Dynamic Memory Model based Framework for Optimization of IP Address Lookup Algorithms

Gene Cheung and Steven McCanne
University of California, Berkeley

Abstract

The design of software-based algorithms for fast IP address lookup targeted for general purpose processors has received tremendous attention in recent years due to its low cost implementation and flexibility. However, all work to date fails to account for the hierarchical memory structure of the processor when designing algorithms. In this work, we propose a dynamic memory model that captures data movement between hierarchical memories and the memory access cost. Using the model, we formulate the design of IP address lookup algorithms as a well-defined optimization problem that minimizes an algorithm's average lookup time. We first show the problem is NP-hard. We then present an optimization framework and associated algorithm based on Lagrange multipliers that terminates in a bounded-error solution. Simulation shows the synthesized algorithm has noticeable performance gain over existing techniques.

1 Introduction

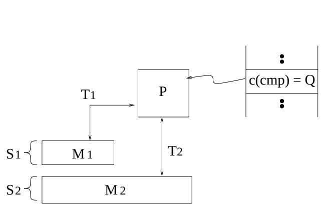
In the Internet Protocol (IP) architecture, *hosts* communicate with each other by exchanging packetized messages or *packets* through a fabric of interconnected forwarding elements called *routers*. Each router consists of input interfaces, output interfaces, a forwarding engine, and a routing table. When an IP packet arrives on an input interface, the forwarding engine performs a *route lookup* — among the set of prefixes in the routing table, it locates the longest prefix that matches the destination address of the IP packet. The entry corresponding to the longest matched prefix determines the output interface through which to forward the packet towards its ultimate destination.

If forwarding performance within a router were infinitely fast, then the overall performance of the network would be determined solely by the physical constraints — i.e., bit rate, delay, and error rate — of the underlying communication links. In practice, however, building high-speed routers is a hard problem and as physical-layer link speeds continue to increase, the route lookup step in the router can easily become the bottleneck in network performance. Thus, to optimize network performance, we must not only optimize the speed of the physical-layer transmission media but also the forwarding performance of the routers. This is the heart of the IP address lookup problem; namely, finding the longest prefix match of an IP address among a finite set of prefixes in a fast and efficient manner.

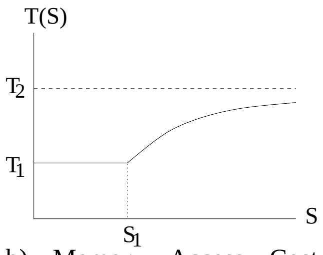
The existing art in this problem domain falls roughly into two categories: i) system approaches where static data structures and algorithms have been designed and tested for a typical set of prefixes at a typical router [1], [2]; and, ii) dynamic optimizations where the performance is enhanced by the particulars of a prefix set [3], [4], [5], [10]. Clearly prefix-set dependent optimizations benefit from data dependent information that is not exploited by data independent techniques, and thus if the dynamic optimizations are not too costly, we should perform such optimizations.

Our work falls into the latter category, but unlike previous work, we exploit *machine dependent* information as well. In particular, we have developed an optimization framework that captures the hierarchical memory characteristics of the underlying processor, and when given a prefix set from a routing table, finds and automatically generates a near-optimal algorithm. By optimal, we mean a prefix decoding algorithm, one that uses a combination of chosen decoding tools, has the minimum average decoding time with respect to our proposed prefix Markov model and machine model. Decoding tools are decoding techniques, such as table lookups and programmed logic, that in combination classify an IP address into one of many classes. Our framework is flexible in two regards: i) in the context of IP address lookup, its generality permits easy adaptation of new decoding tools into the framework; and, ii) IP address lookup can be viewed as a particular instantiation of the framework; other computational optimization problems such as packet classification [6], can potentially be cast as instantiations of the framework.

In section 2, we first discuss two proposed models: a machine model that models the hierarchical memories of a general purpose processor, and a Markov model that models the correlation between prefixes of consecutive IP packets.



a) Dynamic Memory Model



b) Memory Access Cost Function

Figure 1. Dynamic Memory Model for General Purpose Processor

In section 3, we first discuss a common representation of a prefix set, and then formalize the optimization problem based on the two models. In section 4, we show the optimization problem is NP-hard. We present our proposed optimization framework and an associated approximation approach based on Lagrange multipliers in section 5. We discuss specific implementation issues and results in section 6. Finally, we conclude in section 7.

2 Machine and Prefix Modeling

Unlike previous works, our optimization framework optimizes the resulting implementation in a *machine dependent* manner, i.e. the optimal design of the algorithm depends heavily on the characteristics of the underlying processor. To provide a vehicle for optimization, we propose to use an abstract machine model, whose parameters are exploited during optimization; we will first present such a model. Many authors have claimed in the literature that algorithms can benefit from the correlation of destination IP addresses of consecutive packets given the inherent caching mechanism of a general purpose processor. To accurately capture this correlation, we present a simple prefix Markov model.

2.1 Dynamic Memory Model

Modern general-purpose processors use hierarchical memories to enhance performance, where small, fast memories are located near the CPU and larger, slower memories are situated further away. Consequently, the execution speed of a machine instruction that accesses memory depends on the level of memory referenced. A machine model that reflect this characteristic is shown in Figure 1a. If the processor P accesses a datum residing in level 1 memory M_1 (level 2 memory M_2), it incurs memory access time T_1 (T_2). If the instruction does not involve memory access, then the execution time depends on the complexity of the instruction itself. In Figure 1a, we denote the cost of a logical comparison (cmp) as Q. For the chosen machine model, the size of the level 1 memory is S_1, and the size of the level 2 memory is $S_2 = \infty$[1].

Now suppose the size of the data structures of an algorithm in memory, S, is less than or equal S_1. Then the memory access time of a desired datum, $T(S)$, is always T_1, since all the data structures can be loaded into level 1 memory. If $S > S_1$, then the desired datum may not be in level 1 memory. We estimate the memory access time in this case as follow: a size S_1 portion of the S memory will be in level 1 memory at any given time. If all pieces of data are equally likely to be in the level 1 memory, then with probability $\frac{S_1}{S}$ we will find the data in level 1 memory, and with probability $\frac{S-S_1}{S}$ we will find the data in level 2 memory. Using this approximation, the memory access cost as a function of the size of the data structures in memory is:

$$T(S) = \begin{cases} T_1 & \text{if } S \le S_1 \\ (\frac{S_1}{S})T_1 + (\frac{S-S_1}{S})T_2 & \text{otherwise} \end{cases} \quad (1)$$

This function is shown in Figure 1b. If a datum is retrieved multiple times consecutively, then after the first retrieval, the datum will be relocated to level 1 memory, and therefore the subsequent retrieval time is T_1.

