EE384y Project Report

Katerina Argyraki, Jim Washburn

June 6, 2003

Contents

1 Problem Statement **2**

2 Approach we are taking **2**

3 Previous work **2**

4 Background **2**
- 4.1 Single Rotation . . . 3
- 4.2 Move to Root . . . 3

5 Structures **3**
- 5.1 Base Structure: Binary Search Tree . . . 3
- 5.2 Single-rotation Randomized . . . 4
- 5.3 Move-to-root Randomized . . . 4
- 5.4 Move-up with Counters . . . 5
- 5.5 Move-to-root Bounded . . . 5

6 Markov Chain Analysis **5**
- 6.1 Results . . . 6

7 Results **7**
- 7.1 Performance Metrics . . . 7
- 7.2 Some Simulation Results . . . 8
  - 7.2.1 Scenario 1 . . . 8
  - 7.2.2 Scenario 2 . . . 8
  - 7.2.3 Scenario 3 . . . 9
- 7.3 Interpreting the Results . . . 9
  - 7.3.1 Being conservative is bad . . . 10
  - 7.3.2 Shake the tree . . . 10
  - 7.3.3 but don't overdo it . . . 11
  - 7.3.4 What's the perfect probability? . . . 11
  - 7.3.5 Counters can nail you . . . 12
  - 7.3.6 Is worst-case performance really bad? . . . 12
- 7.4 Bounding the Tree Depth . . . 13
  - 7.4.1 Scenario 1 . . . 13
  - 7.4.2 Scenario 2 . . . 13
  - 7.4.3 Does it cost? . . . 13

8 Conclusions **13**

1 Problem Statement

This project is in the area of network lookup algorithms, more specifically, in the area of optimizing lookup times by taking into account the relative frequency of access of the keys. This project differs from previous work in that it does not assume that the relative frequencies of access are known a priori. It would be worthwhile to remove this assumption for the following reasons:

1. In a networking environment, where we are doing lookups on IP addresses or MPLS tags for example, such relative frequency information does not typically exist, or at least is not knowable to software which builds the data structures to be searched . The basic reason it does not exist is that the keys are typically added to the table incrementally and continuously, interspersed in time with table searches. In other words we do not have a model in which a set of keys are given and compiled using relative access frequency information, and searches happen after that point. Requiring a "compile" operation in addition to the basic set of add, delete, and search introduces the issues of when to compile, and what happens to searches during the compile operation.

2. Maintaining access frequency information (in the form of a counter per item stored in the table perhaps) adds to access time and memory consumption. This information is sometimes present anyway for accounting reasons but not in all cases. If we had such information, on what basis would we "age" the information so that we can respond to changing access probability distributions?

2 Approach we are taking

We try to minimize the expected value of the access time to a data structure, by continuously and incrementally readjusting it. In contrast to the usual approach, we'll do this at the time of access to the data structure (lookup time). As our starting point, we use in a binary search tree on intervals, as a lookup data structure for a search with a single field as a key. When we match a key, with some probability p, we will move the the node which matches the key closer to the root of the BST.

3 Previous work

There is previous work in optimizing routing tables by making use of relative access frequency information [1] [2]. Like this previous work we will assume that frequency of accesses to the keys is not uniform, and in fact attempt to take advantage of this fact to improve lookup performance. The papers written in recent years proposing IP lookup algorithms have the paradigm of compiling an optimized data structure once and for all, then doing searches on it henceforth. The reason for the optimization/search dichotomy is probably the underlying assumption that optimization is very complex (so should be done in software) and searches must be simple and fast (so should be done in specialized hardware.) However, in the context of network processors this hardware/software split increasingly does not necessarily apply. That is, in many systems there is only one processor (or array of identical processors) performing both search and table building operations. It's possible to make the randomized, amortized optimization algorithm fast to execute, and in any case only executed with $p \ll 1$ per search.

Looking in the computer science literature, we find that the idea of adjusting lookup structures at search time has previous work: [3] [4] [5] [6] However the idea of making use of this approach in a networking context has not been suggested before as far as we know.

4 Background

Although "self-adjustment" can potentially be done on many types of search data structures, we focused on binary search trees (BSTs). The reason is that this data structure allows us to perform a localized modification to the tree to affect the access time of one node, without having to redraw the entire tree, and leaving the rest of tree relatively unaffected. In this section, we first introduce some BST terminology and then we briefly describe two "self-adjusting" BSTs, which we later use as a basis to build our routing table structures.

