A FRAMEWORK TO CREATE RESOURCE-BOUNDED NETWORK SERVICES

A Dissertation
Submitted to
the Temple University Graduate Board

---

in Partial Fulfillment
of the Requirements for the Degree of
DOCTOR OF PHILOSOPHY

---

by
Jaiwant Mulik
August, 2004

ABSTRACT

A FRAMEWORK TO CREATE RESOURCE-BOUNDED NETWORK SERVICES

Jaiwant Mulik

DOCTOR OF PHILOSOPHY

Temple University, August, 2004

Dr. Longin Jan Latecki, Chair

Active networks and network services hold promise to make the network infrastructure more flexible, providing an infrastructure for innovative applications. However, concerns related to resource consumption by network services at intermediate nodes is a major concern for practical deployment. Since typically more than one network service can be deployed at a network node, it is essential that in order to maintain useful services, requirements on node resources not exceed those that are available. In this document we present a two step technique to limit, with fixed arbitrary probability, demand for resources exceeding total available resources at a node. This technique can be applied to both component-based and ad-hoc service environments. In component-based environments our technique returns a single number such that as long as the end user uses only up to that number of components to create a service at a node, we can guarantee, with arbitrary probability that the resource consumption at that node will not exceed a given limit. Our technique is based on two observations: (a) Several component-based architectures exist that can be used to implement network services by combining components, and (b) such components cannot

be arbitrarily combined. Also, we present efficient counting algorithms and present simulations and experiments demonstrating the application of our algorithms. Some resource scheduling issues that affect per user throughput are also addressed.

ACKNOWLEDGEMENTS

I would like to thank my advisor Dr. Longin Jan Latecki for his support during this research, and to Dr. Jawahar 'Jay' Pathak for inspiring discussions, help with the algebraic setting and always providing ice-cold Pepsi™. Dr. Phillip Conrad provided initial support and was a major influence on my networking studies.

To my family.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Network services are services provided by switching devices within a network. The required network service of most routers is unicast forwarding. However, it has been shown [20, 3] that some end-to-end applications can greatly benefit from network services other than unicast forwarding. Dynamically adding such network services into network elements is the focus of the field of *active networking*. These services range from advanced switching functions such as content specific redirection of data [19] to non-switching functions such as data transformation (e.g. transcoding of multimedia streams) [6]. While active networks hold the promise to make network infrastructure more flexible, there are security concerns related to resource consumption by network services. Since typically, more than one network service can be deployed at a network node it is essential that node resources (CPU, memory and bandwidth) be shared "fairly" among the services. The exact definition of "fair" depends on local resource sharing policies.

We present techniques to assign costs to services and capacity to processing

nodes, and counting algorithms to provide probabilistic guarantees of resource availability to services. The rest of the report is organized as follows: This chapter provides references to related work and an outline of our approach to the problem. Chapter 2 formalizes the concept of sequence of components and presents formulae to compute the average cost of sequences. Chapter 2 also presents counting algorithms that help in determining probabilistic guarantees for end-users. Chapter 3 presents algorithms that improve upon those in Chapter 2 by reducing the domain of the problem. Chapter 4 presents simulation and experimental results demonstrating the application of our algorithms. Chapter 5 concludes this report with a discussion on future work.

1.1 Related Work

Most current approaches to manage resource consumption by network services can be categorized into the following categories:

- **TTL based** ([15, 31]): Here a Time-To-Live field in the active packet is an indicator of the total resources allocated to that packet. Each time the packet is processed, the TTL field is decremented, and discarded when the field value reaches zero. If processing the packet creates new packets, the initial remaining TTL value of the original packet is distributed over the newly created packets. Yamamoto et. al. [33] use a market based approach where each active packet carries a "budget" that is used to pay for resources at each node. The cost of a resource varies by current "market" demand.

- **Time based** ([17, 31]): This approach is well suited for memory resources whereby a

time limit is set on how long active services can use allocated memory [17]. Wetherall et. al. [31] limit the amount of CPU time an active packet can use.

- **Length** ([21, 14]): In this type, resource consumption is limited by estimating resource consumption of network services based on the length of the program. SNAP [21] guarantees that the resource consumption of a program is directly proportional to the length of the program with a known small constant of proportionality. While this approach has been proven to be simple to implement, the main drawback is that either all instructions have to use almost identical amount of resources or dummy instructions need to be introduced to maintain the proportionality between length of program and total resource consumption.

- **Centralized/Directory service** ([22, 25]): These approaches use centralized mechanisms that are aware of resources at active nodes. Typically these resources include both software (active code) and hardware resources. Before applications requiring network services can begin, the centralized systems compute the most efficient path and pre-allocate required resources.

- **OS scheduling** ([27, 23]): Ramchandran et. al. [23] use a packet scheduling scheme that attains fair resource allocation among multiple resources (CPU and bandwidth). In Schwartz et. al. [27] the virtual machine executing active packets runs in the control plane and depends on the native OS scheduling policy for resource allocation.

- **Direct measurement**([8]): This approach uses direct measurement of resources at a node and then translates those results to heterogeneous nodes, so that meaningful

resource consumption metrics and can be obtained at each node.

1.2 Our Approach

In this report we present another approach aimed at controlling overall resource consumption at a node. Our approach is based on the following observations:

1. There exist architectures [18, 4, 26] ([9] presents a survey) that, using a set of elementary components, allow construction of complex services at network elements. Such construction principles are common in other areas of computer science, e.g. software engineering, where simple to moderately complex reusable components can be combined to produce programs of greater complexity.

2. Such components *cannot* be arbitrarily combined. At least, the output of one component must be *compatible* with the input of the other. Compatibility is context dependent and can range from syntactic to semantics characteristics. CCML [28] a markup language that allows active nodes to constrain component composition based on hardware compatibility. For component-based systems in general, Whaley et. al. [32] use finite state models to extract interface models which can then be used by static checkers to determine composition constraints.

While all the above approaches try to deterministically limit access to resources in an effort to control overall resource consumption, we relax this approach and provide probabilistic guarantees on overall resource consumption. The immediate advantage of probabilistic guarantees is an increase in system utilization. The downside is that at times

some components might not be able to maintain required processing capabilities. Our approach can be used to enforce a resource control mechanism so that the demand on a node exceeds the available resources only with a fixed probability P. P allows the system administrator to set an acceptable tradeoff between system utilization and service performance.

Our overall strategy consists of two distinct steps:

1. We show formally (Sections 2.1-2.3), what is now intuitively probable, that if we assign a *cost* (some measure of resource consumption) to each component and impose a restriction on the maximum number of components that can be combined to create services, we can create services with known resource consumption characteristics.

2. Once the resource consumption characteristics of services are known, then given the number of users and system capacity, we estimate (Sections 2.4-2.6) the probability with which demand might exceed capacity.

There are two ways in which the above strategy can be used. In component-based environments and in ad-hoc services environments. We say a service environment is ad-hoc when there are no components available to end-users to create services, but fixed services are already available [17].

1.2.1 Component-Based Environments

In component-based environments, we use Step 1 above to determine the average cost of all services created using l components for each l in some range $1 \leq l \leq L$. Then we use Step 2 to determine the probability of overall resource consumption exceeding the capacity if users create services using up to L components. Hence we can determine

the maximum number of components that the users should be allowed to combine to create services such that the requirement of P is met. Section 4.1 presents simulations that demonstrate this type of usage.

1.2.2 Ad-hoc Services Environments

In ad-hoc environments we can skip Step 1 since we can directly determine the cost of services. In these environments we use Step 2 as above, plug in the cost of services and determine P. Section 4.2 presents experimental results that demonstrate this type of usage.

CHAPTER 2

ESTIMATING RESOURCE CONSUMPTION

This chapter consists of three parts. In Sections 2.1-2.3 we formalize components, sequences of components and present some results on the cost of such sequences. Sections 2.3-2.6 present the set of counting algorithms that can be used to determine the probability of resource consumption exceeding available capacity. Finally in Section 2.7 we compare the complexity of our approach with using generating functions and brute force.

2.1 Preliminaries

In this report we consider an algebra of free words on n components $\{a_1, a_2, \ldots, a_n\}$. Let $X \neq \phi$ be the set of data values, then $a_i : X_1 \rightarrow X_2$, $X_i \subset X$, is called a compo-

nent on X. A *sequences of components*[1] $a_1a_2\ldots a_n$ is a free word $[a_1a_2\ldots a_n]$. An empty sequence is $[\,]$. We define the multiplication operator $*$ as

$$[a_1a_2\ldots a_n]*[b_1b_2\ldots b_m] = [a_1a_2\ldots a_nb_1b_2\ldots b_m]$$

Let S be a set of sequences in $\{a_1a_2\ldots a_n\}$. Then,

1. $\forall s_1,s_2,s_3 \in S, (s_1*s_2)*s_3 = s_1*(s_2*s_3) = s_1s_2s_3$, hence $*$ is associative.

2. $[a_1a_2\ldots a_n][\,] = [\,][a_1a_2\ldots a_n] = [a_1a_2\ldots a_n]$, hence $[\,]$ is an identity.

Hence, $(S,*)$ is a semigroup with identity $[\,]$.

The *height* of a sequence s, denoted by $ht(s)$, is the number of components in s minus 1. Since every sequence is a free product of components, the height is a well defined constant.

The *prefix* of a sequence $s = a_1\ldots a_n$ is the sequence $a_1\ldots a_m$ where $m \leq n$.

The *suffix* of a sequence $s = a_1\ldots a_n$ is the sequence $a_l\ldots a_n$ where $l \leq n$.

A *factor* of a sequence s is the sequence formed by deleting a prefix and suffix from s. A factor is a part of a sequence. For example a_1a_2 is a factor of $a_1a_2a_3$, but not of $a_1a_3a_2$.

A *bifactor* of a sequence is a factor of height 1.

[1] In the rest of this report, where the context is unambiguous, we refer to a *sequence of components* as simply a *sequence*

2.2 Adjacency of Components

Using several simple components in sequences, such that the output of a preceding component can be acted upon by the succeeding component, can provide complex functionality. However, it might not always be possible to arbitrarily combine components. For example, a component that outputs a character string cannot be followed by a component that expects numeric input. Hence, for any two components a_1 and a_2 to appear adjacent to each other in a sequence as a_1a_2, at least the following must be true,

$$range(a_1) \cap domain(a_2) \neq \phi$$

Such adjacency constraints can be specified using an *adjacency matrix*. We can now formally define the adjacency matrix.

Suppose $M = a_1,..a_n$ is a set of components. Let $(a_{ij}) = A$ be a $n \times n$ matrix such that all the entries are 0 or 1. We say that a sequence $a_{i_1}a_{i_2}$ is *A-admissible* (or simply *admissible*) iff $a_{i_1 i_2} = 1$. Sequences with only 1 component are always admissible.

A sequence s is called A-admissible if every bifactor of s is admissible. $S(A)$ denotes the set of admissible sequences. The matrix A is then called an adjacency matrix of M.

Lemma 2.2.1. *If s' is a factor of s where $s \in S(A)$, then $s' \in S(A)$.*

Lemma 2.2.2. *If $a_{i_1} \ldots a_{i_n}$ and $a_{i_n}a_{i_{n+1}} \ldots a_{i_m}$ are two sequences in $S(A)$, then $a_{i_1} \ldots a_{i_n}\, a_{i_{n+1}} \ldots a_{i_m} \in S(A)$.*

Proof. Suppose s is a bifactor of $a_{i_1} \ldots a_{i_m}$, then s is a factor of $a_{i_1} \ldots a_{i_n}$ or $a_{i_n}a_{i_{n+1}} \ldots a_{i_m}$

(i.e s cannot be "split across" the two sequences since there is only one a_{i_n}). Hence, by Lemma 2.2.1 $s \in S(A)$. Hence since any such factor s of $a_{i_1} \dots a_{i_m}$ is admissible, by definition, $a_{i_1} \dots a_{i_m}$ is admissible. Hence, $a_{i_1} \dots a_{i_m} \in S(A)$. □

Theorem 2.2.3. *Let $A \in M_n(\{0,1\})$ and let $S(A)$ be the set of admissible sequences of A. Suppose $A^k = a^k_{(ij)}$, then a^k_{ij} is the number of sequences in $S(A)$ of height k with prefix a_i and suffix a_j.*

Proof. From Lemma 5.1, Chapter 31 in [11]. □

2.3 Cost Analysis

The *cost* of a module is a positive number assigned to it. We denote the cost of $a_i = c_i$. Set,

$$C^{(0)} = (c^{(0)}_{ij})$$

where,

$$c^{(0)}_{ij} = \delta_{ij} c_i \tag{2.1}$$

In Equation (2.1), δ_{ij} is the Kronecker delta defined as,

$$\delta_{ij} = \begin{cases} 0 & \text{for } i \neq j \\ 1 & \text{for } i = j \end{cases}$$

Hence, Equation (2.1) defines a diagonal matrix with the cost of module a_i at c_{ii}. We define the *cost of a sequence* as a sum of the cost of all modules in the sequence. Thus,

For $s = a_{i_1} \dots a_{i_m}$,

$$c(s) = \sum_{l=1}^{m} c_{i_l}$$

We assume that cost of $[\,]$, $c([\,]) = 0$.

Theorem 2.3.1. *Suppose $A \in M_m(\{0,1\})$ and $S(A)$ is the set of admissible sequences. Set*

$$C^{(n)} = A^n C^{(0)} + C^{(n-1)} A$$

Then

$$C_{ij}^{(n)} \quad = \quad \textit{cost of all sequences in } S(A) \textit{ of height n with prefix } a_i \textit{ and suffix } a_j.$$

Proof. We prove by mathematical induction. Put $C^{(0)} = (c_{ij}^{(0)})$ as defined above. Then the $(i,j)^{th}$ entry

$$\begin{aligned} (AC^{(0)} + C^{(0)}A)_{ij} &= \sum_{l=1}^{m} a_{il} c_{lj}^{(0)} + \sum_{l=1}^{m} c_{il}^{(0)} a_{lj} \\ &= a_{ij} c_j + c_i a_{ij} \\ &= a_{ij}(c_i + c_j) \end{aligned}$$

If $a_i a_j$ is admissible then $a_{ij} = 1$ and $c_{ij} = c_i + c_j$. This proves the base case for $n = 1$.