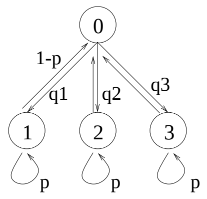


Figure 2. Markov Model for Address Prefixes

2.2 Markov Model for Packet Prefixes

During a typical TCP connection, a burst of packets are sent back-to-back along the same route to the same destination. The result is a sequence of packets with the same destination IP address, and therefore the same longest prefix that requires lookup at the router. To model this dependency, we have constructed the simple Markov model shown in Figure 2. Each prefix in a routing table is represented by a state, say state 1, 2, 3 in the figure. In addition, there is an initial state 0. Starting at the initial state, we enter state i with probability q_i. This represents a packet with longest prefix i has arrived at the router. With probability p, we return to the same state, representing the case when the next packet also has longest prefix i. With probability $1-p$, we return to initial state 0, and a new prefix is selected. The expected number of packets persisting in the same state is $1/(1-p)$.

[1]We can easily extend the model to capture memory access cost of more than two levels of hierarchical memories. The extension is trivial and thus omitted.

Here is a simple example of how the two models can be used to estimate the prefix retrieval time. Suppose that in order to decode prefix i, we need to first access datum x followed by datum y. Suppose further that the size of the data structures of the algorithm in memory is $S > S_1$. The initial prefix decoding time will be twice the quantity in (1), since we need to retrieve two pieces of data. However, on average the next $\frac{p}{1-p}$ packets will have the same longest prefix according to the prefix Markov model. The decoding time for these packets, however, is $2T_1$ each, since the data x, y have been fetched into level 1 memory [2]. The cost of this sequence of packets for prefix i is then:

$$c(i) = 2\left[(\frac{S_1}{S})T_1 + (\frac{S-S_1}{S})T_2\right] + 2(\frac{p}{1-p})(T_1) \quad (2)$$

We could in fact lump the terms together by introducing a new variable T':

$$T' = (\frac{S_1}{S})T_1 + (\frac{S-S_1}{S})T_2 + (\frac{p}{1-p})(T_1) \quad (3)$$

Now $c(i) = 2T'$. Notice the new variable T' depends only on the size of the data structures in memory, S, and is the same for all prefixes. In general, we can lump the initial prefix retrieval cost with the recurring prefix retrieval cost together by using similar introductions of new variables.

3 Problem Formulation

Using the two models described above, we now formulate the prefix decoding design problem (PDD) as a well-defined optimization problem. We first discuss a common representation of a prefix set that we will use throughout the paper, *trie*, in section 3.1. We then present the formal definitions of two classes of prefix decoding tools, *programmed logic* and *hash functions*, in section 3.2. In section 3.3, we present examples of prefix decoding algorithms, then present a formal definition of the optimization problem.

3.1 Prefix Representation

A common representation of a prefix set is a *trie* [8]. A trie is a tree-based data structure, where each node has one or more children, and edges to each one correspond to different bit sequences; in so doing, the path from the root to a particular node reveals the unique bit sequence of the node. If each node has at most two children, then the trie is a binary trie; a PATRICIA-tree [7], which has been widely used for software-based route lookups [3], is a special case of binary trie where a sequence of nodes with one child only is compressed into one node. (It is called path-compressed trie in [4].) Figure 3a shows an example of a routing table

[2] In order to avoid *thrashing*, the event when one datum pushes the other out repeatedly, a careful data layout scheme, such as memory block coloring, has to be used. We will assume such a scheme is employed.

prefix	action
0	c
000	a
001	b
10	d
11	e

a) routing table

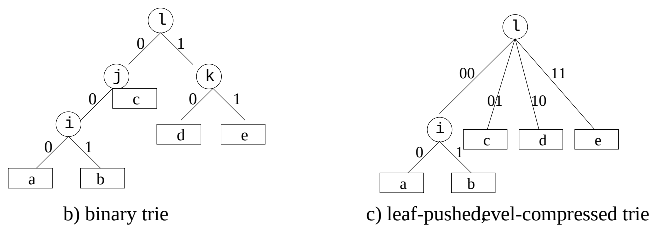


Figure 3. Representation of a Prefix Set

containing a prefix set and a set of corresponding actions. Figure 3b shows a binary trie that represents the prefix set. In its full generality, different tries can represent the same prefix set. For example, the same prefix set can be represented by a different trie in Figure 3c. In this case, prefix 0 is first expanded to prefix 01, since address with prefix 01 would indicate it has a longest prefix of 0 in the prefix set. This is called *leaf pushing* in [5]. We then eliminated nodes j and k in Figure 3b by looking at 2 bits at a time. This is called *level compression* in [4]. See [10] for a more detailed discussion of representations of a prefix set.

Notice the trie in Figure 3c looks quite different from the one in Figure 3b, yet nevertheless represents the same prefix set. In fact, we will use different trie representations of a same prefix set to represent different prefix decoding algorithms discussed in later sections.

3.2 Prefix Decoding Tools

In this section, we discuss two classes of general prefix decoding techniques: programmed logic and hash functions. Note that many common decoding techniques, such as table lookups, are members of these two classes.

3.2.1 Programmed Logic

Looking at the trie representation of the prefix set, a straight-forward decoding technique is to test the bit sequence of the input against the branch labels and follow the corresponding branch in the trie. We called this simple technique *programmed logic*, and as shown in Figure 4, is implemented using `if ...else` statements. Note that in general any number of bits can be examined at a time and in any order. To estimate the computational complexity of an `if` statement, we can assign a cost of Q_i for each test involving i contiguous bits. A natural question to ask is:

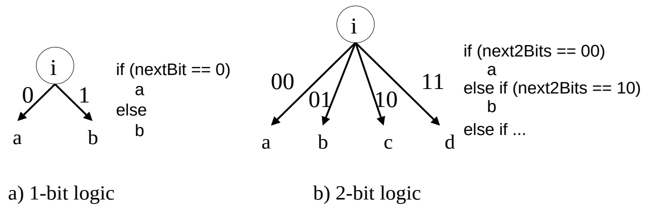


Figure 4. Sequential Programmed Logic for 1 and 2 bit Cases

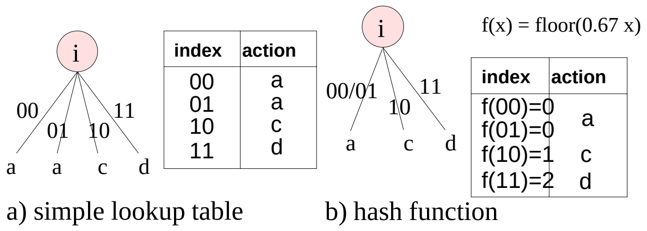


Figure 5. Two Examples of Hash Functions

given a prefix set and associated probabilities, what is the optimal logic design, for a vector of computational costs, $\mathbf{Q} = [Q_1, Q_2, \ldots, Q_l]$, where l is the length of the longest prefix? We denote this problem as the *optimal logic design problem* (LD). This is clearly a subset of the larger PDD problem, since we are restricting ourselves to logic-only algorithms.

It turns out the LD problem alone is NP-hard. We will prove this by showing the corresponding decision problem of the LD problem — Is there a logic design with cost $\leq K$? — includes the binary decision tree problem [9] as a special case. The binary decision tree problem is the following: given a set of of objects $\mathcal{X}$ of size $|\mathcal{X}| = n$ and a collection of binary tests $\mathcal{T} = \{T_1, T_2, ..., T_m\}$, each of form $T_i : \mathcal{X} \to \{0, 1\}$, does there exist a binary decision tree such that the expected number of tests required to identify an object is $\leq K$? We now state the previous NP-hard claim as a theorem and state a formal proof.

Theorem 3.1 *The LD problem is NP-hard.*

Proof 3.1 *We will show the corresponding decision problem of the LD problem is NP-complete via a reduction from the decision tree problem. We can represent the instance of the binary decision tree problem as a matrix A; for each object $x \in \mathcal{X}$, we represent the results of the test set $\mathcal{T}$ on x as a row in the matrix. A is of size $n * m$:*

$$A = \begin{bmatrix} 1 & 1 & 0 & \cdots & 0 \\ \vdots & & & & \\ 0 & 1 & 1 & \cdots & 1 \end{bmatrix} \tag{4}$$

For each object i, we construct a prefix i that corresponds to the object's row of ones and zeros. Note that a necessary condition for every object to be distinguishable from others is that no two rows are identical. As a consequence, each prefix is unique, and successful prefix decoding also means successful object identification. We let $Q_1 = 1$, and $Q_i = \infty, \forall i \geq 2$, thereby restricting ourselves to use one-bit test only. Now if we find a logic design that decodes this prefix set in cost $\leq K$, then there also is a decision tree such that cost $\leq K$. Since the decision tree problem is NP-complete, the decision LD problem is also NP-complete. Therefore, the corresponding optimal LD problem is NP-hard. □

Given the optimal LD problem alone is NP-hard, the optimal PDD problem, a generalization, is also NP-hard. The main cause of the difficulty is the fact that we can decode input bits in any order. So instead, we will restrict the search space of algorithms to a smaller subspace where prefixes must be decoded in a FIFO (first in first out) manner, i.e. contiguously from left to right. This is nevertheless reasonable, since all prefixes, no matter what length, start with the left-most bit. Many existing works [1, 3, 4, 5] also implicitly make this restriction in their search space of algorithms.

For our implementation, we further restrict the search space of programmed logic to one where programmed logic must test h bits at a time, i.e., an h-bit programmed logic at node i of a binary trie decodes the section of trie nodes rooted at i down to and including nodes of level h from i. How the logic is structured to decode this section of the trie will depend on available logic decoding techniques; examples of such techniques are optimal sequential decoding and optimal binary search. This essentially limits our set of decoding tools to a small set[3].

Formally, a set of programmed logic decoding tools is specified as follow: $\mathcal{Q} = \{G_1, \ldots, G_N, \mathbf{Q}\}$, where G_n is a particular logic technique, and $\mathbf{Q}$ is the logic cost vector as discussed previously; we will denote optimal sequential decoding as G_1. Figure 4a shows a logic design where 1 bit is decoded sequentially, and Figure 4b shows one where 2 bits are decoded sequentially. The cost of the 1-bit programmed logic is $(q_0 + q_1)Q_1$, while the cost of the 2-bit programmed logic is $[q_{00} + 2q_{01} + 3(q_{10} + q_{11})]\, Q_2$, where q_i is the probability of event i, and Q_j is the cost of testing j contiguous bits. Note that when performing ≥ 1 bit optimal sequential programmed logic, we should test the most likely event first, followed by the next most likely event and so on. So in the above cost calculation, we assume $q_{00} \geq q_{01} \geq q_{10} \geq q_{11}$.

3.2.2 Hash Functions

A hash function is defined as a many-to-few mapping rule: $f : \mathcal{X} \rightarrow \mathcal{Y}$, where $|\mathcal{X}| \geq |\mathcal{Y}|$. In particular, we are interested in a subset of all possible hash functions where $\mathcal{X} = \{0, 1, \ldots, 2^h - 1\}$ and $\mathcal{Y} = \{0, 1, \ldots, K - 1\}$; this corresponds to a mapping of h input bits to an index of an array of size K. Note that an h-bit table lookup is a special case of hash functions where $f(x) = x$ and $|\mathcal{X}| = |\mathcal{Y}| = 2^h$. Two examples of such hash functions are shown in Figure 5. The first example is a 2-bit table lookup, where $|\mathcal{X}| = |\mathcal{Y}| = 4$. In the second example, $f(x) = \lfloor 0.67(x) \rfloor$ and $|\mathcal{X}| = 4, |\mathcal{Y}| = 3$. Notice while the execution time of the first hash function is faster (since $f(x) = x$ is a direct mapping and has execution time 0), it requires more memory than the second one. In general, hash functions may also result in collision, in which case there must be mechanisms to resolve it — this again results in a speed-memory tradeoff. The design of good hash functions is a rich problem and has been studied extensively; it is not the topic of this paper.