Consider a BST T with n records $\{R_1, R_2, ...R_n\}$, where record R_i is requested with probability p_i, so that $\Sigma_{i=1}^{n} p_i = 1$. The cost c_i of retrieving record R_i, is defined as the number of comparisons used in performing a search for R_i in T. The cost $C_T(p_1, p_2, ...p_n)$ of the tree is defined as the average cost of retrieving a record i.e., $C_T(p_1, p_2, ...p_n) = \Sigma_{i=1}^{n}(p_i \cdot c_i)$.

Now consider the n records $\{R_1, R_2, ...R_n\}$. If the set of probabilities $\{p_1, p_2, ...p_n\}$ is *given*, it is well known how to arrange the n records in an *optimal BST* i.e., a BST of minimum cost $C_{OPT}(p_1, p_2, ...p_n)$ [7] [1].

A *self-adjusting* BST deals with the case where the set of probabilities $\{p_1, p_2, ...p_n\}$, is not given. Thus, on each access, it adjusts itself according to a heuristic, in an effort to approximate the optimal BST. The cost of a self-adjusting BST is defined as the asymptotic average cost of retrieving a record.

4.1 Single Rotation

In [3], Allen and Munro describe the following heuristic: *Every time a record is accessed, (i) if the record resides in the root, do nothing, (ii) otherwise, perform a rotation* [2] *, such that the node in which it resides is moved closer to the root.* The cost of the resulting structure is neither given in closed form nor bounded. It is only computed for the specific case in which all records are accessed with the same probability and shown to be far from the minimum cost i.e., the cost of the corresponding optimal BST [3].

4.2 Move to Root

A more aggressive policy, also presented in [3], is the following: *Every time a record is accessed, repeat the single rotation heuristic until the node in which it resides becomes the root.* The cost of the resulting structure is bounded with the following relation: $C_{MR} < (2ln2)(C_{OPT} + logC_{OPT}) + 3$.

5 Structures

5.1 Base Structure: Binary Search Tree

Our first goal was to design a routing table, in which the routing rules would be stored in a BST. In [10] Lampson *et al.* propose the following solution: Each prefix in the routing table is expressed as an interval on the number line $[0, 2^{32})$. Then, the end-points of the resulting intervals are sorted in increasing order and the duplicates are eliminated. Now consider all the disjoint intervals starting at an end-point and ending at the next. For each of these "basic" intervals, the appropriate routing rule is precomputed. Each precomputed rule is stored in a BST leaf, the key of the node being the start of the corresponding "basic" interval. The internal BST nodes contain keys for guiding the search appropriately.

This solution did not match our needs for the following reason: It always stores routing rules in BST leafs, never in internal nodes. As a result, a leaf cannot be rotated up towards the root. We can always rotate its parent up towards the root, but that does not guarantee that the leaf will move towards the root as well.

Thus, we used a slightly different solution: Each precomputed rule is stored in a BST node (not necessarily a leaf), the key of the node being the corresponding "basic" interval. Note that a key is not a single end-point any more, it is a "basic" interval. Thus, a key is now composed of a *start* and an *end*. To look up an address a in our structure, we start at the root and search down the BST until we find the "basic" interval where a belongs. At each node we traverse,

- if a is smaller than the current *start*, we go left;
- if a is greater than the current *end*, we go right;

[1] Note that $C_{OPT}(p_1, p_2, ...p_n)$ is not known in closed form.

[2] For a description of BST rotations, see [8].

[3] $C_{SR}(\frac{1}{n}, \frac{1}{n}, ...\frac{1}{n}) \approx \sqrt{\pi n} \gg C_{OPT}(\frac{1}{n}, \frac{1}{n}, ...\frac{1}{n}) \approx logn$

- otherwise, a belongs to the current interval

An example of a routing table with 4-bit prefixes is shown in figure 1. A corresponding BST is shown in figure 2.