Now assume,

$$C^{(n)} = (c_{ij}^{(n)}) = A^n C^{(0)} + C^{(n-1)} A$$

then,

$$c_{ij}^{(n)} = \text{cost of all sequences with height } n \text{ with prefix } a_i \text{ and suffix } a_j$$

Now,

$$C^{(n+1)} = (c_{ij}^{(n+1)}) = A^{n+1}C^{(0)} + C^{(n)}A$$

then the $(i,j)^{th}$ entry of $C^{(n+1)}$ is,

$$\begin{aligned} c_{ij}^{(n+1)} &= \sum_{l=1}^{m} a_{il}^{n+1} c_{lj}^{(0)} + \sum_{l=1}^{m} c_{il}^{(n)} a_{lj} \\ &= a_{ij}^{n+1} c_j + \sum_{l=1}^{m} c_{il}^{(n)} a_{lj} \end{aligned} \tag{2.2}$$

Now using Theorem 2.2.3, the first term of Equation (2.2) is

$a_{ij}^{n+1} c_j$ = (number of sequences of height n+1)(c_j)

Also, by induction assumption, $c_{il}^{(n)}$ is the total cost of sequences of height n with prefix a_i and suffix a_l. Now, any sequence, with prefix a_i and suffix a_l can be appended with a_j iff $a_{lj} \neq 0$. Thus,

$$\begin{aligned} a_{ij}^{n+1} c_j + \sum_{l=1}^{m} c_{il}^{(n)} a_{lj} &= \text{(number of sequences of height n + 1)}(c_j)+ \\ &\quad \text{(cost of sequences of height } n \text{ with prefix } a_i \text{ and} \\ &\quad \text{which can be appended by } a_j) \\ &= \text{(cost of all admissible sequences of height } n+1 \text{ with} \\ &\quad \text{prefix } a_i \text{ and suffix } a_j) \end{aligned}$$

Hence, by induction,

$$C_{ij}^{(n)} = \text{cost of all sequences of height } n \text{ with prefix } a_i \text{ and suffix } a_j. \qquad \square$$

2.4 Anchor Points

Let,

$$V = \{(x_1, x_2 \ldots, x_u) \mid 1 \leq x_i \leq n, i = 1 \ldots u\}$$

Define cost of a point,

$$C : V \to \mathbb{R}_+$$

$$C((x_1, x_2 \ldots, x_u)) = \sum_{i=1}^{u} C_{x_i}$$

where,

1. $C_i \leq C_j$ if $i < j$
2. $C_i \in \mathbb{R}_+$

Using the symbols described above, we first introduce some definitions. Let $X \in V$, where V is a virtual space in $\mathbb{R}^u$ on n components.

Definition 2.4.1. A point $J = (j_1, j_2, \ldots, j_u)$ is a *neighbor* of point $I = (i_1, i_2, \ldots, i_u)$, if $I \neq J$ and $|i_k - j_k| \leq 1$, for all k, $1 \leq k \leq u$.

Definition 2.4.2. A point $J = (j_1, j_2, \ldots, j_u)$ is a *smaller neighbor* of point $I = (i_1, i_2, \ldots, i_u)$ if $j_k = i_k - 1$ for exactly one value of k. Note that $(1, 1, \ldots)$ does not have a smaller neighbor.

Definition 2.4.3. A point is an *anchor point* (a.p.) if the following are true for some $T \geq 0$,

- cost of the point is $\geq T$
- If the point has 1 or more smaller neighbors, the cost of at least one of those smaller neighbors is $< T$.

Definition 2.4.4. X is *connected* if for any $I, I' \in X$, $\exists I_1, I_2, \ldots, I_r$ such that $\{I_1, I\}$ are neighbors, $\{I_2, I_1\}$ are neighbors and so on up to $\{I_r, I'\}$ are neighbors and $I_i \in X$, $1 \leq i \leq r$. $\{I, I_1, \ldots, I_r, I'\}$ is called the *connected path.*

Theorem 2.4.1. *If $A = (a_1, a_2, \ldots, a_u)$ is an anchor point such that $A \notin \{(1,1\ldots),(n,n,\ldots)\}$. Then if there are other anchor points, at least one neighbor of A must also be an anchor point.*

Proof. We try to assign non-anchor points to every neighbor of A and then use proof by contradiction. W.l.o.g assume $S = (a_1 - 1, a_2, \ldots, a_u)$ is that smaller neighbor of A such that $C(S) < T$. By the definition of a.p and constraints in the theorem statement we know that such a point exists.

Let $B = (a_1, a_2, a_3 + 1, \ldots, a_u)$. By definition we know that $C(A) \geq T$. Hence, $C(B) \geq T$ since costs are ascending.

Consider $N = (a_1 - 1, a_2, a_3 + 1, \ldots, a_u)$. N is a neighbor of A and a small neighbor of S. We now try to assign some cost to N.

CASE I: $C(N) < T$

Then B is an anchor point since,

1. $C(B) \geq T$

2. C is a smaller neighbor of B and $C(N) < T$

So B is a neighbor of A and B is an anchor point.

CASE II: $C(N) \geq T$

Then N is an anchor point since,

1. $C(N) \geq T$.

2. S is a smaller neighbor of N and $C(S) < T$.

So N is a neighbor of A and N is an anchor point.

Hence, it is impossible to assign any cost to N so that no neighbor of A is an anchor point. Since N must have some cost, we can say that there is at least one neighbor of A that is an anchor point. □

Let σ be a permutation on u objects. We define action of σ on a point $(x_1, \ldots, x_u)$ in V as follows:

$$\sigma((x_1, \ldots, x_u)) = (x_{\sigma(1)}, \ldots, x_{\sigma(u)})$$

Lemma 2.4.2. *If A is a set of anchor points, then A is permutation stable.*

Proof. If I is an anchor point and J is a small neighbor of I with cost less than T, then $\sigma(J)$ is a small neighbor of $\sigma(I)$. Since cost is invariant under this action, $\sigma(I)$ is indeed an anchor point. □

Lemma 2.4.3. *Suppose $I = (i_1, i_2, \ldots, i_u)$ and $J = (j_1, j_2, \ldots, j_u)$ are two points in V, $I \neq J$ and A is the set of anchor points. If $i_k > j_k \forall 1 \leq k \leq u$, then either $I \notin A$ or $J \notin A$ (both I and J cannot be anchor points).*

Proof. Suppose I is an anchor point. By definition there is k such that the cost of $(i_1,\ldots,i_k-1,\ldots,i_u)$ is less than T. Since $i_k > j_k \forall j$, cost of J is also less than T. □

Lemma 2.4.4. *Suppose* $(i_1,i_2,\ldots,i_u)$ *and* $(i_1,\ldots,\ i_l+\ r,\ldots,i_u)$ *are anchor points, then* $(i_1,\ldots,\ i_l+s,\ldots,i_u)$ *is an anchor point* $\forall s$, $1 \leq s < r$.

Proof. From definition of anchor points and construction of virtual space described above, it is clear that $\forall s$, $1 \leq s < r$, $C(i_1,\ldots,i_l+s,\ldots,i_u) \geq T$. Now, $(i_1,\ldots,\ i_l+r,\ldots,i_u)$ has a smaller neighbor say $N = (i_1,\ldots,i_j-1,\ldots,i_l+r,\ldots,i_u)$ ($j > l$, is also possible, and does not affect this proof), where $C(N) < T$ (required by definition of an anchor point). Now, $N_1 = (i_1,\ldots,i_j-1,\ldots,i_l+r-1,\ldots,i_u)$ is a smaller neighbor of $(i_1,\ldots,i_l+r-1,\ldots,i_u)$. Also since $C(N) < T, C(N_1) < T$. Hence $(i_1,\ldots,i_l+r-1,\ldots,i_u)$ is an anchor point. Inductively we can now prove that all points $(i_1,\ldots,i_l+s,\ldots,i_u)$ $1 \leq s < r$ are anchor points. □

Lemma 2.4.5. *Suppose X is any connected set that contains points I, J such that* $C(I) < T$ *and* $C(J) \geq T$*, then X contains an anchor point or X contains a neighbor of an anchor point.*

Proof. Suppose X does not contain an anchor point. Let $I_1,I_2,\ldots,I_r$ be a connected path from I to J. Since $C(I) < T$ and $C(J) \geq T$, $\exists s$, $1 \leq s \leq r$ ($I_{r+1} \equiv I'$) such that $C(I_s) < T$ and $C(I_{s+1}) \geq T$. There can be more than one such s. We can also have $C(I_{s+1}) < T$ and $C(I_s) \geq T$ but for simplicity we assume that $C(I_s) < T$ and $C(I_{s+1}) \geq T$. The method of this proof holds in both case.

Since X does not contain an anchor point, I_s is not a smaller neighbor of I_{s+1}, since if that was the case then $I_{s+1} \in X$ would be an anchor point. Now choose $P_1,P_2,\ldots$ as common neighbors of I_{s+1} and I_s using the following construction.

Choose $P_1,\ldots,P_r$ so that P_1 differs from I_s in exactly one coordinate, P_2 in two coordinates etc.. Also P_2 and P_1 differ in one coordinate etc..

If $C(P_1) < T$ then look for P_2. If $C(P_2) \geq T$ then P_2 is an anchor point. By induction, one of these P_i is an anchor point. □

Theorem 2.4.6. *The set A of anchor points is connected.*

Proof. For $I = (i_1,\ldots,i_u)$, $J = (j_1,\ldots,j_u)$, define the distance,

$$d(I,J) = \sum_{l=1}^{u} |i_l - j_l|$$

Let $I,J \in A$, $I \neq J$, $d(I,J) = d$. We prove that $\exists I_1 \in A$ such that $d(I_1,J) < d$. Since $I \in A$, we assume w.l.o.g that there exists a small neighbor $I_0 = (i_1 - 1,\ldots,i_u)$ with $C(I_0) < T$.

CASE I: $i_1 < j_1$

By Lemma 2.4.3 $\exists k$ such that $i_k > j_k$. Set $I' = (i_1,\ldots,i_k - 1,\ldots,i_u)$. Clearly $d(I',J) < d$.

CASE Ia: $C(I') \geq T$

Take $I_1 = I'$. Note that $C(i_1 - 1,\ldots,i_k - 1,\ldots i_u) \leq C(I_0) = C((i_1 - 1,\ldots,i_u)) < T$. Thus I_1 is an anchor point with smaller neighbor $(i_1 - 1,\ldots,i_k - 1,\ldots i_u)$ and we are done.

CASE Ib : $C(I') < T$

Consider $I'' = (i_1 + 1,\ldots,i_k - 1,\ldots,i_u)$, then I' is a smaller neighbor of I''. If $C(I'') \geq T$, then $I_1 = I''$ is an anchor point with smaller neighbor I'. If $C(I'') < T$, then $I_1 = (i_1 + 1,\ldots,i_k,\ldots,i_u) \in A$ and $d(I,J) < d$.

CASE II : $i_1 > j_1$

By Lemma 2.4.3 $\exists k$ such that $i_k < j_k$. Set $I' = (i_1, \ldots, i_k + 1, \ldots, i_u)$. Clearly $C(I') \geq T$ and $d(I', J) < d$. Let $I'' = (i_1 - 1, \ldots, i_k + 1, \ldots, i_u)$, then $d(I'', J) < d$.

If $C(I'') < T$, then $I_1 = I'$ and $I_1 \in A$. If $C(I'') \geq T$, then $I_1 = I''$ and $I_1 \in A$.

CASE III: $i_1 = j_1$

Find smallest k s.t. $i_k \neq j_k$. We can now use Case I or Case II.

By induction, we can find a connected path from I to J.

□

2.5 Probabilistic Estimates on Consumption

This section tries to answer the following question: Given a set of components each with its associated costs C, adjacency constraints on these components in the form of an adjacency matrix A and total available resource is exceeded at T,then, within an environment with U simultaneous users, what should be the maximum allowed sequence length for each user so that the probability of exceeding the total resource consumption is P ?

Hence, given some sequence length constraint l, each of the U users can pick any string that is $\leq l$. We want to ensure that the total cost of all U such sequences equal or exceeds T with probability P. A conservative estimate of $P = 0$ will never let the users overload the system but will at most times lead to underutilization. On the other hand $P \approx 1$ will almost always overload the system. Hence, it is the system designer's choice to pick an appropriate P that best reflects his/her context. Given this P we try to find an l that we can give the end users as a *maximum* length of sequences that they can use.

Our approach to the problem is that we iterate over some range for l and for each value find the corresponding P. Once we find the closest P we select the corresponding l as our answer.

There can be several sequences of each length $\leq l$. Hence we need an efficient way to find the probability that the sum of the sequences of length $\leq l$ is not $\geq T$. We use a counting approach wherein we work within a *virtual*[2] space, $V = \{(C_{i_1}, \ldots C_{i_u}) \mid 1 \leq C_{i_1}, \ldots, C_{i_u} \leq n\}$[3], $V \subset \mathbb{R}^u$ and the number of elements in V, $|V| = n^u$. Each C_i, $1 \leq i \leq n$, is the *average* cost of sequences of height $i-1$. We use identities developed in Theorem 2.2.3 and Theorem 2.3.1 to find the average cost C_i. We define the cost of any point $I \equiv (c_{i_1}, \ldots, c_{i_u})$ as $cost(I) = \sum_{j=1}^{u} C_{c_{i_j}}$. I is a favorable event if $cost(I) \geq T$. Hence, if G is the number of favorable points then $P(sum \geq T) = \frac{G}{n^U}$.

In order to find G we use a 2-stage approach. The first stage consists of finding the *anchor points* (described below). The second stage counts favorable events. The overall problem solving strategy is as follows:

1. Sort average costs.

2. Determine anchor. (Algorithm 1)

3. Using the set of anchor points to determine total number of points in virtual space $\geq T$. (Algorithm 2)

If we can find one anchor point, then using Theorem 2.4.1 we can find all the anchor points by searching neighbors . This is the approach we take in Algorithm 1 to find

[2]We call this space *virtual* since we never enumerate all the points in this space, so this space is only conceptual.

[3]Instead, one can use two indices such as $i_1, i_2, \ldots$, however we found this method of indexing more suitable.