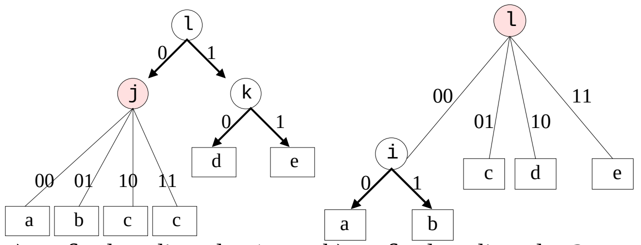
Instead, we will assume we are given a set of hash functions, $\mathcal{F} = \{f_0, f_1, \ldots, f_M\}$, where f_0 denote simple table lookup, from which we can construct a prefix decoding algorithm using a combination of them in conjunction with programmed logic decoding tools. For each hash function f_i, we must specify its associated mapping $f_i : \mathcal{X} \rightarrow \mathcal{Y}$ and its execution cost $c(f_i)$. The number of input bits for f_i and the memory required is implicitly specified in $|\mathcal{X}|$ and $|\mathcal{Y}|$. In addition, we must specify a collision cost P for each collision resolution. For table lookup, such mapping and execution cost specifications are not necessary; they are understood from the definition of an h-bit table lookup.

3.3 Examples and Problem Definition

Consider the set of prefixes in Figure 3. Using only 1-bit logic sequential decoding and simple table lookup as our prefix decoding tools, two prefix decoding algorithms are constructed in Figure 6: a) a 1-bit programmed logic is followed by either a 2-bit table lookup or another 1-bit programmed logic; b) a two-bit table lookup is followed possibly by a programmed logic. Graphically, we denote a programmed logic at a node with dark branch arrows, and hash function by shading the node. To determine the average decoding time of these algorithms, we first note that the size of data structures in memory for both algorithms is 4 (one 2-bit lookup table). We can now write the execution time of the first algorithm b_1, denoted as $H(b_1,\ T(4)\ )$, as follow:

$$H(b_1,\ T(4)\ ) \quad = \quad (q_a + q_b + q_c)\left[Q + T(4) + (\frac{p}{1-p})(Q + T_1)\right]$$

^{3}We can generalize the set of programmed logic to one that includes logic covering section of binary trie rooted at i of any shape. This will increase the computational cost of the optimization algorithm to be discussed in section 6, however.



a) prefix decoding alg. 1 b) prefix decoding alg. 2

Figure 6. Examples of Prefix Decoding using 1-bit Logic and Table Lookup

$$+(q_d + q_e)\left[2Q + (\frac{p}{1-p})2Q\right]$$
$$= \quad (q_a + q_b + q_c)(Q' + T') + (q_d + q_e)(2Q') \qquad (5)$$

where $T' = T(4) + (\frac{p}{1-p})T_1$, and $Q' = (\frac{1}{1-p})Q$. We can rewrite cost in terms of the probability flow of the internal nodes; for example, probability flow of node j is $w_j = q_a + q_b + q_c$. We can now write $H(b_1)$ as:

$$H(b_1,\ T(4)\ ) = w_l Q' + w_j T' + w_k Q' \qquad (6)$$

Similarly, the execution cost of the second algorithm, b_2 will be: $H(b_2,\ T(4)\ ) = w_l T' + w_i Q'$.

In general, to find the cost of an algorithm b, we first find the size of data structures in memory of b, denoted as $R(b)$. Then we find the memory access cost $T(R(b))$, from which we can compute cost of b, $H(b)$. We are now ready to formalize the optimal PDD problem as follow:

Prefix Decoding Design (PDD) Problem: Given:

1. Parameters of the machine model: S_1, T_1, T_2
2. A prefix set and associated parameters of the prefix Markov model: $q_i\ \ \forall i,\ p$
3. Programmed logic decoding tools $\mathcal{Q} = \{G_1, \ldots, G_N, \mathbf{Q}\}$ and hash function decoding tools $\mathcal{F} = \{f_0, f_1, \ldots, f_M\}$ and collision cost P

What is the optimal algorithm so that the average decoding time is minimized? Mathematically, we write:

$$\min_{b \in B} \{H(b,\ T(R(b))\ )\} \qquad (7)$$

where B is the set of algorithms that are combinations of available decoding tools.

Using a similar proof to the one shown in [11], one can show that even for the case of using only simple table lookup and optimal sequential logic decoding, this problem is NP-hard. We will outline such a proof in the next section, then turn our attention to an optimization framework that uses Lagrange multipliers to speed up the computation of the optimal algorithm with only a minor impact on optimality.

	x_2	x_1	x_0	y_2	y_1	y_0	z_2	z_1	z_0
a_0	1	0	0	1	0	0	1	0	0
a_1	0	0	1	0	0	1	0	0	1
a_2	0	0	1	1	0	0	0	1	0
a_3	0	1	0	0	1	0	0	1	0
K	1	1	1	1	1	1	1	1	1

Figure 7. Corresponding Partial Sum Problem of 3D Matching Problem: $N = 3$, $M = 4$

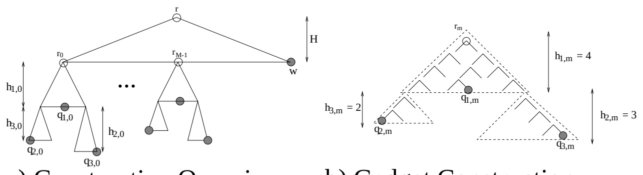
4 NP-Completeness Proof

We first rephrase the optimal PPD problem as a decision problem: given a set of prefixes with associated probabilities, does there exist an algorithm, using only simple lookup tables $\mathcal{F} = \{f_0\}$ and sequential programmed logic $\mathcal{Q} = \{G_1, \mathbf{Q}\}$, that has an average lookup cost below a target cost $\bar{C}$? We will show the decision problem is NP-complete, and as a consequence, the corresponding optimization problem is NP-hard.

4.1 3D Matching Problem

The proof is by reduction from a version of the "3D matching problem" [13]. This well-known NP-complete problem assumes the input is categorized into three distinct groups, say men, women and pets, each of size N. A list of 3-tuples of size $M > N$ specifies all possible matches of men, women and pets. For example, tuple m, (x_m, y_m, z_m), specifies man x_m, woman y_m and pet z_m is a possible match. The decision problem is: given a list of 3-tuples, is there a subset N of M matches such that each of N men, women and pets is uniquely assigned to one match.