	Prefix	StartPoint	EndPoint
P_1	*	0000	1111
P_2	0*	0000	0111
P_3	00*	0000	0011
P_4	001*	0010	0011
P_5	1001	1001	1001

Figure 1: Routing Table with 4-bit prefixes

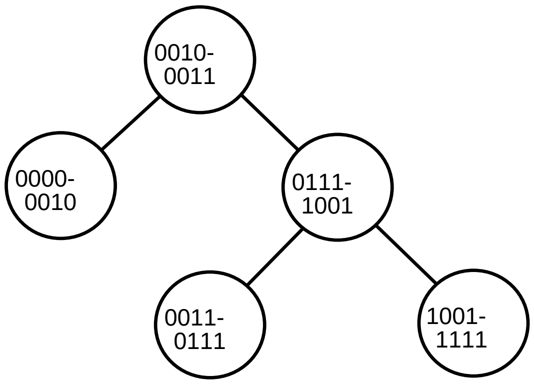

Figure 2: Binary Search Tree on Intervals

The drawback of this structure, compared to the BST suggested by Lampson *et al.*, is that each node stores *two* key components, the interval start and the interval end. This increases the amount of data that needs to be read per node. On the other hand, it enables incremental adjustments to the BST through rotations. As we show later, the benefit of incrementally adjusting the BST is very significant and, thus, outweighs the disadvantage of increasing node size.

Starting from this basic structure, we built a series of self-adjusting structures. We started with implementing single-rotation and move-to-root and evolved them based on insights provided by our simulation results. In the following subsections, we present only three of them – the ones that yield the most interesting results.

5.2 Single-rotation Randomized

First, we built SR (single-rotation): A routing table structure, which applies the single-rotation heuristic on every lookup. Each single rotation affects two nodes (the one being rotated up and its parent). Thus, the SR structure requires 2 write accesses per lookup, unless the accessed node is already at the root.

Doing 2 write accesses per lookup seems like a lot of work. Therefore, we built SRR (single-rotation randomized): A routing table structure, which applies the single-rotation heuristic with some small probability p. I.e., on every lookup, we flip a coin with bias p. If the result is heads, we do the single-rotation heuristic, otherwise, we don't.

The intuition behind SRR is simple: The goal of SR is to approximate the optimal BST by eventually moving frequently accessed nodes up near the root. Hopefully, even if we do not rotate on every lookup, frequently accessed nodes will still be moved up near the root – it is just that it will take longer. If this turns out to be true, then we get most of the SR benefit without all its cost.

5.3 Move-to-root Randomized

We also built MTR (move-to-root): A routing table structure, which applies the move-to-root heuristic on every lookup. In order to move a node to the root, we have to rotate it up against all its ancestors. Thus, the MTR structure requires as many write accesses as read accesses per lookup, unless the accessed node is already at the root.

If doing 2 write accesses per lookup seems like a lot of work, doing as many write as read accesses seems unacceptable. Therefore, we built MTRR (move-to-root randomized): A routing table structure, which applies the move-to-root heuristic with a small probability. We tried both using a fixed probability p and a probability proportional to the depth of the accessed node – which favors nodes that are deep down the tree.

The intuition behind MTRR is pretty much the same: Maybe we can get most of the MTR benefit, without the cost of moving a node all the way to the root on every lookup. The intuition behind favoring nodes deep down the tree is the following: If a node is close to the root, it has good access time anyway and so, moving it even higher does not worth its cost. Of course, one might argue that, in this way, we also favor infrequently accessed nodes, which *should* be deep down the tree. As we will see in the results section (and will try to explain), this does not turn up to be a problem.

5.4 Move-up with Counters

Then, we tried a "middle ground" solution between SR and MTR. MUP (move-up) is a deterministic routing table structure, which applies the move-to-root heuristic partially. A hit counter is kept per node. Whenever a node is accessed, it is moved up towards the root, until it meets another node with a higher hit counter. A node is never rotated above another node with a higher hit counter.

MUP attempts to strike a balance between the "conservativeness" of SR and the "aggressiveness" of MTR. The role of hit counters is to prevent unnecessary node oscillations. E.g., consider a very rarely accessed node at the bottom of the tree. Under SR or MTR, whenever this node is accessed, it is rotated up, either once or to the root. Because it is accessed very rarely, it is quickly rotated back towards the bottom. As a result, the next time the node is accessed, it is still at the bottom of the tree. This means that we paid the cost to rotate it up for nothing.

5.5 Move-to-root Bounded

Finally, noting that all our previous structures had bad worst-case performance, we built MTRB (move-to-root bounded): A routing table structure, which applies the move-to-root heuristic, but only if doing so will not increase tree depth beyond some maximum.

Bounding tree depth efficiently is not as trivial as it sounds: In order to do it, we have to somehow know what the current depth of the tree is and also be able to tell whether applying some adjustment will increase it. It seems that the only way to do that is by keeping some extra information per node e.g., the size of the longest branch starting from that node. The problem is that, maintaining this extra information requires extra updates.