Algorithm 1 *AllAnchors*(C, U, T)

1: {C is a $1 \times n$ matrix with n costs}
2: {U is the number of users}
3: {T is the max total cost}
4: {Find first *diagonal* point $\geq T$}
5: **for** $i = 1$ to n **do**
6: $A \Leftarrow$ Point with all elements i
7: **if** $cost(A) \geq T$ **then**
8: break
9: **end if**
10: **end for**
11: **if** no diagonal point $\geq T$ **then**
12: return empty set {done}
13: **end if**
14: **if** A is anchor point **then**
15: Add A to result set
16: **else**
17: **for** each axis x **do**
18: **for** $j = 1$ to i **do**
19: set xth element of A to j
20: **if** $cost(A) \geq T$ **then**
21: break
22: **end if**
23: **end for**
24: **if** A is anchor point **then**
25: Add A to result set
26: **end if**
27: **end for**
28: **end if**
29: {at this stage the result set contain the first *seed* anchor point}
30: **for all** B such that B is an unprocessed point in result set **do**
31: mark B as processed
32: $K \Leftarrow$ all neighbors of B
33: **for all** k in K **do**
34: **if** k is anchor point and k is not in result set **then**
35: Add k to result set
36: **end if**
37: **end for**
38: **end for**
39: return result set

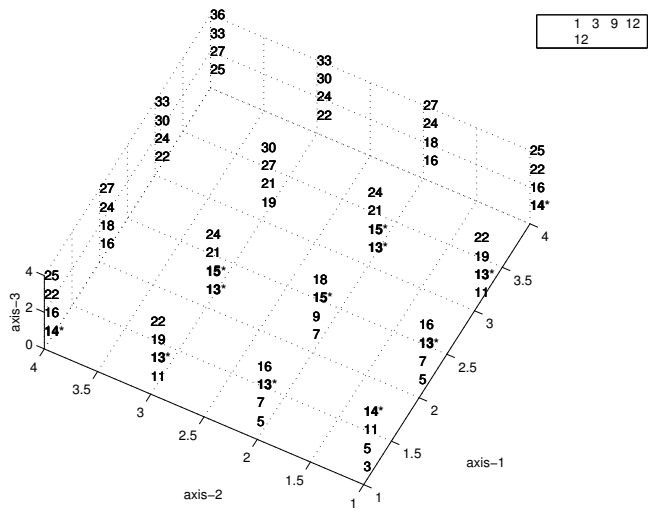


Figure 2.1: Anchor points (shown with "*") for cost matrix $[1,3,9,12]$ and $T = 12$.

the set of anchor points.

Lines 6-29 determine the first anchor point. We begin by rapidly traversing (lines 6-11) along the diagonal ($(1,1,\ldots,1) \rightarrow (n,n,\ldots,n)$) of the virtual space looking for a point $\geq T$. If no such point is found we know that there is no anchor point and we return and empty result set. If such a point is found, it might not be an anchor point so we check along each axis for a anchor point (lines 18-28). In lines 31-39 we use the result from Theorem 2.4.1 to find the rest of the anchor points.

Figure 2.1 shows the anchors points for a 3 dimensional virtual space with average costs matrix $[1,3,9,12]$ and $T = 12$.

2.6 Anchors to Points

Lemma 2.6.1. *(a) Suppose* $(a_1, a_2, \ldots, a_u) \in A$ *is such that*

$$a_k = max\{b_k \mid (a_1, \ldots, a_{k-1}, b_k, b_{k+1}, \ldots, b_u) \in A\}$$

If $(a_1, \ldots, a_{k-1}, a'_k, \ldots, a'_u) \in V$ *and if* $a'_k > a_k$ *then* $C((a_1, \ldots, a_{k-1}, a'_k, \ldots, a'_u)) \geq T$.

(b) Suppose $(b_1, \ldots, b_u) \in V$, $C((b_1, \ldots, b_u)) \geq T$. *Then* $\exists (a_1, \ldots, a_u) \in A$ *such that,*

1. $a_i = b_i$ *for* $i = 1, \ldots, k-1$, $a_k < b_k$ *and,*

2. $a_k = max\{c_k \mid (a_1, \ldots, a_{k-1}, c_k, \ldots, c_u) \in A\}$

Proof. (a) Suppose $C(a_1 \ldots, a_{k-1}, a'_k, \ldots, a'_u) < T$. Since $a'_k > a_k$, not all $a'_i \geq a_i$, $i = k+1, \ldots, u$. Assume that $a'_{l_i} < a_{l_i}$ for $i = 1, \ldots, r$, $k+1 \leq l_i \leq u$. Now by adding s to a'_{l_i} and checking the cost of a point $(a_1, \ldots, a_{k-1}, a'_k, \ldots, a'_{l_1} + s, a'_{l_2}, \ldots, a'_u)$ we can eventually find a point $v = (a_1, \ldots, a_{k-1}, a'_k, \ldots, a'_{l_j}, \ldots, a'_u)$, such that $C(v) < T$ but $C((a_1, \ldots, a_{k-1}, a'_k, \ldots, a'_{l_j} + 1, \ldots, a'_u) \geq T$. But then $(a_1, \ldots, a_{k-1}, a'_k, \ldots, a'_{l_j} + 1, \ldots, a'_u) \in A$ with $a'_k > a_k$, which contradicts the maximality of a_k.

(b) We choose $(a_1, \ldots, a_u) \in A$, with the properties,

- $a_i = b_i$ for $i = 1, \ldots, k-1$,
- k is the largest value such that any anchor point has first $k-1$ coordinates equal to $b_1, \ldots, b_{k-1}$,
- $a_k = max\{c_k \mid (a_1, \ldots, a_{k-1}, c_k, \ldots, c_u) \in A$ and $a_k < b_k\}$.

We will show that $a_k = max\{b'_k \mid (a_1,\ldots,a_{k-1},b'_k,\ldots,b'_u) \in A\}$

Suppose this is not true. Then, there exists $(a_1,\ldots,a_{k-1},b'_k,\ldots,b'_u) \in A$, such that $b'_k > a_k$. Since k is largest so that $a_1,\ldots,a_{k-1}$ are the coordinates of the given point $b'_k \geq b_k$.

Consider $v_1 = (a_1,\ldots,a_{k-1},b_k,1,\ldots,1)$. If $v_1 \in A$, then we have a contradiction since longest prefix is $a_1,\ldots,a_{k-1},b_k$ and not $a_1,\ldots,a_{k-1}$.

CASE I: $C((a_1,\ldots,a_{k-1},b_k,1,\ldots,1)) < T$

Then we have, $C((a_1,\ldots,a_{k-1},b_k,1,\ldots,1)) < T$ and $C((a_1,\ldots,a_{k-1},b_k,\ldots,b_u)) \geq T$. Then there exists a stage such that $C((a_1,\ldots,a_{k-1},b_k,b_{k+1},\ldots,b_l-1,1,\ldots,1)) < T$ and $C((a_1,\ldots,a_{k-1},b_k,b_{k+1},\ldots,b_l,1,\ldots,1)) \geq T$. But then $(a_1,\ldots,a_{k-1},b_k,\ldots,b_l,1,\ldots,1) \in A$, a contradiction.

CASE II: $C((a_1,\ldots,a_{k-1},b_k,1,\ldots,1)) \geq T$

Now $(a_1,\ldots,a_{k-1},b'_k,\ldots,b'_u) \in A$, and $b'_k \geq b_k$. This implies that

$$T \leq C((a_1,\ldots,a_{k-1},b_k,1,\ldots,1)) \leq C((a_1,\ldots,a_{k-1},b'_k,\ldots,b'_u))$$

Suppose $(a_1,\ldots,a_l-1,\ldots,a_{k-1},b'_k,\ldots,b'_u)$ is a small neighbor of the anchor point $(a_1,\ldots,a_{k-1},b'_k,\ldots,b'_u)$, then since

$$C((a_1,\ldots,a_l-1,\ldots,a_{k-1},b_k,1,\ldots,1))) \leq C((a_1,\ldots,a_l-1,\ldots,a_{k-1},b'_k,\ldots,b'_u)) < T,$$

$(a_1,\ldots,a_{k-1},b_k,1,\ldots,1) \in A$ a contradiction.

If $(a_1,a_{k-1},b'_k-1,b'_{k+1},\ldots,b'_u)$ is a small neighbor of $(a_1,\ldots,a_{k-1},b'_k,\ldots,b'_u)$, then $(a_1,a_{k-1},b_k-1,1,\ldots,1)$ is a small neighbor of $(a_1,a_{k-1},b_k,1,\ldots,1)$ again showing

that $(a_1, a_{k-1}, b_k, 1, \ldots, 1) \in A$ since,

$$C((a_1, a_{k-1}, b_k, 1, \ldots, 1)) \leq C((a_1, a_{k-1}, b'_k, \ldots, b'_l - 1, \ldots, b'_u))$$

for any $b'_l > 1$. $(a_1, \ldots, a_{k-1}, b'_k, \ldots, b'_u)$ cannot be a smaller neighbor of the following type: $(a_1, \ldots, a_{k-1}, b'_k, \ldots, b'_l - 1, b'_u)$.

□

2.6.1 Points Counting

In V we are working with the surface defined by the set of anchor points. This surface can be thought of as a boundary between points $< T$ and those $\geq T$. Algorithm 2 presents a technique that, given a *sorted*[4] list of anchor points, calculates the number of points $\geq T$.

Algorithm 2 is a recursive algorithm and is initially called as *anchors2Points* with the following parameters: a sorted list of anchor points A, the max value of any element in an anchor point n, and a 2×2 matrix representing the dimensions of the anchor list. This algorithm relies on the *makegroups* algorithm (Algorithm 3).

The *makegroups* algorithm generates recursive groups within the list of anchor points such that the first column in every generated group is identical. The effect of such grouping is that with each level of recursive grouping we reduce a dimension of our virtual space. This process is illustrated by continuing the example from Section 2.5. Anchor points generated for costs $[1, 3, 9, 12]$ and $T = 12$, its grouping obtained by *makegroups* and

[4] Lexicographically sorted

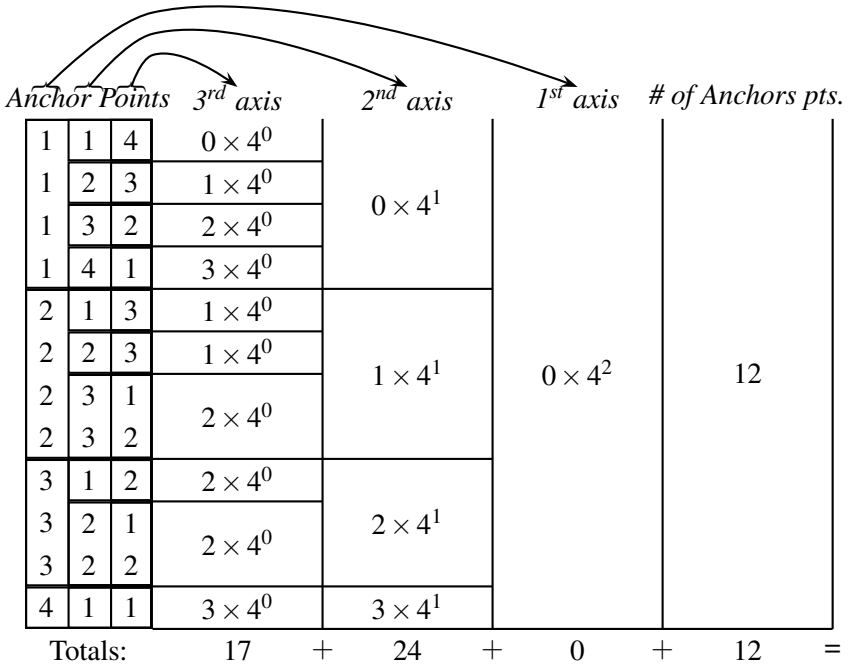


Figure 2.2: Illustrating Algorithm 2, counting points from anchor points

counting done by *anchors2Points*, is illustrated in the Figure 2.2.

2.6.2 Generalized Points Counting

In Section 2.4 we assumed that the costs were unique and Algorithm 2 depends on this assumption. In this section we present Algorithm 4, a generalized version of Algorithm 2 that relaxes this uniqueness constraint and allows counting points without generating additional anchor points. The main difference between Algorithm 4 and Algorithm 2 is that in Algorithm 4 during the counting steps the known number of duplicates of each costs are accounted for.