The same problem can be reformulated as the *partial sum* problem as follow. For each tuple in the list of 3-tuples, we transform it, (x_m, y_m, z_m), to a number $a_m = (M+1)^{2N+x_m} + (M+1)^{N+y_m} + (M+1)^{z_m}$, $\forall x_m, y_m, z_m \in \{0...N-1\}$. We now have a set $\mathcal{A}$ of M numbers in numeric base $M+1$, each with $3N$ digits. Notice each number has exactly three 1's in three digit positions and the rest of the digits are zeroes. Note further that overflow in any digit position is avoided for any subset of numbers by selecting the numeric base to be $M+1$. Now, the decision problem is: does there exist a subset $\mathcal{A}' \subset \mathcal{A}$ such that $\sum_{a_m \in \mathcal{A}'} a_m = K$, where K is a number with ones in all $3N$ digit positions. An example of the partial sum version is shown in Figure 7 for $N = 3$ and $M = 4$. We see that the numbers in the subset $\mathcal{A}' = \{a_0, a_1, a_3\}$ add up to K. This version of the 3D matching problem is equivalent to the original version discussed earlier.



a) Construction Overview b) Gadget Construction

Figure 8. Proof Constructs used in the NP-complete Proof of PDD Problem

4.2 Overview of Proof

For each instance of the partial sum version of the 3D matching problem, we create a corresponding instance of the PDD problem as follow. First, we change the base of the numbers a_m's to be 2^B, where $B = \lceil \log(M+1) \rceil$. We create a full binary trie of height H rooted at node r, where $H = \lceil \log M \rceil + 1$. We first attach a "heavy" leaf w, with probability q_w, at the right bottom of the trie, as shown in Figure 8a. Then for each $a_m \in \mathcal{A}$, we construct a *sub-trie* with root r_m and attach it to the full binary trie at height H. The sub-tries are the "gadgets" necessary to map the numbers a_m's in the 3D matching problem to the PDD problem. Each sub-trie m is a concatenation of three *mini-tries* of height $h_{1,m}$, $h_{2,m}$ and $h_{3,m}$, where $h_{1,m} = 2N+x_m$, $h_{2,m} = N+y_m$, $h_{3,m} = z_m$, and has a single leaf with non-zero probability $q_{1,m}$, $q_{2,m}$ and $q_{3,m}$ respectively. Mini-trie 2 and 3 are single sided, and mini-trie 1 has three branches of the same height $h_{1,m}$, with non-zero probability leaf in the middle branch and concatenations to mini-trie 2 and 3 at the other branches. See Figure 8b for an example of a sub-trie m.

Given the construction, we first find the cost of a *default* prefix decoding algorithm, which is one that uses a H-bit lookup table at node r, then uses $(h_{1,m} + h_{2,m})$-bit sequential programmed logic of height for sub-trie m. The corresponding decision problem to the 3D matching problem is as follows: does there exist a prefix decoding algorithm that can reduce the cost by at least K, when compared to the cost of the default algorithm? Our claim is that there exists such an algorithm if and only if there also exists a subset $\mathcal{A}' \subset \mathcal{A}$ such that $\sum_{a_m \in \mathcal{A}'} a_m = K$. Such an algorithm must have the following construction: an H-bit lookup table at node r, a subset of sub-tries in *3-Δ configuration*, and the rest of the sub-tries in sequential programmed logic as the default. By 3-Δ configuration, we mean a decoding strategy for a sub-trie where it uses three lookup tables, one for each mini-trie. Note that the total size of the three lookup tables is exactly a_m. By setting the parameters $q_{1,m}$, $q_{2,m}$ and $q_{3,m}$ carefully, the lookup cost reduction at each sub-trie m of 3-Δ over default is also a_m. Moreover, any other lookup strategies at the sub-trie will either have higher cost than default,

Variable	Value
$Q_i \; \forall i$	1
T_1	0.75
T_2	∞
S_1	$2^H + K$
p	0
q_w	$1 + \sum_{i=0}^{M-1} a_i$
q_1	1γ
q_2	2γ
q_3	4γ
γ	$4a_m$

Figure 9. Parameters for Corresponding Instance of PDD Problem

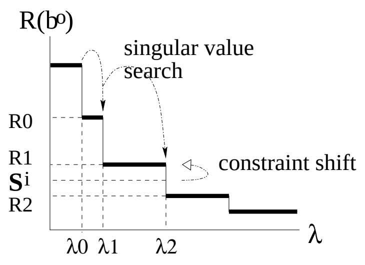


Figure 10. Size Constraint as Function of Multiplier Value

or have worse cost-reduction to size-increase ratio than 3-Δ configuration. A set of parameters that works for this claim is in Figure 9.

5 Optimization Framework

The general optimization problem (7) is a difficult problem for two reasons: i) the problem is combinatorial; ii) the problem is non-linear due to the dependence on the non-linear function $T(R(b))$. To attack this problem, we propose to first remove the non-linearity of the problem by a mathematical manipulation, then solve the simplified linear combinatorial problem in an efficient manner.

Suppose we know *a priori* the size of the data structures of the optimal algorithm b^* in memory is exactly S^*. Then the memory access cost is $T(S^*)$. To find the optimal algorithm, we need only search among all algorithms whose data structures has size exactly S^*. If we define $H_{T(S)}(b)$ as the cost function where the memory access cost is fixed at $T(S)$, then the PDD problem can be written as:

$$\min_{b \in B} \{H(b,\ T(R(b)))\} = \min_{b \in B} \left\{H_{T(S^*)}(b)\right\} \quad \text{s.t. } R(b) = S^* \tag{8}$$

Solving $H_{T(S^*)}(b)$ seems easier, since the non-linearity has been removed. The problem, of course, is that we do not know S^* a priori. So to find the optimal value, we need to search through all possible value of S:

$$\min_{b \in B} \{H(b,\ T(R(b)))\} = \min_{S_1 \le S \le S_{\max}} \left\{H'(S)\right\} \tag{9}$$

where $H'(S)$ is the optimal value of the simplified combinatorial optimization problem:

$$H'(S) = \min_{b \in B} \left\{H_{T(S)}(b)\right\} \quad \text{s.t.} \quad R(b) = S \tag{10}$$

Searching through all possible S in (9) is an enormous burden, and in the next section, we propose an efficient alternative. In the meantime, we will focus on solving (10).

5.1 Lagrange Approximation

Using a similar proof to the one in section 4, one can show that solving (10) given S is still NP-hard. So in a way, we have not simplified the problem at all. But it turns out for *some* particular values of S, (10) can be solved efficiently. To realize this, we first define the corresponding Lagrangian problem to (10) as follow:

$$\min_{b \in B} \left\{H_{T(S)}(b) + \lambda R(b)\right\} \tag{11}$$

The following theorem, similar to one in [12], relates the two problems for particular values of S.

