, 1994. Elsevier.

[11] H. Dubner and J. Abate. Numerical inversion of laplace transforms by relating them to the finite fourier cosine transform. *J. ACM*, 15:115–123, 1968.

[12] G. L. Choudhury. Calculating normalization constants of closed queueing networks by numerically inverting their generating functions. *J. ACM*, 42:935–970, 1995.

[13] E. Pinsky and A. Conway. Exact computation of blocking probabilities in state-dependent multi-facility blocking models. In *Proc. of Performance of Distributed Systems and Integrated Communication Networks, IFIP Transactions*, pages 383–392. North-Holland, 1992.

[14] P. Lassila and J. Virtamo. Nearly optimal importance sampling for monte carlo simulation of loss systems. *ACM Trans. Modeling and Computer Simulation*, 10:326–347, 2000.

[15] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms.* MIT PRESS, McGraw Hill, Boston, MA, second edition, 2001.

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