E.g., consider the single-rotation heuristic. Suppose each node knows the size of its longest branch. Now suppose we access node x and we rotate it up. By doing that, we may have increased the size of x's longest branch. If this is true, we may have also increased the size of all of x's ancestors' longest branches, which means that we have to update all of them. Hence, single-rotation does not require only 2 node updates any more.

Fortunately, when we apply the move-to-root heuristic, we update all the ancestors of the accessed node anyway. Thus, maintaining the extra information does not require extra accesses. So, here is how it works: Each node stores two extra variables:

- The size of the longest branch on the left.
- The size of the longest branch on the right.

As illustrated in Figures 3 and 4, if we rotate node x up against its parent y, by accessing only x and y we have enough information to update both of them. Obviously, if we rotate x all the way up to the root, at every step, we also have enough information to update the 2 nodes involved.

6 Markov Chain Analysis

(Jim)

An observation in [3], that the different tree configurations for a given set of keys forms a Markov chain, gives us to a way to analyze the self-modifying heuristics. The number of possible BST configurations is the Catalan number of the number of nodes in the tree. The Catalan numbers are $\{1, 2, 5, 14, 42, 132, 429, ...\}$

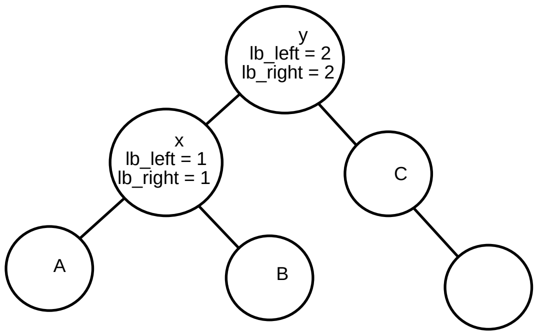

Figure 3: MRTB before rotation

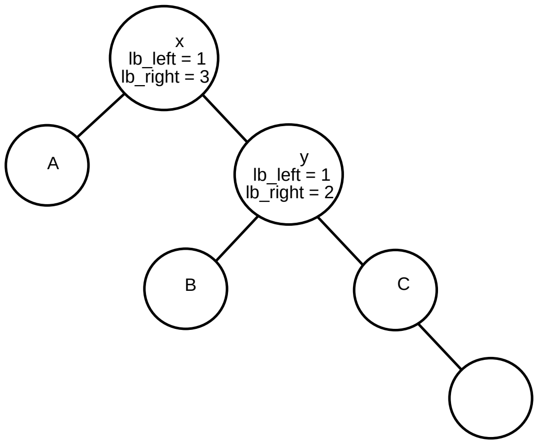

Figure 4: MRTB after rotation

or $\begin{pmatrix} 2n \\ n \end{pmatrix}/(n+1)$. So for 3 nodes , the number of tree configurations is 5. A 5x5 matrix for the Markov chain is certainly practical to solve. For each heuristic there is a particular transition matrix. For example, given a particular assignment of tree configurations to states, the matrix for single rotation looks like:

$$\begin{pmatrix} p_3 & p_1 & p_2 & 0 & 0 \\ p_2 & p_3 & 0 & p_1 & 0 \\ p_3 & 0 & p_2 & 0 & p_1 \\ 0 & p_3 & 0 & p_1 & p_2 \\ 0 & 0 & p_2 & p_3 & p_1 \end{pmatrix}$$

Where the p_i are the probabilities of accessing key K_i.

Given a particular set of p_i we have calculated various quantities of interest.

1. We calculated the stationary distribution, and using that have found the expected value of number of nodes traversed per search, for each heuristic, and compared it to the optimal tree configuration.

2. We are interested in gaining some insight into convergence to optimal configurations for the different heuristics. There is a technique in stochastic processes called the method of absorption [12]. Using this it is possible to calculate the expected value of the number of steps taken before a given state in the Markov chain is entered, given that the chain could start in any state. This can be thought as a generalization of the Gambler's Ruin problem. Using this technique we have calculated the expected number of accesses until the first passage to the optimal tree configuration for a given access distribution. This is a way to analyze convergence to optimal configurations.

3. We have calculated the entropy of the stationary distribution[11]. This gives us an idea of how much the tree configuration changes dynamically.

   Code was written in MATLAB to perform the above.