Theorem 5.1 *Let b^o be the optimal solution to (11) for a particular multiplier λ. Suppose further that $R(b^o) = S$. Then b^o is also the optimal solution to (10).*

Proof 5.1 *By optimality of b^o to (11):*

$$\begin{aligned} H_{T(S)}(b^o) + \lambda R(b^o) &\le H_{T(S)}(b) + \lambda R(b) \qquad \forall b \in B \\ H_{T(S)}(b^o) &\le H_{T(S)}(b) - \lambda\left[R(b^o) - R(b)\right] \end{aligned} \tag{12}$$

In particular, this is also true for subset $B' \subseteq B$, where $B' = \{b \in B | R(b) = S\}$. Since $R(b^o) = S$, we get:

$$H_{T(S)}(b^o) \le H_{T(S)}(b) \quad \forall b \in B' \tag{13}$$

We can conclude that b^o is an optimal solution to (10). □

In general, solving (11) is easier than (10), since the problem is now unconstrained. The problem is that given S, there may not be multiplier value λ such that the optimal solution to (11), b^o, has the property $R(b^o) = S$. In that case, we propose the following iterative algorithm that converges to a size constraint value S and a multiplier value λ such that $R(b^o) = S$:

Iterative Projection Method:

1. Let $i := 1$. Initialize S^1.
2. Given S^i, find the multiplier value λ^i such that the optimal solution b^o to (11) minimizes $R(b^o) - S^i$ while $R(b^o) \ge S^i$, i.e. find λ that is the argument of the following:

$$\min_{-\infty \le \lambda \le \infty} \left\{R(b^o) - S^i\right\} \quad \text{s.t.} \quad R(b^o) \ge S^i \tag{14}$$

3. If $R(b^o) = S^i$. Done. Else, let $i := i + 1$, and let $S^i := R(b^o)$. Goto step 2.

An efficient algorithm for this problem, inspired by the classic result in [12], is presented in [11] where (14) is solved efficiently, and we only provide a brief outline here. The basic idea is the following: in general, $R(b^o)$ as a function of multiplier λ is a non-increasing function, since λ acts as a weight parameter in the penalty function $\lambda R(b)$ in (11). If the search space is discrete, then $R(b^o)$ is a non-increasing step function as shown in Figure 10. Observe that there is a finite set of multiplier values where the drops occur in the step function, e.g. $\{\lambda_0, \lambda_1, \lambda_2, \ldots\}$ in Figure 10. These multiplier values are called *singular values* in [12]; at each singular value, there are multiple solutions that are simultaneously optimal. By iteratively stepping through these singular values towards the size constraint S^i, say from λ_1 to λ_2 in the figure, we will eventually arrive at a drop in $R(b^o)$ where the drop spans S^i, e.g. $S^1 \in [R_1, R_2]$ in the figure. At this point, we have arrived at the closest $R(b^o)$ value to S^i, i.e. the solution to (14), which is R_1 in the figure. This procedure is called *Singular Value Search*.

5.2 Lagrangian Sampling

Instead of searching for all possible values S in (9), by finding solutions to the dual (11) using the iterative projection method, we are actually only sampling a relatively small number of points on $H'(S)$, since the method converges to a small subset of points no matter what S^1 is initialized to. We call this phenomenon *Lagrangian sampling*, since each sample point is a solution to the Lagrangian (11). By sampling, however, we may not be able to find the optimal solution $H'(S^*)$. We present the following theorem that bounds the error of a neighboring sample point from a local minimum point.

Theorem 5.2 *Let there be a locally optimal solution to (7), b^*, such that $S^* = R(b^*) \in [S^1, S^2]$, where $H'(S^1)$, $H'(S^2)$ are the two neighboring points during construction of function $H'(S)$. During singular value search for (11) for $S = S^1$, we can find an optimal solution to (11), b^B, such that one of the* **Bounding Conditions** *is satisfied:*

1. *$(R(b^B) \geq S^2)$ and $(\lambda \geq 0)$*
2. *$(R(b^B) \leq S^1)$ and $(\lambda < 0)$*
3. *$\lambda = 0$*

The locally optimal solution b^* is lower bounded by b^B, i.e.: $H'(S^*) = H_{T(S^*)}(b^*) \geq H_{T(S^1)}(b^B)$.

We can now locally bound the error of a neighboring pair of sample points, as shown in Figure 11. The global error bound is the difference between the best performance sample point and the best performance bound of all pairs.

Proof 5.2 *We first note that at least one of the following three cases must be true: i)* **case I***: vertical drop* [4] *of $R(b^o)$ at S^2 and horizontal plane of S^1 both occur for $\lambda \geq 0$, ii)* **case II***: drop of $R(b^o)$ at S^2 and plane at S^1 both occur for $\lambda < 0$, iii)* **case III***: drop of $R(b^o)$ at S^2 occurs at $\lambda \leq 0$, a portion of plane of S^1 occurs at $\lambda \geq 0$. See Figure 12 for an illustration. It is clear that for each case, there exists an optimal solution b^B to (11) such that one of the bounding conditions is satisfied. We now prove that if one of the conditions is satisfied, then the error bound holds.*

[4]Same argument works if it is a plane instead of a drop at S^2.

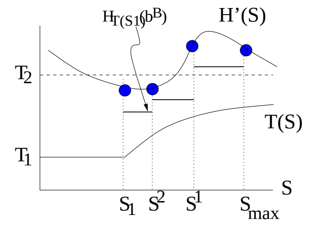


Figure 11. Lookup Cost vs. Memory Usage

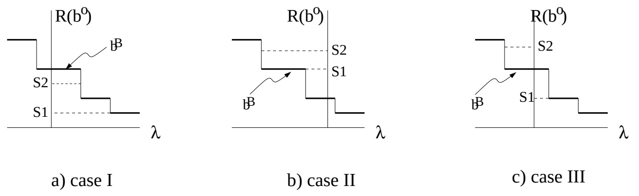


Figure 12. Three Cases for Theorem 2

We first define two related optimization problems to (10) by constraint relaxations:

$$H'_{\leq}(S^*) = \min_{b \in B} \left\{ H_{T(S^*)}(b) \right\} \quad s.t. \;\; R(b) \leq S^* \tag{15}$$

$$H'_{\geq}(S^*) = \min_{b \in B} \left\{ H_{T(S^*)}(b) \right\} \quad s.t. \;\; R(b) \geq S^* \tag{16}$$