Figure 2.3 illustrates the working of Algorithm 4 for costs $[2,4,4,10,10,10]$. The anchors points were obtained by calling *AllAnchor* with costs $[2,4,10,]$. *GeneralizedAnchors2Points* was called with the anchors points and $[1,2,3]$. With $T = 14$ and $U = 3$, there

Algorithm 2 *Anchors2Points*(A, n, r)

1: $\{A$ is the $k \times U$ matrix of k of anchor points$\}$
2: $\{n$ is the max value of any element in an anchor point$\}$
3: $\{r$ is a 2×2 matrix where $[rs\ cs; re\ ce]$ define the upper left hand corner (rs, cs) and (re, ce) define the lower right hand corner of the space in A that is currently being processed$\}$
4: **if** A is empty **then**
5: return 0
6: **end if**
7: $[rs\ cs; re\ ce] \Leftarrow r$
8: **if** $r(2,2) - r(1,2) = 0$ **then**
9: $result = result + r(2,1) - r(1,1) + 1$ {count number of anchor pts}
10: $m = n - max(cs^{th}$column from rows rs to re in $A)$
11: $result = result + m$
12: return $result$
13: **end if**
14: $r1 = makegroups(A, r)$
15: **for all** ri such that ri is a 2×2 group specification in $r1$ **do**
16: $result = result + anchors2Points(A, n, ri)$
17: **end for**
18: $femax = max(cs^{th}$column from rows rs to re in $A)$
19: $rp = (n - femax) \times n^{U-cs}$
20: return $result + rp$

Algorithm 3 *MakeGroups*(A, r)

1: $\{A$ is the $k \times U$ matrix of k of anchor points$\}$
2: $\{r$ is a 2×2 matrix where $[rs\ cs; re\ ce]$ define the upper left hand corner (rs, cs) and (re, ce) define the lower right hand corner of the space in A within which groups are to be made$\}$
3: $[rs\ cs; re\ ce] \Leftarrow r$
4: $ri \Leftarrow rs$
5: **while** $ri \leq re$ **do**
6: Record $[ri\ cs + 1]$ as beginning of group
7: $fe \Leftarrow A(ri, cs + 1)$
8: **while** $ri \leq re$ **do**
9: **if** $fe = A(ri, cs + 1)$ **then**
10: $rf \Leftarrow ri$
11: $ri \Leftarrow r1 + 1$
12: **end if**
13: **end while**
14: Record $[rf\ ce]$ as end of group
15: **end while**
16: return all groups

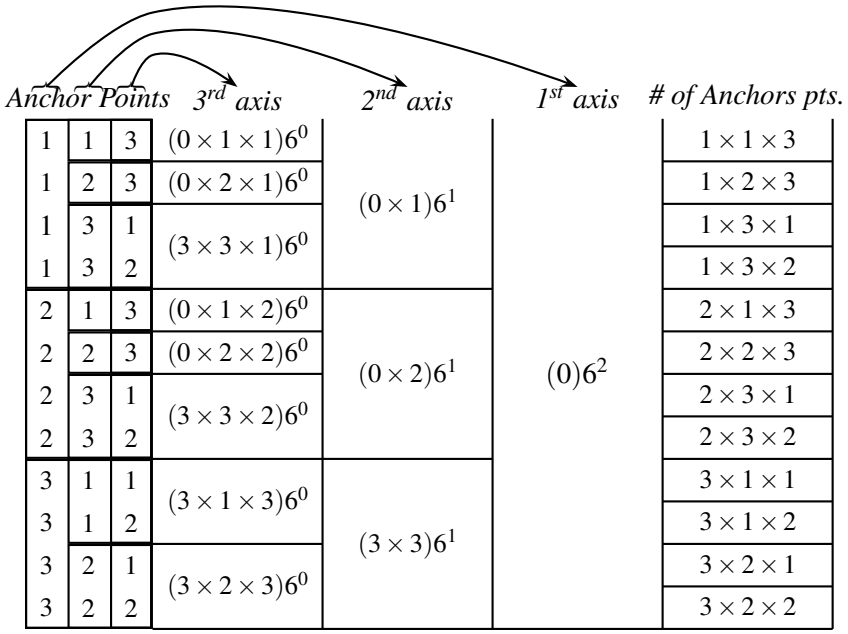


Figure 2.3: Illustrating Algorithm 4, counting points from anchor points

were 189 points with cost $\geq T$.

2.7 Complexity Analysis

We now perform the asymptotic running time analysis for Algorithms 1 and 2 and then compare it with the brute force method (Section. 2.7.1) and generating functions (Section 2.7.2). We use the notations of Section 2.5. The complexity of finding the cost of any point in V is $O(u)$ and to decide whether a point is an anchor point is $O(u^2)$.

For Algorithm 1

- Lines 6-11: $O(nu)$ since max i is n.
- Lines 18-28: $O(u)$ since x is number of axes u.
  - Lines 19-24: $O(nu)$ since j till max $i = n$.

Algorithm 4 *GenralizedAnchors2Points*(*A*,*N*,*r*)

1: $\{A$ is the $k \times U$ matrix of k of anchor points$\}$
2: $\{N$ is $1 \times n$ matrix where n is the number of unique cost and N(1,i) is the number of times the ith cost is repeated.$\}$
3: $\{r$ is a 2×2 matrix where $[rs\ cs; re\ ce]$ define the upper left hand corner (rs, cs) and (re, ce) define the lower right hand corner of the space in A within that is currently being processed$\}$
4: $[rs\ cs; re\ ce] \Leftarrow r$
5: **if** $r(2,2) - r(1,2) = 0$ **then**
6: {count number of anchor pts}
7: **for all** p such that p is an a.p in rows rs to re **do**
8: $pc \Leftarrow 1$
9: **for all** q such that q is an element of p **do**
10: $pc \Leftarrow pc \times$ (# of times q is repeated)
11: **end for**
12: $result \Leftarrow result + pc$
13: **end for**
14: {Count points along U^{th} axis}
15: $femax = max(cs^{th}$column from rows rs to re in $A)$
16: $m =$ # of costs (with duplicates) greater than $femax$
17: **for all** q value at indices in row rs less than cs **do**
18: $m = m \times$(# of times q is repeated)
19: **end for**
20: $result = result + m$
21: return $result$
22: **end if**
23: $r1 = makegroups(A, r)$
24: **for all** ri such that ri is a 2×2 group specification in $r1$ **do**
25: $result = result + GeneralizedAnchors2Points(A, N, ri)$
26: **end for**
27: $femax = max(cs^{th}$column from rows rs to re in $A)$;
28: $m =$ # of costs (with duplicates) greater than $femax$
29: **for all** q value at indices in row rs less than cs **do**
30: $m = m \times$(# of times q is repeated)
31: **end for**
32: $rp = m \times$(total# of costs (with duplicates))$^{U-cs}$
33: return $result + rp$

- Lines 25-27: $O(u^2)$ since testing for anchor point.

Hence total for lines 18-28 is $O(u)[O(nu)+O(u^2)] = O(nu^2)+O(u^3)$.

- lines 31-39: The cost of enumerating all neighbors of a points is $O(3^u)$. Hence checking all neighbours for anchor points is $O(3^u u^2)$. Assuming that a such anchor points are processed. The total cost is $O(a3^u u^2)$. Insertion into list without duplicates is $O(a^2)$.

Hence complexity of Algorithm 1 is

$$O(nu)+O(nu^2)+O(u^3)+O(a3^u u^2)+O(a^2)$$

Algorithm 1 requires sorted n. We assume this sorting complexity to be $O(n\log n)$.

Algorithm 3 requires a lexicographically sorted list of anchors points ($O(au\log(au))$ and since there can be at most au groups with exactly one anchor point in each group, the complexity is $O(au)$. Hence, $O(au\log(au))+O(au) = O(au\log(au))$.

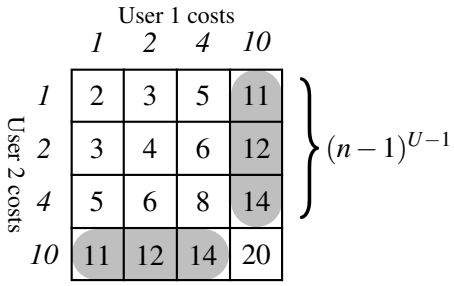
Algorithm 2 relies on Algorithm 3 and performs exactly 1 multiplication per group (line 19) and at most a additions per axis (line 16). Hence, the complexity is $O(au)$.

Total complexity is then,

$$\begin{aligned} & O(nu)+O(nu^2)+O(u^3)+O(a3^u u^2)+O(a^2)+O(au\log(au))+O(n\log n) \\ & +O(au) \\ = \;& O(nu)+O(nu^2)+O(u^3)+O(a3^u u^2)+O(a^2)+O(au\log(au))+O(n\log n) \\ = \;& O(nu+nu^2+u^3+a3^u u^2+a^2+au\log(au)+n\log n) \end{aligned} \tag{2.3}$$

The number of anchors points a is variable and depends on T, however we do know that the best case (minimum number of anchor points) is $a = 1$ with the point $(1,1,1,\ldots)$, in which case every point in V is $\geq T$. We hypothesize the worst case to be $a = u(n-1)^{u-1}$. Proof of this hypothesis will be the subject of future work. This worst case corresponds to the case when anchor points consist of all points on the the outer surfaces of V, such that the surface contains the max point in V, $(n,n,n,\ldots)$.

The figure below shows the case of max anchor points for costs $[1,2,4,10]$, $U = 2$. The max number of anchor points here are for $T = 11$, $2 \times (4-1)^{2-1} = 6$.



In order to understand the relevance of Equation (2.3), we now look at it's growth characteristics separately w.r.t. u and n and analyze each for best and worst case of a.

Using constant u, Equation (2.3) becomes $n\log n + a^2 + a\log a$. With worst case, $a = u(n-1)^{u-1}$,

$$\begin{aligned} & O(n\log n + u^2(n-1)^{2(u-1)}) \\ = \; & O((n-1)^{2(u-1)}) \\ = \; & O(n^{2u}) \end{aligned} \tag{2.4}$$

With best case $a = 1$, Equation (2.3) is $O(n \log n)$. This is simply the time to sort the n costs.

Using constant n Equation (2.3) becomes $O(u^3 + a3^u u^2 + a^2 + au\log(au))$. With worst case, $a = u(n-1)^{u-1}$,

$$\begin{aligned} & O(u^3 + u(n-1)^{u-1}3^u u^2 + u^2(n-1)^{2(u-1)} + u(n-1)^{u-1}u\log(u(n-1)^{u-1}u)) \\ = \; & O((n-1)^{2(u-1)}) \\ = \; & O(n^{2u}) \end{aligned} \tag{2.5}$$

With best case, $a = 1$, Equation (2.3) is $O(3^u)$.

2.7.1 Comparison with Brute Force Method

The brute force method is one in which we test each point in V for $\geq T$. The complexity of this approach is always $O(un^u)$. We can conclude that while in the worst case (Equations (2.4) and (2.5)) our algorithms exhibit worse scalability than brute force whereas in the best case we do significantly better. Our future work will include an average case complexity analysis in which we hope to show that our approach works much better that brute force in the average case. This is primarily because the worst case, as outlined above, occurs only for a very narrow range of T.

2.7.2 Comparison with Generating Functions

A classic method of counting points in the U-dimensional space is using generating functions. Though our problem requires counting points $\geq T$, when using generating functions it is easier to calculate points $\leq T$, so that is what we will do in this section. We

begin with explaining this method and then compare it with our algorithms.

Let the costs be $[1,2,5]$, $U = 2$ and $T = 3$. This first step is to create a generating function *for each axis* from the costs. Such a generating function is,

$$g(x) = x + x^2 + x^5 \tag{2.6}$$

Since there are two such axes we multiply the generating function twice and divide the result by $(1-x)$ [10] in order to accumulate coefficients. We then differentiate the result T times, evaluate at $x = 0$ and divide by $T!$ to get the number we want. Multiplying generating functions had the effect of creating a multinomial expansion of the generating function. Such an expansion has the property that for each term in the expansion, the coefficient of a given term in x happens to be the number of ways to get the exponent of that x. Multiplying the generating function by $\frac{1}{1-x}$ has the effect of accumulating coefficients [10]. Now, in order to get the coefficient of the x term with an exponent of T we differentiate T times. We then evaluate the result at $x = 0$ to cancel all terms with exponents greater than T. Finally to negate the effect of repeated derivative on the coefficient of x we divide the end result by $T!$. Hence we evaluate,

$$\begin{aligned} N &= \frac{d^T}{dx^T}\frac{(x+x^2+x^5)^2}{1-x}\Big]_{x=0}\frac{1}{T!} \\ &= \frac{d^3}{dx^3}\frac{(x+x^2+x^5)^2}{1-x}\Big]_{x=0}\frac{1}{3!} \\ &= 3 \end{aligned} \tag{2.7}$$

We can now say that there are 3 ways that $T \leq 3$, they are $(1,1),(1,2),(2,1)$.

Equivalently there are $6 = 3^2 - 3$ ways such that the sum is ≥ 4. So, the required probability of $T \geq 4$ is $\frac{6}{9} = 0.66$. There are two computationally expensive steps in the evaluation of Equation (2.7): Evaluating the T^{th} derivative and computing $T!$. We will now look at how we can eliminate or reduce one or both of these steps.

If we try to prevent explicit computation of $T!$ we must compute $\frac{d^T}{dx^T}$ in T steps, $\frac{d}{dx}, \frac{d^2}{dx^2}, \ldots, \frac{d^T}{dx^T}$ and at each step divide the result by $n, n-1, n-2, \ldots, 1$. Division of $\frac{d^T}{dx^T}$ by $T!$ or division of each successive derivative by $n, n-1, n-2, \ldots$ is required to cancel the effect of multiplication of the coefficient of x by it's exponent in each successive derivative.

If we try to prevent successive computation of derivatives then we must compute $T!$. While the gamma function $\Gamma(T) = \int_0^\infty x^T e^{-x} dx$ can be used to exactly calculate $T!$, $T! = \Gamma(T+1)$, it involves at least T integration steps. An approximation to $T!$ can be obtained using T or fewer steps by using Sterling's approximation as $T! \approx \sqrt{2\pi T}\left(\frac{T}{e}\right)^T$. In either case, the factorial value is large for even modest values of T, for example, 100! contains 158 digits. Given that T represents the sum of costs, finding the factorial is constrained to costs that can be handled by the precision of the system on which the computation is being carried out. It is unlikely that most commonly used general-purpose systems will be able to compute the factorial of large cost values (in the order to several hundreds or thousands) without overflow.

Hence we cannot eliminate both these computationally intensive steps simultaneously. So the choice is between finding the T^{th} derivative, one at a time or calculating $T!$. Both these are choices severely constrain using this generating functions technique for large values of T (several hundreds or thousands).

Though the computational requirement of this method is sensitive to T, it is fairly immune to U, the number of users. This is the case since in Equation (2.7), U is the exponent of the generating function and does not affect the computational effort to find a derivative.

CHAPTER 3

SLICE ALGORITHMS

The set of anchor points A determined using Algorithm 1 is permutation stable. We then used those anchor points to count the number of points with cost $\geq T$ (Algorithm 2). In this chapter we improve upon that approach by determining only permutation unique anchor points and then counting using this reduced set. The set of permutation unique anchor points, that we call *slice anchor points* (Definition 3.1.3) is smaller by a factor of $u!$ than the set of anchor points.

3.1 Preliminaries

Set,

$$V_+ = \{P \in V \mid x_i(P) \leq x_{i+1}(P), i = 1, \ldots, u-1\}$$

where,

$$x_i(P) = i^{th}\text{coordinate of } P.$$

Definition 3.1.1. Let $< X >$ be an operator that generates all permutations of X.

Example 3.1.1.

$$< (1,2,3) >= \{(1,2,3),(1,3,2),(2,1,3),(2,3,1),(3,1,2),(3,2,1)\}$$

$$< \{(1,2),(2,3)\} >= \{(1,2),(2,1),(2,3),(3,2)\}$$

Hence, $< V_+ >= V$.