6.1 Results

60 probability distributions were randomly generated and used as input for the calculations. The following numbers are averages over all the distributions.

1. Expected value for number of nodes traversed during search

Single Rotation	Move to Root	Optimal
1.4352	1.4243	1.3018

We did not actually see a significant difference here in expected access time between the two basic heuristics, single rotation and move to root, for a 3 node tree. Most likely this is because the result depends on the stationary distribution, which implies that infinite time has passed and thus both single-rotation and move-to-root should converge toward a similar result.

Changing the probability of performing the optimization to a low value (emulating heuristics SRR and MTRR) had no effect on these results, as expected because the results depend on the stationary distribution.

2. Expected value for number of accesses until first passage to optimal tree configuration

Single Rotation	Move to Root
31.0081	20.6836

This indicates that move-to-root converges faster toward the optimal distribution. This is understandable because move-to-root makes more changes to the tree per access.

Here also we experimented with setting the coin flip for performing the optimization to different values. We found that in both cases the first passage time increased, inversely proportion to the coin flip bias. For example if we changed the probability of performing the optimization from 1.0 to 0.1, the first passage time increased by a factor of 10x. This is perhaps expected in a linear system but it is good to confirm it.

3. Entropy of stationary distribution

Single Rotation	Move to Root
1.2513	0.9113

Move to root has lower entropy for its stationary distribution. Looking at the stationary distributions as they are calculated, we observe that the distributions for move to root typically have more than one state with zero probability, whereas single rotation tends to spend at least some time in the least optimal states (tree configurations). This is probably due to the fact that successive single rotations can cancel each other out and leave no net effect on the tree, whereas this does not happen with move-to-root.

7 Results

(Katerina)

7.1 Performance Metrics

Today, the factor that affects most dramatically the performance of a classification algorithm is the number of memory accesses. This number depends on three factors:

- The number of nodes accessed until a matching rule is found.
- The amount of data that needs to be read/written from/to each node.
- The memory bus width.

In our BST structures, each node stores at least 160 bits of data – $32 + 32$ bits for the interval start and end, $32 + 32$ bits for the left and right pointers and 32 bits for the routing rule. The MUP structure stores 32 more bits per node for the hit counter. The MTRB structure stores 64 mire bits per node for the left and right longest branches' sizes. In contrast, a Trie node stores only 96 bits of data – $32 + 32$ bits for the left and right pointer and 32 bits for the routing rule. Thus, a small memory bus width – e.g. 32 bits - would favor the Trie over our BST structures. At this point, we have not decided on a bus width yet. For now, we use only the number of nodes accessed as a performance metric.

7.2 Some Simulation Results

We evaluated the performance of our structures, using the PALAC simulator [9]. From our testing scenarios, we present 3 representative ones. All results refer to routing tables of 256 rules, 250,000 lookups per scenario. SRR applies the single-rotation heuristic with probability $p = 0.1$. MTRR applies the move-to-root heuristic with probability $p = node\_depth/maximum\_depth$. For each scenario, we show:

- Average number of reads, writes and total accesses.
- Worst-case reads, writes and total accesses.
- Worst per node average reads, writes and total accesses. I.e., we choose the node with the worst average reads, the node with the worst average writes and the node with the worst average total and show the corresponding numbers. In a static structure, these numbers are the same with the overall worst-case numbers. However, in our dynamic structures, they can be pretty different.

7.2.1 Scenario 1

- 99% of hits are concentrated on 7 rules. The rest 1% is equally spread on the remaining rules.
- Input distribution: $p_1 = 0.5$, $p_2 = 0.25$, $p_3 = 0.12$, $p_4 = 0.06$, $p_5 = 0.03$, $p_6 = 0.02$, $p_7 = 0.01$, $p_i = 0.01/249, i > 7$.
- Average performance:

	OPT	SR	MTR	SRR	MTRR	MUP
Average Reads	4.56	9.21	4.82	7.51	4.95	5.7
Average Writes	0	1.13	4.55	0.11	0.15	0.16
Average Total	4.56	10.35	9.38	7.62	5.1	5.87

- Worst-case performance:

	OPT	SRR	MTRR	MUP
Worst Reads	12	57	21	31
Worst Writes	0	2	21	14
Worst Total	12	57	42	35

- Worst per node average performance:

	OPT	SRR	MTRR	MUP
Worst Average Reads	12	38.72	11.49	21.12
Worst Average Writes	0	0.3	0.68	1.35
Worst Average Total	12	40.72	11.81	21.5