Let $b^*_{\leq}$ and $b^*_{\geq}$ be the optimal solutions to (15) and (16) respectively. Since the feasible spaces for both problems are both supersets of (10), it is clear that $H'_{\leq}(S^*) \leq H'(S^*)$ and $H'_{\geq}(S^*) \leq H'(S^*)$. We now prove each of the three cases separately:

Case I*: Given optimal solution b^B to (11) for $\lambda \geq 0$, and $R(b^B) \geq S^2$. By optimality:*

$$H_{T(S^1)}(b^B) + \lambda R(b^B) \leq \quad H_{T(S^1)}(b) + \lambda R(b) \quad \forall b \in B \tag{17}$$

$$\lambda \left[R(b^B) - R(b) \right] \leq \quad H_{T(S^1)}(b) - H_{T(S^1)}(b^B) \tag{18}$$

If we let $b = b^*_{\leq}$, given $\lambda \geq 0$ and $R(b^*_{\leq}) \leq S^* \leq S^2$, term on left is non-negative. Hence:

$$\begin{aligned} H_{T(S^1)}(b^B) &\leq H_{T(S^1)}(b^*_{\leq}) \leq H_{T(S^*)}(b^*_{\leq}) && (19) \\ &\leq H_{T(S^*)}(b^*) = H'(S^*) && (20) \end{aligned}$$

where the second inequality of (19) is true because $S^1 \leq S^*$ implies $T(S^1) \leq T(S^*)$, and therefore $H_{T(S^1)}(b) \leq H_{T(S^*)}(b) \;\; \forall b \in B$. Therefore, error bound holds for case I.

Case II*: Given optimal solution b^B to (11) for $\lambda < 0$, and $R(b^B) \leq S^1$. Following the same optimality argument, we again get (18). For $b = b^*_{\geq}$, we can again argue the left term is non-negative, since the two products are strictly negative and non-positive respectively. Hence:*

$$H_{T(S^1)}(b^B) \leq H_{T(S^1)}(b^*_{\geq}) \leq H_{T(S^*)}(b^*_{\geq}) \tag{21}$$

$$\leq \quad H_{T(S^*)}(b^*) = H'(S^*) \tag{22}$$

Therefore, error bound holds for case II.

Case III*: Given b^B is optimal solution to (11) for $\lambda = 0$. Following the optimality argument, we again get (18). Now with $\lambda = 0$, we get:*

$$H_{T(S^1)}(b^B) \leq H_{T(S^1)}(b) \quad \forall b \in B \tag{23}$$

This is also true for the locally optimal solution b^*. Hence:

$$\begin{aligned} H_{T(S^1)}(b^B) &\leq H_{T(S^1)}(b^*) \\ &\leq H_{T(S^*)}(b^*) = H'(S^*) \end{aligned} \tag{24}$$

Therefore, bound holds for case III. We have proven all cases, and so the theorem is proven. □

6 Implementations and Results

6.1 Implementation

We will now discuss specific implementation issues. In particular, we will discuss how (11) is solved, i.e. given λ, how to find: $\min_{b \in B} \left\{ H_{T(S)} b + \lambda R(b) \right\}$. We start with a binary trie representation of the prefix set with root node r. Let $f_\lambda(i)$ be the minimum cost function that returns the minimum Lagrangian cost for a binary trie rooted at node i given λ. At each node i, we have to decide which decoding tool to use. Suppose we decide to use an h-bit programmed logic technique $G_n \in \mathcal{Q}$ at node i, i.e. programmed logic for the section of binary trie rooted at i down to nodes of height h from i. We will denote this logic cost as $G_n(i,h)$. Then the cost is the sum of $G_n(i,h)$ and the recursive children costs. Suppose we decide to use a hash function $f_m \in \mathcal{F}$, with h_m the number of input bits associated with f_m, and $\mathcal{Y}_m$ the output mapping of f_m. Then the cost is sum of the hash function cost $c(f_m)$, the memory access cost $T(S)$, the penalty term $\lambda R(b) = \lambda|\mathcal{Y}_m|$, and recursive children costs. $f_\lambda(i)$ is the minimum among all these choices. Assuming no collisions, we can write:

$$\begin{aligned} f_\lambda(i) &= \min \left\{ f_\lambda^{\mathcal{Q}}(i), f_\lambda^{\mathcal{F}}(i) \right\} \\ f_\lambda^{\mathcal{Q}}(i) &= \min_{1 \leq h \leq H_i} \left[G_n(i,h) + \textstyle\sum_{j \in L_{h,i}} f_\lambda(j) \right] \quad \forall G_n \in \mathcal{Q} \\ f_\lambda^{\mathcal{F}}(i) &= w_i \left[c(f_m) + T(S) \right] + \lambda |\mathcal{Y}_m| \\ &\quad + \textstyle\sum_{j \in L_{h,i}} f_\lambda(j) \qquad \forall f_m \in \mathcal{F} \end{aligned} \tag{25}$$

where H_i is the height of the binary trie rooted at i, and $L_{h,i}$ is the set of nodes at level h of trie rooted at i.

To look at a more concrete case, we consider the case where we only have a one-bit sequential programmed logic and simple table lookup as prefix decoding tools. This is an abbreviated discussion to the one presented in [11]. $f_\lambda(i)$ is the minimum cost between the two choices: i) using logic at node i, ii) using a lookup table at node i. Of course, the lookup table can be of any height h up to H_i. For each height of the lookup table, there will be a penalty $\lambda 2^h$, that

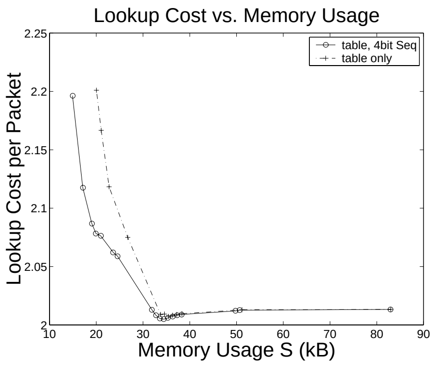


Figure 13. Ave. Cost as Function of Memory

corresponds to $\lambda R(b)$ in (11). Mathematically, we summarize the above analysis as follow:

$$f_\lambda(i) = \min \left\{ \begin{array}{l} w_i Q + \sum_{j \in L_{1,i}} f_\lambda(j) \\ \min_{1 \leq h \leq H_i} \left[w_i T(S) + \lambda 2^h + \sum_{j \in L_{h,i}} f_\lambda(j) \right] \end{array} \right. \tag{26}$$

Because there are overlapping subproblems, each time $f_\lambda(i)$ is solved, the value is copied to a dynamic programming table $F_\lambda[i]$ of size $N * 1$, where N is the number of nodes in the binary trie. Next time $f_\lambda(i)$ is called, algorithm simply returns the value in $F_\lambda[i]$; this way, a subproblem is solved only once. See [11] for more details.