Definition 3.1.2.

$$\begin{aligned} \overline{\ }: \ V &\to \ V_+ \\ P &\to \ \overline{P} \end{aligned}$$

with $\overline{P}$ obtained by rearranging coordinates of P suitably.

Lemma 3.1.1. *For any $P \in V$, the cardinality of the set $\{Q \in V \mid \overline{Q} = \overline{P}\}$ is* $\binom{u}{r_1,\ldots,r_s}$ *where,*

$$\begin{aligned} r_1 &= \textit{repetition of } x_1(\overline{P}) \\ r_2 &= \textit{repetition of } x_{1+r_1}(\overline{P}) \\ &\ldots \end{aligned}$$

Proof. See second paragraph on page 16 in [29]. □

Example 3.1.2. Let $V = \{(x_1,x_2,x_3,x_4) \mid x_i \in \mathbb{Z}^+, i = 1,\ldots,4\}$, and $P = (1,2,1,3)$. Then

$\overline{P} = (1,1,2,3)$ and $r_1 = 2, r_2 = 1, r_3 = 1$, and

$$\left|\{Q \in V \mid \overline{Q} = \overline{P}\}\right| = \binom{4}{2,1,1} = \frac{4!}{2!1!1!} = 12.$$

Definition 3.1.3. *Slice anchor points* are $A_+ = A \bigcap V_+$, where A is the set of anchor points.

It is clear that $< A_+ >= A$.

Definition 3.1.4. Constant $\binom{u}{r_1,\ldots,r_s}$ associated with any point Q is called the permutation degree (p-degree) of Q.

Lemma 3.1.2. *Suppose $Q_1, Q_2 \in V$ are such that $x_i(Q_i) \leq x_i(Q_2)$, $1 \leq i \leq u$ and $C(Q_1) < T$, $C(Q_2) \geq T$. Then there exists $P \in A_+$, such that $x_i(Q_1) \leq x_i(P) \leq x_i(Q_2)$.*

Proof. We use induction on $m = \sum_{i=1}^{u} [x_i(Q_2) - x_i(Q_1)]$. If $m = 1$, then $Q_2 \in A_+$ and we are done. Suppose j is a maximum index such that $x_j(Q_1) \neq x_j(Q_2)$. Set $Q = (x_1(Q_1), \ldots, x_j(Q_1) + 1, \ldots, x_u(Q_1))$. Now,

$$x_j(Q_1) < x_j(Q_2) \leq x_{j+1}(Q_2) = x_{j+1}(Q_1)$$

$$\implies x_j(Q_1) + 1 \leq x_{j+1}(Q_1)$$

$$\implies x_j(Q) \leq x_{j+1}(Q) \implies Q \in V_+$$

If $C(Q) \geq T$, then $Q \in A_+$. If $C(Q) < T$, then since

$$\sum_{i=1}^{u} [x_i(Q_2) - x_i(Q_1)] \leq \sum_{i=1}^{u} [x_i(Q_2) - x_i(Q_1)] + 1$$

by induction there exists a $P \in A_+$ with required property. □

Lemma 3.1.3. *Suppose* $(a_1, \ldots, a_r, \ldots, a_u) \in A_+$ *has the following property:*

$$a_r = max\{x_r(P) \mid x_i(P) = a_i, i = 1, \ldots, r-1, P \in A_+\}$$

Then for any $Q \in V_+$*, such that,*

$$\begin{aligned} x_i(Q) &= a_i, \quad 1 \leq i < r \quad \textit{and,} \\ x_r(Q) &> a_r, \end{aligned}$$

$C(Q) > T$.

Proof. Suppose $(a_1, \ldots, a_r, \ldots, a_u) \in A_+$ had the property given in the hypothesis and $(b_1, \ldots, b_u) \in V_+$ with $a_i = b_i$ for $1 \leq i < r$ and $b_r > a_r$. We claim that $C(b_1, \ldots, b_u) \geq T$.

We prove this by contradiction. Suppose $C(b_1, \ldots, b_u) < T$. Set $Q = (a_1, \ldots, a_{r-1}, b_r, c_{r+1}, \ldots, c_u)$ with $c_i = \max(a_i, b_i)$, $r+1 \leq i \leq u$. Note that $b_r = \max(a_r, b_r) \leq \max(a_{r+1}, b_{r+1})$ since $a_r \leq a_{r+1}$ and $b_r \leq b_{r+1}$. Thus $Q \in V_+$. Further, $x_i(Q) \geq b_i \ \forall i$ and $C(Q) \geq C(a_1, \ldots, a_u) \geq T$. Hence by Lemma 3.1.2, there exist $P \in A_+$ with $x_i(Q) \geq x_i(P) \geq b_i$. But for $1 \leq i < r$, $x_i(Q) = b_i = a_i$ and $x_r(Q) = b_r$. Thus we have an anchor point P with $x_i(P) = a_i$, $1 \leq i < r$ and $x_r(P) > a_r$. This is a contradiction. Thus we must have $C(b_1, \ldots, b_r) \geq T$. □

Conversely,

Lemma 3.1.4. *For any* $Q \in \{V_+ \setminus A_+\}$ *with* $C(Q) > T$*, there exists* $(a_1, \ldots, a_r, \ldots, a_u) \in A_+$,

such that,

$$\begin{aligned} x_i(Q) &= a_i, \quad 1 \le i \le r-1 \quad \textit{and,} \\ x_r(Q) &> a_r. \end{aligned}$$

Proof. Suppose $(b_1,\ldots,b_u) \in \{V_+ \setminus A_+\}$ with $C(b_1,\ldots,b_u) \ge T$. Choose a point $(a_1,\ldots, a_r,\ldots,a_u) \in A_+$ such that r is a maximum integer such that $a_i = b_i$, $1 \le i < r$ and $a_r \ne b_r$.This condition is vacuous if $r = 1$. We claim $a_r < b_r$.

We prove by contradiction. Suppose $a_r > b_r$. Set $Q = (a_1,\ldots, a_{r-1}, b_r, c_{r+1}, \ldots, c_u)$ with $c_i = \min(a_i, b_i)$. It is easy to show that $Q \in V_+$. Since $C(Q) \le C(a_1,\ldots,a_u)$, for any j,

$$C(x_i(Q),\ldots,x_j(Q)-1,\ldots,x_u(Q)) \le C(a_1,\ldots,a_j-1,\ldots,a_u)$$

If $(a_1,\ldots,a_u) \in A_+$, then there exists j such that $C(a_1,\ldots,a_j-1,\ldots,a_u) < T$ by definition of anchor point. Hence, $C(x_i(Q),\ldots,x_j(Q)-1,\ldots,x_u(Q)) < T$. Thus if $C(Q) \ge T$, then Q is an anchor point, which contradicts the minimality of r. Thus $C(Q) < T$. Also $x_i(Q) \le b_i$. Now we use Lemma 3.1.2 to find $P \in A_+$, such that $x_i(Q) \le x_i(P) \le b_i$. Since $x_i(Q) = b_i = a_i$, $1 \le i \le r$ and $x_r(Q) = b_r$, we found $P \in A_+$ such that $x_i(P) = b_i$ for $1 \le i \le r$. This violates the maximality of $(a_1,\ldots,a_u)$. Thus we must have $a_r < b_r$. □

Definition 3.1.5. Let $(a_1,\ldots,a_r,\ldots,a_u) \in A_+$ be an anchor point of Lemma 3.1.3. Consider,

$$\tau(a_1,\ldots,a_r) = \{P \in V \mid x_i(\overline{P}) = a_i, 1 \le i < r, \text{ and } x_r(\overline{P}) > a_r\}$$

Lemma 3.1.5. *Suppose $P_1, P_2 \in A_+$ are such that,*

$$x_r(P_1) = max\{x_r(Q) \mid x_i(Q) = x_i(P_1), 1 \leq i < r, Q \in A_+\}$$

$$x_s(P_2) = max\{x_s(Q) \mid x_i(Q) = x_i(P_2), 1 \leq i < s, Q \in A_+\}$$

Then $\tau(x_1(P_1), \ldots, x_r(P_1))$ and $\tau(x_1(P_2), \ldots, x_s(P_2))$ are disjoint or identical.

Proof. We will show that if $(c_1, \ldots, c_u)$ is in,

$$\tau(x_1(P_1), \ldots, x_r(P_1)) \bigcap \tau(x_1(P_2), \ldots, x_s(P_2))$$

then $x_i(P_1) = x_i(P_2)$ for $r = s$ and $1 \leq i \leq r$.

We can assume that $r \leq s$. If $r < s$, then we have $x_r(P_1) \geq x_r(P_2)$ since $x_i(P_1) = x_i(P_2) = c_i$ for $1 \leq i < r$. However by Lemma 3.1.4 we know that $x_r(P_1) < c_r = x_r(P_2)$. This is a contradiction. Hence we must have $r = s$ and $x_r(P_1) = x_r(P_2)$. □

Lemmas 3.1.3 and 3.1.5 show that for the right choices of $\{a_1, \ldots, a_r\}$, $1 \leq r \leq u$,

$$\{Q \in V_+ \mid C(Q) \geq T\} \equiv A_+ \dot{\bigcup}_{(a_1,\ldots,a_r)} \tau(a_1, \ldots, a_r)$$

3.2 Counting Points in Slice

Algorithm 5 illustrates the algorithm used to compute the points with cost $\geq T$ in V using only A_+. Below we give some definitions and formulae used in the Algorithm 5.

Definition 3.2.1. If $part_u = \{\text{paritions of } u\}$, then for $E \in part_u$, $E = s_1^{p_1} s_2^{p_2} \dots s_r^{p_r}$. Also,

$$E_n = \sum_{i=1}^{r} p_r \tag{3.1}$$

$$mult_E = \binom{E_n}{p_1, \dots, p_r} \tag{3.2}$$

$$perm_E = \binom{u}{\underbrace{s_1, s_1 \dots}_{p_1} \cdots \underbrace{s_r, s_r \dots}_{p_r}} \tag{3.3}$$

Let,

$$F_u = \sum_{E \in part_u} mult_E \times perm_E \times D_F \qquad D_F = \begin{cases} 1 & E_n = 1,2 \\ 0 & \text{otherwise} \end{cases} \tag{3.4}$$

$$R_u = \sum_{E \in part_u} mult_E \times perm_E \times D_R \qquad D_R = \begin{cases} 1 & E_n = 2,3 \\ 0 & \text{otherwise} \end{cases} \tag{3.5}$$

$$Q_{j,E_n} = \begin{cases} 0 & E_n < 3 \\ \dfrac{1}{(E_n-3)!} & E_n = 3 \\ \dfrac{1}{(E_n-3)!} \displaystyle\prod_{m=0}^{E_n-4} (j - E_n + 3 + m) & \text{otherwise} \end{cases} \tag{3.6}$$

3.3 Complexity of Counting Points in Slice

The computational complexity of Algorithm 6 is dominated by line 19. Both F_u and R_u requires the computation of all unrestricted paritions of u. The number of unre-

Algorithm 5 *Anchors2PointsSlice*(S, n, r)

1: $\{S$ is the $k \times U$ matrix of k of slice anchor points$\}$
2: $\{n$ is the max value of any element in an anchor point$\}$
3: $\{r$ is a 2×2 matrix where $[rs\ cs; re\ ce]$ define the upper left hand corner (rs, cs) and (re, ce) define the lower right hand corner of the space in A_+ that is currently being processed$\}$
4: *result* = Anchor2PointsSliceRecur(S, n, r)
5: **for all** p, slice anchor point in S **do**
6: *result* = *result* + permutation degree of p
7: **end for**
8: return *result*

Algorithm 6 *Anchors2PointsSliceRecur*(S, n, r)

1: $\{S$ is the $k \times U$ matrix of k of slice anchor points$\}$
2: $\{n$ is the max value of any element in an anchor point$\}$
3: $\{r$ is a 2×2 matrix where $[rs\ cs; re\ ce]$ define the upper left hand corner (rs, cs) and (re, ce) define the lower right hand corner of the space in A that is currently being processed$\}$
4: $\{$If the final value of a summation variable is smaller than its initial value, let that summation be zero.$\}$
5: **if** S is empty **then**
6: return 0
7: **end if**
8: $[rs\ cs; re\ ce] \Leftarrow r$
9: **if** $ce > cs$ **then**
10: $r1 = makegroups(A, r)$
11: **for all** *ri* such that *ri* is a 2×2 group specification in $r1$ **do**
12: $result = result + Anchors2PointsSliceRecur(S, n, ri)$
13: **end for**
14: **end if**
15: $femax$ =max(cs^{th} column from rows rs to re in A)
16: $u = ce - cs + 1$
17: $k = n - femax - 1$
18: $\{F_u, R_u$ and Q_{j,E_n} below are from Equation (3.4), (3.5) and (3.6) respectively.$\}$
19:

$$fsum = 1 + kF_u + \frac{(k-1)k}{2}R_u + \sum_{E \in part_u} \left[mult_E \times perm_E \times \sum_{j=2}^{k-1} \left(\frac{(k-j)(k+1-j)}{2} Q_{j,E_n} \right) \right]$$

20: d = denominator of multinomial coefficient of $S(re,1), S(re,2), \ldots, S(re, cs-1)$
21: $rp = \dfrac{U!}{d \times u!} \times fsum$
22: return $result + rp$

stricted paritions of u, $P(u)$ is given by [13] as

$$P(u) \approx \frac{e^{2\pi\sqrt{2u/3}}}{4u\sqrt{3}}$$

Assuming that each paritions can be generated in constant time, since $u! = O(u^u)$ the complexity of line 19 in Algorithm 6 is

$$\begin{aligned} &= O\left(u\frac{e^{2\pi\sqrt{2u/3}}}{4u\sqrt{3}}\right) + \left[O\left(\frac{e^{2\pi\sqrt{2u/3}}}{4u\sqrt{3}}\right)(unO(u^u))\right] \\ &= O(nu^u e^{2\pi\sqrt{2u/3}}) \end{aligned} \tag{3.7}$$

A_+ has fewer points than A by a factor of $u!$ hence in the worst case the number of anchor points is $a = \frac{u(n-1)^{(u-1)}}{O(u^u)}$.