7.2.2 Scenario 2

- 60% of hits are concentrated on 12 rules. The rest 40% is equally spread on the remaining rules.
- Input distribution: $p_1 = 0.13$, $p_2 = 0.09$, $p_3 = 0.08$, $p_4 = 0.07$, $p_5 = 0.06$, $p_6 = 0.05$, $p_7 = 0.04$, $p_8 = 0.03$, $p_9 = 0.02$, $p_10 = 0.01$, $p_11 = 0.01$, $p_12 = 0.01$, $p_i = 0.4/244, i > 12$.
- Average performance:

	OPT	SRR	MTRR	MUP
Average Reads	6.79	14.76	7.34	7.37
Average Writes	0	0.18	0.26	0.23
Average Total	6.79	14.94	7.6	7.6

- Worst-case performance:

	OPT	SRR	MTRR	MUP
Worst Reads	10	72	22	25
Worst Writes	0	2	21	14
Worst Total	10	74	42	29

- Worst per node average performance:

	OPT	SRR	MTRR	MUP
Worst Average Reads	10	49.45	10.64	16
Worst Average Writes	0	0.26	0.63	0.96
Worst Average Total	10	51.45	11	16.15

7.2.3 Scenario 3

- All rules are equiprobable.
- Input distribution: $p_i = 1/256, \forall i$.
- This scenario is interesting, because it represents the worst case for self-adjusting heuristics. Given that we always start from a randomly built BST, when all rules are accessed with the same probability, the best we can do is simply nothing: The randomly built tree is as close to optimal as we can get.
- Average performance:

	OPT	SRR	MTRR	MUP
Average Reads	7.04	22.65	8.97	9.29
Average Writes	0	0.2	0.34	0.31
Average Total	7.04	22.85	9.32	9.61

- Worst-case performance:

	OPT	SRR	MTRR	MUP
Worst Reads	9	61	22	22
Worst Writes	0	2	20	16
Worst Total	9	62	40	32

- Worst per node average performance:

	OPT	SRR	MTRR	MUP
Worst Average Reads	9	31.95	9.92	13.81
Worst Average Writes	0	0.26	0.5	0.71
Worst Average Total	9	32.16	10.28	14

7.3 Interpreting the Results

Some of these results are clearly surprising and even counter-intuitive. Why doesn't single-rotation work? How come move-to-root is so close to optimal? How is it possible that move-up with counters does worse than move-to-root? Although we cannot claim to have all the answers, our experience with the simulations did provide us with some insights, which we present in the rest of this subsection.

7.3.1 Being conservative is bad

The results clearly show that single-rotation does not work: The average number of reads per lookup for SR and SRR are far from the optimal. However, intuitively, it *should* work: Frequently accessed nodes are rotated up more often and, thus, they should eventually end up near the root. So, what is going on?

All heuristics that do not use statistics suffer from the same problem: false adjustments. Consider frequently accessed node a, which is initially placed at the bottom of the tree. Under SR, the first time a is accessed, it is rotated up. However, if the next lookup hits

- on one of a's children
- on a node that is not an ancestor of a, but is the child of an ancestor of a

a is moved down. Because a starts at the bottom of the tree, there are many nodes that, if accessed, will cause it to move down. E.g. consider node a in Figure 5 with access probability 0.4 (the numbers on the nodes denote probabilities of access) Suppose a has just been rotated up from the bottom. At the next lookup, a will be moved down, if either of nodes b, c, d or e are accessed i.e., with probability 0.5! In general, the deeper down in the tree the initial location of a node, the greater the difficulty the node will face in moving towards the root, because it is "competing" against a large number of nodes. Clearly, as tree size grows, the situation becomes worse. In practice, single-rotation is so conservative, that frequently accessed nodes which start near the bottom never "take off".