6.2 Results

To demonstrate the efficacy of our optimization framework, we obtained a routing table named PAIX from [14] on 6/19/98; it has 2638 prefixes. We were unable to obtain the statistics of these prefixes, however, and we will use two prefix probability distributions for our simulation: i) *Equal*, where all prefixes are equally probable; and, ii) *Scaled*, where an h-bit prefix is twice as likely as an $h+1$-bit prefix (so longer prefixes are less likely than shorter ones). We additionally assume the recurring probability p in the Markov model is 0.67, resulting in 3 consecutive packets on average with the same longest matched prefix before another prefix is randomly selected again. The implementation platform is a Pentium II 266 MHz, with L1 cache 16kBytes and L2 cache 512kBytes. The machine model parameters we use are $(T_1, T_2, Q) = (2, 4, 3)$.

Recall that to find the optimal algorithm, we first construct the Lagrangian sampled function $H'(S)$ and find the minimum sample point empirically. We will construct two such functions, each has a different collection of decoding tools available. For the first function, we use only one decoding tool — simple table lookup up to any height. This is the top curve in Figure 13. For the second function, we use an additional decoding tool — optimal sequential programmed logic of up to height 4. We see that the first func-

Srinivasan & Varghese [5]	3.902 mil. lookups/s
Cheung & McCanne 98 [10]	4.546 mil. lookups/s
Lookup Tables only	4.938 mil. lookups/s
Tables + 4bit Seq. Logic	5.000 mil. lookups/s

Figure 14. Results for Equal Prob. Prefixes

Srinivasan & Varghese [5]	3.961 mil. lookups/s
Cheung & McCanne 98 [10]	7.143 mil. lookups/s
Tables + 4bit Seq. Logic	7.273 mil. lookups/s

Figure 15. Results for Scaled Prob. Prefixes

tion is above the second one for all memory S; this agrees with our intuition since the search space of the second function includes the search space of the first one.

To test the generated algorithms using our proposed optimization framework, we compare the performance of our algorithms against two other algorithms in the literature: i) a lookup table design algorithm known as *controlled prefix expansion* presented in [5] which minimizes worst case instead of average case, and ii) an optimization algorithm that minimizes the average case using a static memory model in [10]. Using the PAIX prefix set and the two probability distributions as previously discussed, we simulated 10 million IP addresses using the Markov model discussed in section 2. For each algorithm, we decoded the addresses 40 times to find the average decoding speed. In Figure 14, we see our proposed algorithm outperforms Srinivasan & Varghese by 28.2%, and Cheung & McCanne 98 by 10.0%. Note again that the optimal algorithm of the second function, "Tables + 4bit Seq. Logic", is faster than the optimal algorithm of the first function.

For prefixes with the scaled probability distribution, the optimal algorithm of the second function outperforms Srinivasan & Varghese by 83.6%, and Cheung & McCanne 98 by 1.82%. The reason for the dramatic improvement over Srinivasan & Varghese is that since the statistics are very skewed, the worst-case optimal solution is far from the average-case optimal solution.

7 Conclusion

In this paper, we formalized the IP address lookup algorithm as an optimization problem, where optimality is defined with respect to a machine model and a prefix Markov model. We have shown the problem is NP-hard, and presented an optimization framework and an associated approximate algorithm using Lagrange multipliers. We have shown that our proposed algorithm has noticeable improvement over existing algorithms in the literature. Finally, we note that although the framework and Lagrangian approximation approach is presented in the context of IP address lookup, it can potentially be used for other computational optimization problems that involve tradeoffs between decoding steps and memory usage, such as packet classification.

References

[1] M.Degermark, A.brodnik, S.Carlsson, S.Pink, "Small Forwarding Tables for Fast Routing Lookups," SIGCOMM 97, pp.3-13, 1997.

[2] M.Waldvogel, G.Varghese, J.Turner, B.Platter, "Scalable High Speed IP Routing Lookups," SIGCOMM 97, 1997.

[3] Keith Sklower, "A Tree-based Routing Table for Berkeley UNIX". Technical Report, University of California, Berkeley.

[4] S.Nilsson and G. Karlsson, "Fast Address Lookup for Internet Routers," *International Conference on Broadband Communication"*, April 1998.

[5] V.Srinivasan and G.Varghese, "Faster IP Lookups using Controlled Prefix Expansion," ACM Sigmetrics 98.

[6] A.Begel, S.McCanne, S.Graham, "BPF+: Exploiting Global Data-flow Optimization in a Generalized Packet Filter Architecture," *to appear in* SIGCOMM 99.

[7] D.Morrison, "PATRICIA- Practical Algorithm To Retrieve Information Coded in Alphanumeric," *Journal of the ACM*, vol.15, No.4, pp.514-534, October 1968.

[8] E.Fredkin, "Trie memory," *Communications of the ACM*, 3:490-500, 1960.

[9] L.Hyafil and R.Rivest, "constructing Optimal Binary Decision Trees is NP-Complete," Information Processing Letters, vol.5, no.1, May 1976.

[10] G.Cheung and S.McCanne, "Optimal Routing Table Design for IP Address Lookups Under Memory Constraints," INFOCOM 99, 1999.

[11] G.Cheung, S.McCanne, C.Papadimitriou, "Software Synthesis of Variable-length Code Decoder using a Mixture of Programmed Logic and Table Lookups," DCC 99, pp.121-130, 1999.

[12] Y.Shoham and A.Gersho, "Efficient Bit Allocation for an Arbitrary Set of Quantizers," *IEEE Trans. ASSP*, vol.36, pp.1445-1453, September 1988.

[13] Garey and Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, pp.247, 1979.

[14] *http://www.merit.edu/ipma*