Continuing the complexity analysis from Section 2.7, the overall complexity with constant u and worst case a,

$$= O\left(n\log n + \frac{u(n-1)^{2(u-1)}}{u^{2u}} + n\right) \tag{3.8}$$

$$= O(n^{2u}) \tag{3.9}$$

Hence the *asymptotic* complexity does not change. This is to be expected since the reduction is based on u and we let u be the constant.

However, with constant n, the overall worst case complexity is,

$$\begin{aligned} &= 0\left(\begin{array}{l} u^3+\dfrac{u(n-1)^{(u-1)}}{u^u}3^u u^2+\dfrac{u^2(n-1)^{2(u-1)}}{u^2 u}+ \\ \dfrac{u(n-1)^{(u-1)}}{u^u}u\log\left(\dfrac{u(n-1)^{(u-1)}u}{u^u})\right)+nu^u e^{2\pi\sqrt{2u/3}}\end{array}\right) \\ &= O\left(\frac{n^{2u}}{u^{2u}}\right) \end{aligned}$$

Here we can see a substantial reduction in complexity due to the slice algorithms.

3.4 Examples

In this section we illustrate the technique of counting points using two examples.

3.4.1 Example 1

Consider costs $[2,3,4,9,12]$ with $U=3$ and $T=10$. We omit the detail on obtaining A_+ since the procedure is fairly straightforward. Figure 3.1 follows the style of Figure 2.2 and show the counting of points using the anchor points in A_+.

For each group returned by Algorithm 3, Algorithm 6 computes rp in line 21. We now show the computation of $rp_1,\ldots,rp_7$.

Computing rp_1

Here, $femax=4, u=1, k=0, part_u=\{1^1\}$. Now for each $E\in part_u$ we evaluate $mult_E$(Equation (3.2)) and $perm_E$ (Equation (3.3)). For $E=1^1$, $E_n=1$ (Equation (3.1)) and,

$$mult_E=\binom{1}{1}$$

$$perm_E = \binom{1}{1}$$

Hence using Equation (3.4),

$$F_1 = (1 \times 1 \times 1) = 1 \tag{3.10}$$

Similarly using Equation (3.5),

$$R_1 = (1 \times 1 \times 0) = 0 \tag{3.11}$$

Hence from Algorithm 6 line 19

$$fsum = 1 + 0 + 0 + 0 \tag{3.12}$$

The last term in Equation (3.12) is 0 since in that term the final condition of the summation (-1) is less that the initial value (2). In line 20, d is the denominator of the multinomial coefficient of 1,1, which is the denominator of $\binom{2}{2}$. Hence $d = 2!$ From Algorithm 6 line 21

$$rp_1 = \frac{3!}{2!1!} fsum = 3$$

Computing rp_2

Similar to rp_1, here $femax = 4, u = 1, k = 0, part_u = \{1^1\}$. However, $d = 1!1!$.

Hence,

$$rp_2 = \frac{3!}{1!1!!1!} fsum = 6$$

Computing rp_3

Here, $femax = 3, u = 1, k = 1, part_u = \{1^1\}$. $mult_E$ and $perm_E$ identical rp_1.

$$fsum = 1 + 1 + 0 + 0 \tag{3.13}$$

Hence,

$$rp_3 = \frac{3!}{1!1!1!} fsum = 12$$

Computing rp_4

This computation is similar to rp_3 except that $d = 2!$. Hence,

$$rp_4 = \frac{3!}{2!1!} fsum = 6$$

Computing rp_5

In this case, $femax = 3, u = 2, k = 1, part_u = \{1^2, 2^1\}$ and $d = 1!$. Since there are two paritions of u,

$$F_2 = \left[\binom{2}{2} \binom{2}{1,1} 1 \right] + \left[\binom{1}{1} \binom{2}{2} 1 \right] = 3 \tag{3.14}$$

$$R_2 = \left[\binom{1}{1} \binom{2}{1,1} 1 \right] + \left[\binom{1}{1} \binom{2}{1} 0 \right] = 2 \tag{3.15}$$

Hence,

$$fsum = 1 + 1 \cdot 3 + 0 \cdot 2 + 0 = 4$$

$$rp_5 = \frac{3!}{1!2!} fsum = 12$$

Computing rp_6

Here, $femax = 2, u = 2, k = 2, part_u = \{1^2, 2^1\}$ and $d = 1!$. Using Equations (3.14) and (3.15),

$$fsum = 1 + 2 \cdot 3 + 1 \cdot 2 + 0 = 9 \tag{3.16}$$

Hence,

$$rp_6 = \frac{3!}{1!2!} fsum = 27$$

Computing rp_7

Finally, here $femax = 2, u = 3, k = 2, part_u = 1^3, 1^1 2^1, 3^1$ and $d = 1!$. So,

$$F_3 = \left[\binom{3}{3} \binom{3}{1,1,1} 0 \right] + \left[\binom{2}{1,1} \binom{3}{1,1} 1 \right] + \left[\binom{1}{1} \binom{3}{3} 1 \right] = 7 \tag{3.17}$$

$$R_3 = \left[\binom{3}{3} \binom{3}{1,1,1} 1 \right] + \left[\binom{2}{1,1} \binom{3}{1,1} 1 \right] + \left[\binom{1}{1} \binom{3}{3} 0 \right] = 12 \tag{3.18}$$

$$fsum = 1 + 2 \cdot 7 + 1 \cdot 12 + 0 = 27$$

Hence,

$$rp_7 = \frac{3!}{1!3!} fsum = 27$$

Line 5 in Algorithm 5 returns the sum of all rp_i, $1 \leq i \leq 7$ and in line 7, the p-degree of each slice anchor point is added. The the overall number of points $\geq T$, 108, is returned in line 9. The p-degree of each point in A_+ is shown in the rightmost column in Figure 3.1.

We can see that Equations (3.4) and (3.5) are independent of k and can be reused

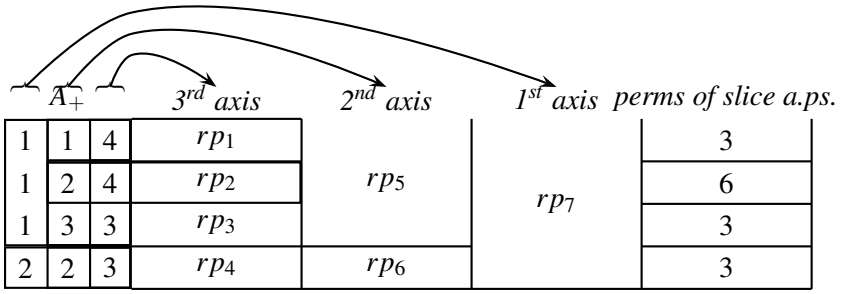


Figure 3.1: Illustrating Algorithm 5, for Example 1

in computing rp of a column from Figure 3.1

3.4.2 Example 2

Now, we present another example similar to that in Section 3.4.1, but with $T = 8$. This example has fewer slice anchor points and exercises computation of Q_{j,E_n} (Equation (3.6)). Figure 3.2 illustrates this computation.

Consider costs $[2,3,4,9,12]$ with $U = 3$ and $T = 8$.

Computing rp_1

Here, $femax = 3$, $u = 1$, $k = 1$, $part_u = \{1^1\}$. $fsum$ in this case is identical to Equation (3.12). Hence,

$$rp_1 = \frac{3!}{2!1!} fsum = 6$$

Computing rp_2

Here, $femax = 2$, $u = 1$, $k = 2$, $part_u = \{1^1\}$ and $fsum$ is again identical to Equation (3.12). Hence,

$$rp_2 = \frac{3}{1!1!1!} fsum = 18$$

Computing rp_3

Here, $femax = 2$, $u = 2$, $k = 2$, $part_u = \{1^2, 2^1\}$ and $fsum$ is identical to Equation (3.16). Hence,

$$rp_3 = \frac{3!}{1!2!} fsum = 27$$

Computing rp_4

Here, $femax = 1$, $u = 3$, $k = 3$, $part_u = \{1^3, 1^1 2^1, 3^1\}$. Using F_3 and R_3 from Equations (3.17) and (3.18) respectively,

$$\begin{aligned} fsum &= 1 + 3 \cdot 7 + 3 \cdot 12 + \left\{ \binom{3}{3} \binom{3}{1,1,1} \left[\frac{(3-2)(3+1-2)}{2} \frac{1}{(3-3)!} \right] \right\} \\ &\quad + \left\{ \binom{2}{1,1} \binom{3}{1,2} \left[\frac{(3-2)(3+1-2)}{2} \cdot 0 \right] \right\} \\ &\quad + \left\{ \binom{1}{1} \binom{3}{3} \left[\frac{(3-2)(3+1-2)}{2} \cdot 0 \right] \right\} \\ &= 1 + 21 + 36 + 6 + 0 + 0 \\ &= 64 \end{aligned}$$

Hence,

$$rp_4 = \frac{3!}{3!} fsum = 64$$

The rightmost column of Figure 3.2 shows the p-degree of each point in A_+. Adding up all rp_i, $1 \leq i \leq 4$ and the p-degrees, give us 121, the total number of points with cost ≥ 8.

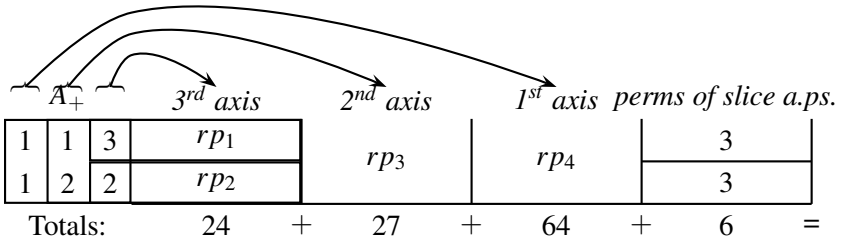


Figure 3.2: Illustrating Algorithm 5, for Example 2

CHAPTER 4

APPLICATIONS

In this chapter we demonstrate the application of our techniques. In Section 4.1 we use simulations to demonstrate the steps for application of our techniques to component-based environments. In Section 4.2 we outline and report on experiments that demonstrate the application of our techniques to ad-hoc service environments.

4.1 Application to Component-Based Environments

In this section we use simulations to demonstrate a usage of our technique in component-based environments. Consider a component-based environment of 5 components with costs [5, 9, 10, 13, 20]. Let the adjacency matrix be,

$$A = \begin{bmatrix} 1 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 \end{bmatrix},$$

the system's total resources be exceeded at 250 and there be 3 concurrent users. Figure 4.1 illustrates the max sequences lengths computed (Y-axis) by our algorithms for varying system designer defined probabilities (X-axis).

The problem was solved using following steps:

1. Using historical information (or just 1 if no historical information) we selected a starting maximum sequence length.

2. For each length from 1 to current maximum length, we used the formulae developed in Section 2.2 to find the *average* cost of all sequences of that length.

3. We then used the algorithms in Section 2.5 to compute the required probability. The average costs calculated in the previous step were the input costs for these algorithms, the number of users 3 became the dimension of the virtual space. Then we counted "points" with cost ≥ 250. The number of points computed divided by the total points gave us the probability Pr.

4. If Pr was the required probability or was as close as possible to the required probability, we were done, else we modified the length (decrease if $Pr >$ required probability else increase) and repeated from step 2.

To verify our results, we simulated 3 concurrent users with each user generating sequences up to a maximum suggested length. Then we added the costs of those sequences and considered it to be the systems instantaneous total cost. We generated 400 such instantaneous costs and then computed the probability that any instantaneous cost was over 250 (the total system resources). This was considered one experiment. We repeated this exper-

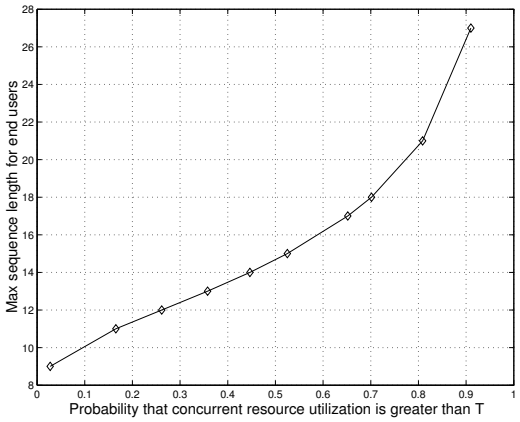


Figure 4.1: Max sequences lengths

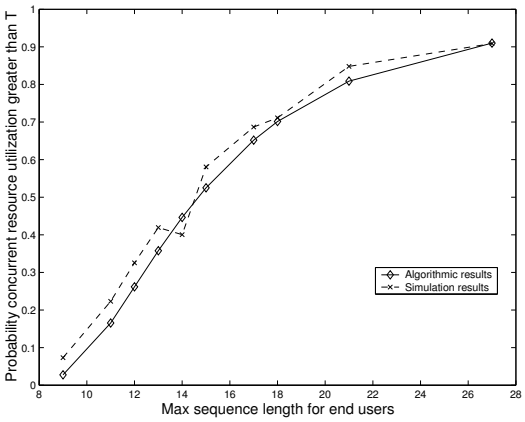


Figure 4.2: Probability errors

iment 4 times and reported the averaged probabilities. Figure 4.2 illustrates the simulation and algorithmic results. The two results compare favorably, the mean error is 0.04 while the maximum error is 0.06.

4.2 Application to Ad-hoc Environments

In this section we estimate the performance of a set of services deployed at a server. Using exact counting algorithms might not always be the best way to estimate service performance. While the counting algorithms are efficient at returning exact results, useful approximations can often be obtained using computationally cheaper Markov Chain Monte Carlo methods [16, 24]. However we choose to use our exact counting algorithms instead of approximation algorithms since the value that we are computing, the probability of overload, is uncertain by definition and using approximation algorithms can only worsen this uncertainty. The purpose of selecting this application is that this application showcases the flexibility of our algorithms and proves that meaningful measures can be associated with the cost C_i, number of users U and capacity measure T used by the counting algorithms.

The application consists of a set of services deployed at a server such that the services can be accessed by multiple users concurrently. Examples of such applications are Web Services and remote procedure calls. Every service is associated with a server based Service Level Agreement (SLA). The server based SLA of a service is the maximum time that the server is allowed to take to complete a service call. We say that a service call, or simply service, *passes* if the service call meets its SLA, otherwise it *fails*. User perceived performance of services at a server is the percentage of service calls made by a user that fail. Clearly, user perceived performance is determined by server capacity, computational requirements of services, SLAs and number of concurrent users. In this application we restrict ourselves to CPU bound services, each executing in its own process with round-robin scheduling and equal time slices.

The rest of this section is organized as follows. Section 4.2.1 deals with assigning server capacity measures and service costs, Section 4.2.2 discusses the procedure of estimating and simulating concurrent users. Section 4.2.3 details the case when services *fail*, i.e. fail to meet their SLA. Section 4.2.4 illustrates how our model can be used to exactly compute the probability that, given the parameters determined in Section 4.2.1, a user's service call will fail. Section 4.2.5 illustrates how our counting algorithms can be efficiently adapted to changing service access patterns and how service costs can be easily ported to new servers. Finally, we compare our approach with others in Section 4.2.7.

4.2.1 Assigning Costs

Here we define cost of a service as the *rate* of resource consumption. Hence, suppose a service requires X number of CPU cycles for completion, the cost of that service

is X/S cycles per second (cps), where S is the required SLA for that service in seconds. Subsequently, server capacity is the rate at which resource are available for services, measured in cps. Since we are only interested in the relation between resource consumption and availability, i.e. between service cost and server capacity, we do not need to determine the true server capacity or service cost in terms of Million Instructions Per Second (MIPS) or CPU clock cycles. For our experiments we assume server capacity is exceeded at $T = 100$ cps where cycles are not CPU clock cycles but some abstract measure of CPU capacity. Now we can readily determine service costs using the following calibration procedure:

1. Start U concurrent accesses to service S1
2. Wait till all U accesses finish
3. Compute mean time to finish, R_U
4. $C = \frac{R_U * T}{U}$
5. Cost of S1 $= \frac{C}{\text{SLA}}$.

The above procedure should be repeated for a few different values of U to ensure stable cost. Recall that we assume round-robin scheduling with equal time slices, which is why we can compute the total cycles consumed by S1 using the relation in Line 4 above.

Table 4.1 tabulates the total cycles consumed and cost for a SLA of 5 seconds for 4 services assuming $T = 100$. These services were calibrated on a 300 MHz Intel Pentium II processor based PC running Linux kernel 2.4.20. These services were calibrated using 5 concurrent accesses. It is worth pointing out that though we imply that each service

Service	Total cycles	Cost
R1	328	65.6
R2	130	26
R3	196	39.2
R4	63	12.6

Table 4.1: Computing service costs with SLA = 5 seconds.

is different, they could in fact be invocations of the same code with different parameters. For purposes of computing resource consumption this distinction is moot. The services calibrated in Table 4.1 simply consume CPU time in `for` loops. In this application we restrict the application to services with equal SLAs.

4.2.2 User Access Discipline

Once a user makes a service call, and the service call passes (i.e. returns on or before the SLA period), the user must wait until the end of the SLA period before the user can make the next service call. This restriction arises from the method of computing service cost. The disadvantage of this access discipline is that users might have to wait until they can call the next service, however in return once a user make a service call the user gets a probabilistic guarantee of success. Such a guarantee is not possible in back-to-back service calls. Figure 4.3 compares the effect of using and not using this access discipline.

In Figure 4.3 we consider the case of 3 users each calling two services. U1 calls services S1 followed by S2, U2 calls S2 followed by S1 and U3 class S1 twice. Required SLA for all services is 30 seconds. Available resources are exceeded at 100 cps. The cost of S1 is 750 and S2 is 1500 and SLA for both is 30 seconds. In the upper part of the figure, U1's call to S1 finished at 22.5 seconds, however due to the access discipline, U1 must wait

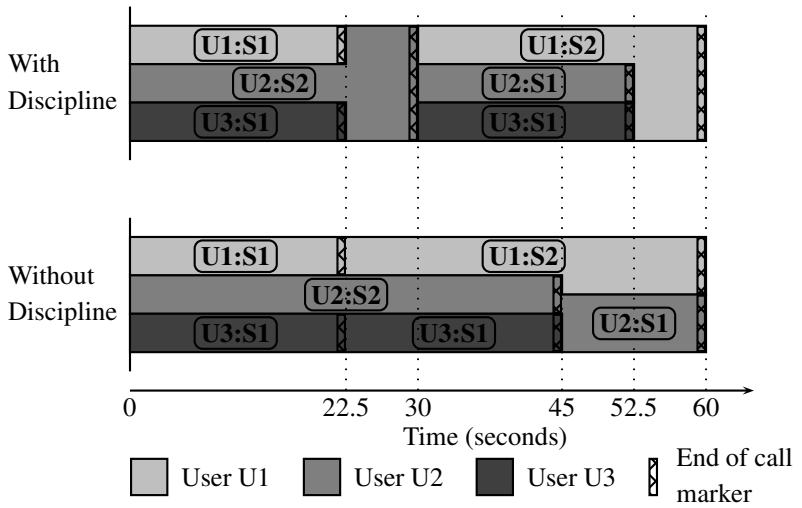


Figure 4.3: Effect of with and without discipline with 3 users.

till the end of its SLA period (30 seconds) before it can make the call to S2. We see that, given the round-robin equal time-slice assumption, without using access discipline the calls to S2 fail to meet the required SLA, whereas with the access discipline all calls pass. This happens because without access discipline, users U1 and U3 fail to "give up" the server resources long enough for U2's S2 call to finish. When using access discipline users U1 and U3 have to wait before they can call their next services, consequently allowing U2's S2 call to finish. The reason for users having to wait up to end of SLA period (and not some other time interval) will be discussed in Section 4.2.3. Our algorithms assume users use access discipline, and in return users get probabilistic service guarantees.

4.2.3 Service Failures

We now look at the context of service failures using the model of the counting algorithms. Consider the following example: Assume 2 users uniformly access 4 services S1, S2, S3 and S4 with costs $[1,3,7,12]$, using the access discipline described in Section 4.2.2.

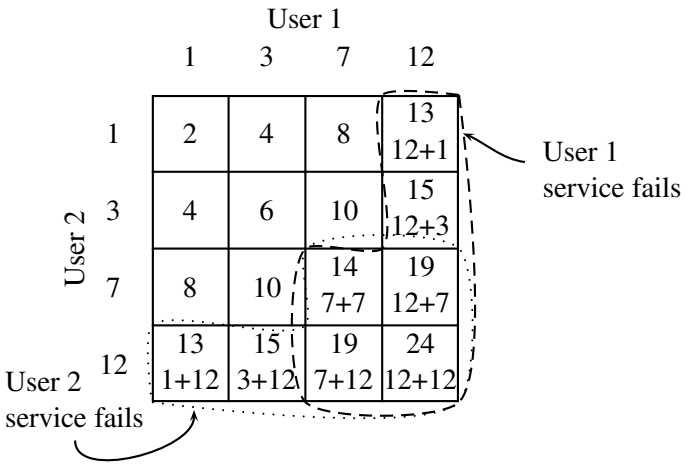


Figure 4.4: Service failure events. $T = 13$.

Let resources be exceeded at $T = 13$ cps. Then, as per our model the probability that resource demand exceeds T is $8/16$. This virtual space is shown in Figure 4.4. In this figure when, for example, user 1 calls service with cost 12 and user 2 calls service with cost 3, the total resource consumption exceeds T. Since there are 2 users in the system, each will get up to $T/2 = 6.5$ cps and hence the call by user 1 will succeed but the call by user 2 will fail. In general, given U users, a user's service will fail in the event when the sum of costs is greater than T *and* the cost of that user's service is greater that T/U. In the example of Figure 4.4, the dashed line encloses the events when User 1's services fail, hence the probability of User 1's services failing is $6/16 = 0.375$. In the next section we show how the counting algorithms can be used to determine a user's failure rate.

4.2.4 User Perceived Performance

Recall, that the counting algorithms work in two stages. In the first stage we determine *anchor points* in the virtual space V. V has U dimensions and the number of points on each axis is equal to the number of services N. In the second stage we use these

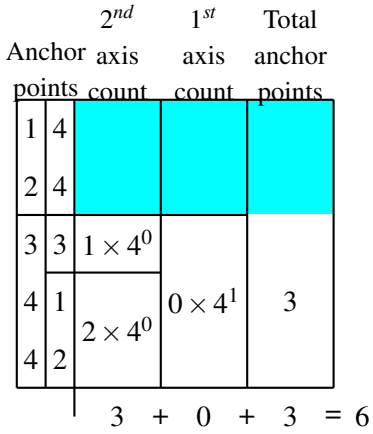


Figure 4.5: Counting points $\geq T$ and a user's costs $\geq T/U$. Shaded area is discarded a.ps.

anchor points to compute G, number of points in V with value $\geq T$. Then from the server perspective, given U concurrent users, the probability that at least one user's service call will fail is $\frac{G}{N^U}$. User perceived performance in terms of probability of service call failure, (Section 4.2.3) can be computed efficiently using the same set of anchor points used to compute performance from the server's perspective. All that needs to be done is to discard anchors points with cost value of any fixed co-ordinate $\leq T/U$. Applying the second stage to this new set of anchor points will give us the probability that sum of costs $\geq T$ and cost of a user's service $\geq T/U$. This is the required probability of a user's service call failing.

We illustrate the procedure of determining user perceived performance by continuing the example from Section 4.2.3. Using co-ordinate (1,1) for the top left hand corner of Figure 4.4, the anchor points determined for $T = 13$ using the first stage algorithms are: $A = \{(1,4),(2,4),(3,3),(4,1),(4,2)\}$. Since there are 2 users, per user share is $T/2 = 6.5$. Given (sorted) services costs $[1,3,7,12]$, we see that the 1^{st} and 2^{nd} costs, 1 and 3 are ≤ 6.5, hence we discard all anchors points whose 1^{st} coordinate is 1 or 2. We are left with $A' = \{(3,3),(4,1),(4,2)\}$, applying the second stage of the algorithm to A' (Figure 4.5) we

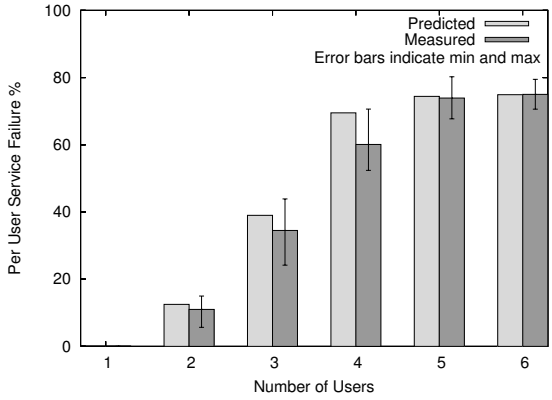


Figure 4.6: User perceived performance: Service R1, R2, R3, R4 with uniform access.

get the number of points with cost $\geq T$ and a fixed user's cost $\geq T/U$ as 6. Hence, from a given user's perspective the probability that a service will fail is $6/16 = 0.375$.

We tested our algorithm's predictions using the 4 services R1,R2,R3 and R4 from Section 4.2.1. Each experiment was conducted for 520 seconds and the mean of services failures by all users was taken. Each experiment was repeated 25 times and the average taken. Figure 4.6 summarizes the results. The results closely match predictions. The measured per user loss rate correctly followed the loss rates predicted by the counting algorithms.

4.2.5 Service Access Patterns

In Section 4.2.4 we assumed that users access services uniformly, i.e. each service is accessed with equal probability. However, our model can easily adapt to *arbitrary* access patterns. The patterns *do not* need to fit any known distribution and do not affect the first stage of our algorithm. The anchors points do not need to be recomputed for changing access patterns. The only change needed is in the second stage. This change is illustrated

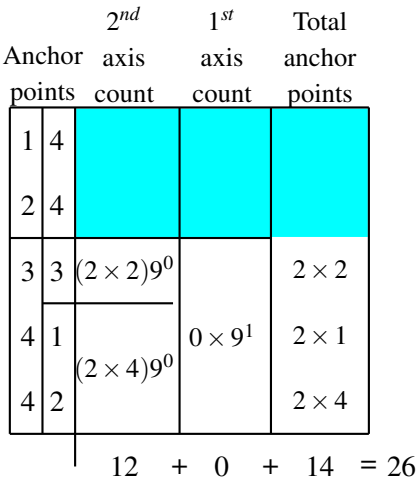


Figure 4.7: Counting points $\geq T$ and a user's costs $\geq T/U$ with cost weights $[1,4,2,2]$.

by continuing the example from Section 4.2.4, but this time the 4 services S1,S2,S3 and S4 are weighted (and normalized to smallest integer) as 1, 4, 2, 2, respectively. Hence, for example S2 is twice as likely to be accessed as compared to S4. If P is the number of anchor points, and normally $P \gg U$, the worst case computational complexity of $O(PU^2)$ for arbitrary access patterns is only slightly greater compared to $O(PU)$ in the uniform access case (Figure 4.5). This increased complexity is independent of the access pattern. Figure 4.7 shows the second stage of the algorithm. The multiplier 9 in each group of Figure 4.7 is the sum of weights. With the given access pattern the probability that a user service will fail is $26/81 = 0.32$. This reduction in probability of failure as compared to uniform access (0.375) is intuitive since now users are more likely to access cheaper services.

Figure 4.8 continues the experiment from Sections 4.2.1 and 4.2.4, but this time with access weights 2,1,2 and 4 for R1, R2, R3 and R4 respectively. Again, we see that the measured loss rate follows the loss rate predicted by the counting algorithms.

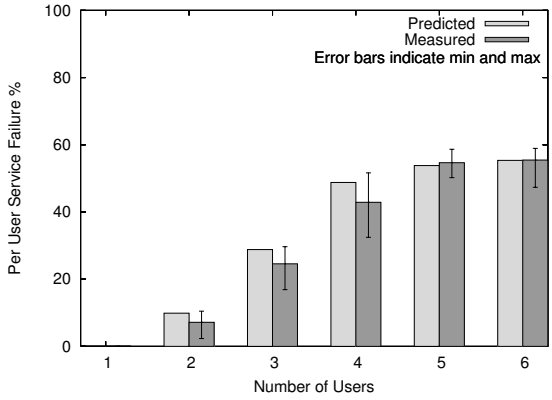


Figure 4.8: User perceived performance: Access weights 2,1,2,4.

4.2.6 Porting Costs

Another feature of our model is that costs can be easily ported between different servers. In Section 4.2.1 we described the process of calibrating services, with the assumption that server capacity $T = 100$. In order to determine user perceived performance when services are moved to another server, we could re-calibrate every service. However, it is much more efficient to keep service costs the same (as with $T = 100$) and change T to a T' that reflects the new server's capacity. The procedure to compute T' is as follows:

1. Choose any service S.

2. Compute C_o, the total cycles of S with server $T = 100$.

3. Compute C_n, the total cycles of S with new server (assume $T = 100$).

4. Capacity of new server $T' = 100C_o/C_n$

Figure 4.9 shows the results from porting $T = 100$ for R1, R2, R3 and R4 onto a 550 MHz SUN SPARCv9 based workstation running SunOS 5.9. Service R1 was recal-

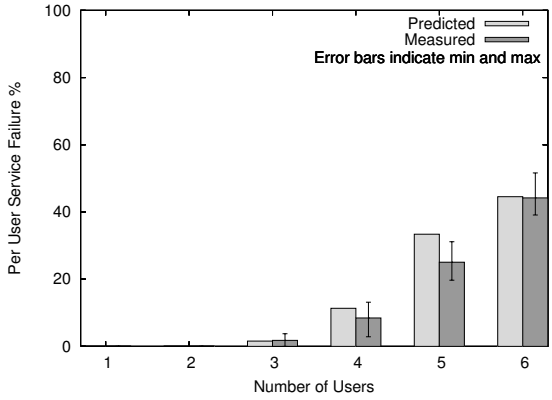


Figure 4.9: User perceived performance: Services R1, R2, R3, R4 on a faster workstation

ibrated for this workstation resulting in $C_n = 190$, ($C_o = 328$ from Table 4.1), hence the capacity of this workstation was computed (and rounded) to be $T' = 100 \times 328/190 = 172$. In both cases (Pentium and SPARC) the same server source code (written in C) was compiled using the gcc compiler. We notice that the faster SPARC workstation could handle the same number of users as compared to the Pentium computer with a lower per user service failure. The counting algorithms correctly predicted these lower loss rates without having to recalibrate every service.

4.2.7 Related Work

Several mechanisms exist for modeling resource consumption and probabilistically meeting SLAs [5, 30, 7]. SHARP [7] uses a ticket based mechanism to allocate resources. However, by "oversubscribing" tickets the SHARP architecture is able to achieve gains in resource utilization at the server being managed. They report a utilization increase of 30% by allocating 1.5 times the available resources. Upon allocating 2 times the available resources, utilization reaches close to 100% but services requests begin being denied.

Our algorithms are supplementary to this work in that we focus on admitting users (rather than allocating resources/tickets) and can precompute and provide the system admininstrator with a continium of the number of users and expected service failure rates.

Urgoankar et. al. [30] report that applications capsules (components) being allocated resources "overbooked" by a factor or 0.01 to 0.05 lead to a 2 to 5 fold improvement in resource utilization. They use an operating system specific kernel profiling toolkit to determine the resource usage and distribution of several applications and then report on how many such applications can be supported at a hosting facility.

Our approach is similar to the above approaches in that we provide efficient mechanisms for computing the probabilities that can aid in such probabilistic guarantee methods. However, as shown in the application presented, our approach does not need OS specific profiling techniques since we do not need to find the actual resources consumed but rather the fraction of the available resources consumed by a service. This greatly simplifies our profiling technique (Section 4.2.1) and makes it portable (Section 4.2.6). Also, our focus is on per-user perceived behavior in addition to server utilization.

In this section we illustrated the application of the counting algorithms to predicting the number of users that can be supported at a server. We demonstrated that the model is flexible enough to efficiently accommodate changing user access patterns without having to recompute anchor points. Also, we showed that the model is portable since we can efficiently predict user perceived performance on changing server capacities without having to recalibrate all services.

CHAPTER 5

CONCLUSION

5.1 Future Work

While the motivation for this research is to create resource-bounded network services, it is clear that the algorithms developed are general enough to be applied to many combinatorial enumeration problems. As part of future work we plan to investigate the application of the counting algorithms to the following variants of classic problems:

1. **(Knapsack Problem)**: Given a set $I = \{i_1, \ldots, i_n\}$ of n types of items with weight of $i_j = w_j$, $1 \leq j \leq n$, a Knapsack with capacity T and infinite number of instances of any type. Find the number of ways to fill the Knapsack with exactly U items without breaking.

2. **(Frobenius Coin Exchange Problem)**: Given a set $I = \{i_1, \ldots, i_n\}$ of n coin types with value of $i_j = w_i$, $1 \leq j \leq n$. Find the total number of ways we can make change of amounts greater than or equal to T using exactly U coins.

Our approach to counting is essentially that of volume estimation in a polytope. Computing volume by counting lattice points in *convex* polytopes is a widely studied area [1]. Reference [2] presents and practically tests algorithms for a wide range of *convex* polytopes, while [12] surveys applications of such computations and provides in-depth treatment of a few. We define a directionally convex polytope as a polytope that is convex only if lines parallel to a given direction are considered. The polytope defined by anchor points is *directionally convex* and is more general than strictly convex polytopes. It would be interesting to see if our algorithms can be used to efficiently compute the volume of any arbitrarily specified convex polytope.

The work on sequences of components presented in Sections 2.2 and 2.3 could be extended to *trees*. The root of the tree would correspond to the first component of a sequence but then a tree would allow for a choice between subsequent components. The *cost* of a tree can be defined as the cost of a most expensive sequence in the tree. Such an extension might lead of a better representation of services since it will allow for conditional execution of components.

5.2 Conclusion

In this dissertation we presented and proved efficient counting algorithms that allow the construction of frameworks for creating resource-bounded services. Such frameworks provide probabilistic guarantees. Though the primary motivation for such a framework was to implement network services, we showed that it can also be used for provisioning services at servers. We showed through simulation and experimentation that our

algorithms can be applied to both component-based and ad-hoc service environments. Finally we outlined how the counting algorithms are general enough to be applied to some combinatorial enumeration problems.

APPENDIX A

Experimental Data

Users	Predicted	Measured		
		min	mean	max
1	0	0	0	0
2	12.5	5.66	11	14.97
3	39	24.17	34.5	43.88
4	69.5	52.39	60.1	70.62
5	74.4	67.71	73.9	80.23
6	74.9	70.60	75	79.47

Table A.1: Data for Figure 4.6. Per user service failure percentage with uniform access.

Users	Predicted	Measured		
		min	mean	max
1	0	0	0	0
2	9.88	2.34	7.16	10.48
3	28.81	16.89	24.57	29.67
4	48.78	32.45	42.87	51.65
5	53.82	50.21	54.66	58.67
6	55.36	47.33	55.46	58.92

Table A.2: Data for Figure 4.8. Per user service failure percentage with weighted access.

Users	Predicted	Measured		
		min	mean	max
1	0	0	0	0
2	0	0	0	0
3	1.56	0	1.79	3.76
4	11.33	2.85	8.46	13.14
5	33.39	19.69	25.04	31.14
6	44.55	39.09	44.19	51.63

Table A.3: Data for Figure 4.9. Per user service failure percentage after porting services to faster workstation.

REFERENCES

[1] A. Barvinok. Lattice points and lattice polytopes. In Jacob E. Goodman and Joseph O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 7, pages 133–152. CRC Press LLC, 1997.

[2] B. Bueler, A. Enge, and K. Fukuda. Exact volume computation for polytopes: A practical study. In G. Kalai and G. M. Ziegler, editors, *Polytopes-Combinatorics and Computation*. Birhauser Verlag, Basel, 2000.

[3] Y. Chae, S. Merugu, E. Zegura, and S. Bhattacharjee. Exposing the network: Support for topology sensitive applications. In *Proceedings of IEEE OpenArch 2000*, March 2000.

[4] Dan Decasper, Zubin Dittia, Guru Parulkar, and Bernhard Plattner. Router plugins: a software architecture for next-generation routers. *IEEE/ACM Transactions on Networking (TON)*, 8(1):2–15, 2000.

[5] Ronald P. Doyle, Jeffrey S. Chase, Omer M. Asad, Wei Jin, and Amin Vahdat. Model-Based Resource Provisioning in a Web Service Utility. In *Fourth USENIX Symposium on Internet Technology and Systems (USITS)*, pages 57–71, Seattle, Washington, USA, March 2003. USENIX.

[6] Bart Duysburgh, Thijs Lambrecht, Bart Dhoedt, and Piet De meester. Data Transcoding in Multicast Sessions in Active Networks. In Hiroshi Yasuda, editor, *LNCS 1942, Second International Working Conference, IWAN 2000, Tokyo, Japan, October 2000. Proceedings*, pages 130–144, 2000.

[7] Yun Fu, Jeffrey Chase, Brent Chun, Stephen Schwab, and Amin Vahdat. SHARP: an architecture for secure resource peering. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 133–148. ACM Press, 2003.

[8] V. Galtier, K. Mills, Y. Carlinet (NIST), S. Bush, and A. Kulkarni (GE). Predicting and Controlling Resource Usage in a Heterogeneous Active Network. In *Proceedings of the DARPA Active Networks Conference and Exposition, IEEE*, pages 511–533, May 2002.

[9] Yitzchak Gottlieb and Larry Peterson. A Comparative Study of Extensible Routers. In *OPENARCH 2002*, pages 51–62, June 2002.

[10] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics*. Addison Wesley, Reading, Massachusetts, 1988.

[11] R.L. Graham, M. Grötschel, and L. Lovász, editors. *Handbook of combinatorics*. Elsevier, 1995.

[12] P. Gritzmann and V. Kleen. On the Complexity of some basic problems in computational convexity: II Volume and mixed volumes. In T. Bistriczky, P. McMullen, R. Schneider, and A.I. Weiss, editors, *Polytopes: Abstract, convex and computational*, pages 373–466. NATO Adv. Sci. Inst. Ser. C Math. Phys. Sci., 1994.

[13] G. H. Hardy and S. Ramanujan. Asymptotic Formulae in Combinatory Analysis. *Proc. London Math. Soc.*, 17:75–115, 1918.

[14] Michael Hicks, Jonathan T. Moore, and Scott Nettles. Compiling PLAN to SNAP. *Lecture Notes in Computer Science*, 2207, 2001.

[15] Michael W. Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles. PLAN: A packet language for active networks. In *International Conference on Functional Programming*, pages 86–93, 1998.

[16] Mark Jerrum and Alistair Sinclair. The Markov Chain Monte Carlo Method: An Approach to Approximate Couting and Integration. In Dorit S. Hochbaum, editor, *Approximation algorithms for NP-hard problems*, chapter 12, pages 482–520. PWS Publishing Co., 1997. ISBN 0-534-94968-1.

[17] Kenneth L. Calvert and James Griffioen and Su Wen . Lightweight Network Support for Scalable End-to-End Services. In *Proceedings of SIGCOMM 2002*, Pittsburg, PA, August 2002.

[18] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[19] Ulana Legedza and John Guttag. Using Network Level Support to Improve Cache Routing. In *Proceedings of 3rd International Web Caching Workshop*, Manchester, England, June 1998.

[20] L. H. Lehman, S. J. Garland, and David L. Tennenhouse. Active Reliable Multicast. In *Proceedings of the 17^{t}h INFOCOM,*, pages 581–589, March 1998.

[21] Jonathan T. Moore, Michael Hicks, and Scott Nettles. Practical programmable packets. In *Proceedings of the 20th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'01)*, April 2001.

[22] Akihiro Nakao, Larry Peterson, and Andy Bavier. Constructing End-to-End Paths for Playing Media Objects. In *2001 IEEE Open Architectures and Network Programming Proceedings*, pages 117–128, Ancorage, AK USA, April 2001.

[23] V. Ramachandran, R. Pandey, and S. Chan. Fair resource allocation in active networks. In *Proceedings of the IEEE International Conference on Computer Communications and Networks (ICCCN)*, pages 468–475, Las Vegas, Nevada, Oct. 2000.

[24] Ravi Kannan. Markov chains and polynomial time algorithms. In *35th IEEE Annual Symposium on Foundations of Computer Science*, pages 656–671, Santa Fe, New Mexico, 1994.

[25] F. Sabrina and S. Jha. A novel architecture for resource management in active networks using a directory service. In *Proceedings of ICT03*, February 2003.

[26] S. Schmid, J. Finney, A.C. Scott, and W.D. Shepherd. Component-Based Active Network Architecture. In *Sixth IEEE Symposium on Computers and Communications (ISCC'01)*, pages 114–122, Hammamet, Tunisia, July 2001.

[27] Beverly Schwartz, Alden W. Jackson, W. Timothy Strayer, Wenyi Zhou, R. Dennis Rockwell, and Craig Partbridge. Smart packets: applying active networks to network management. *ACM Transactions on Computer Systems*, 18(1):67–88, 2000.

[28] Steven Simpson, Mark Banfield, Paul Smith, and David Hutchison. Component selection for heterogeneous active networking. *Lecture Notes in Computer Science*, 2207, 2001.

[29] Richard P. Stanley. *Enumerative Combinatorics*, volume 1 of *Cambridge Studies in Advanced Mathematics 49*. Cambridge University Press, 1997.

[30] Bhuvan Urgaonkar, Prashant Shenoy, and Timothy Roscoe. Resource overbooking and application profiling in shared hosting platforms. *SIGOPS Oper. Syst. Rev.*, 36(SI):239–254, 2002.

[31] D. Wetherall, J. Guttag, and D. Tennenhouse. ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols. In *Proceedings of IEEE OPENARCH 1998*, April 1998.

[32] J. Whaley, M. Martin, and M. Lam. Automatic extraction of object-oriented component interfaces. In *Proceedings of the International Symposium of Software Testing and Analysis*, 2002.

[33] Lidia Yamamoto and Guy Leduc. An Agent-inspired Active Network Resource Trading Model Applied to Congestion Control. In Eric Horlait, editor, *Proceedings of the Second International Workshop on Mobile Agents for Telecommunication Applications (MATA 2000)*, pages 151–170, Paris, France, 2000. Springer-Verlag: Heidelberg, Germany.