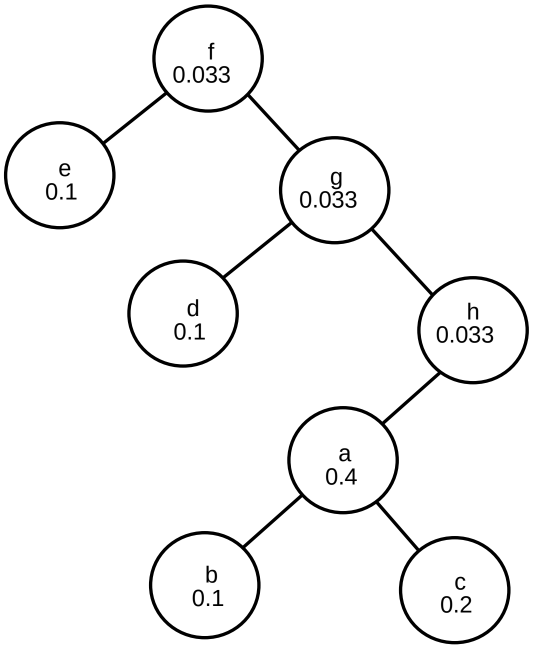

Figure 5: Unlucky frequently accessed node under SR.

To make things worse, consecutive single rotations tend to imbalance the tree – more than consecutive move-to-root's! This, in combination with conservativeness, lead single-rotation average search time away from optimal. Moreover, imbalancing the tree leads to horrible worst-case search time.

7.3.2 Shake the tree

The most significant result is that move-to-root rocks: The average number of reads per lookup for MTR is really close to optimal. Yet, intuitively, it should *not* work: If false adjustments degrade single-rotation performance so much, imagine what they should do to move-to-root, where a false adjustment leads a rarely accessed node to the root!

It turns out that moving up rarely accessed nodes does not hurt. What really hurts is *not* moving up frequently accessed nodes. Consider frequently accessed node a. Under MTR, the first time a is accessed, it is moved to the root. Suppose that a is accessed again after n lookups i.e., $n-1$ lookups hit on other nodes. The worst that can happen to a is that it is moved down $n-1$ levels. However, if a is frequently accessed, n is small. This means that the second access will find a not at the root, but still close to it. In short, move-to-root gives all nodes a chance at the root. How close they remain to the root is proportional to their probability of access.

Of course, the catch is that each lookup costs as many write accesses as read accesses - which is unacceptable for a routing table.

7.3.3 but don't overdo it

Fortunately, randomization works: MTRR is the same with MTR in terms of average number of reads per lookup! This is not a surprising result. Even if we do the move-to-root heuristic with a very small probability, frequently accessed nodes *will* eventually be moved to root and stay close to it.

As mentioned earlier, we tried MTRR both with a fixed probability and with a probability proportional to node depth. Both attempts yield similar results. The second one, though, sounds counter-intuitive: Why favor nodes that are deep down the tree?

The initial idea was the following: If we are to move nodes infrequently, we should at least favor those that are deep down the tree, because it is better to risk bringing up a rarely accessed node, than risk leaving down a frequently accessed one. The problem with this argument is that bringing up a rarely accessed node hurts more when we reorganize the structure infrequently. In practice, favoring "lower" nodes turned out similar to treating all nodes equally, at least in terms of average performance.

However, favoring "lower" nodes is slightly better in terms of worst per node average performance. The reason is obvious: Nodes tend to circulate more. Hence, even the most rarely accessed node spends less time at the bottom.

7.3.4 What's the perfect probability?

Suppose we choose to apply a heuristic with a fixed probability p. Which p will give us the best performance?

Intuitively, there should be the following trade-off: The more we apply the heuristic, the better the tree approximates the optimal BST and, thus, the lower the average number of reads per lookup. Of course, the more we apply the heuristic, the more we have to update the tree and, thus, the higher the average number of writes per lookup.

In reality, applying the heuristic infrequently, decreases the average number of writes – as expected, but, surprisingly enough, does not really decrease the average number of reads per lookup! Of course, there exists some threshold p_{min}. When p drops below p_{min}, the heuristic is not applied frequently enough to have some impact and the average number of reads per lookup starts increasing.

Figures 6 and 7 show average number of reads and average number of writes under MTRR as a function of the probability p. The setup is the same as in scenario 1 i.e., 256 rules, $250,000$ lookups and 99% of the lookups are concentrated on 7 rules. The first graph seems at first surprising: With $p = 0.00001$ i.e., only $0.00005 * 250,000 \approx 13$ applications of the heuristic, average number of reads is already down to 5.23 – note that the optimal average number of reads in this case is 4.56. Of course, this is closely related to the fact that the input distribution of scenario 1 is highly skewed e.g., 50% of lookups hit on a single rule. Thus, it does not take that many applications of the heuristic to bring the few popular nodes to the top.

We have not run enough simulations to provide results on how performance changes as a function of p under different input distributions. However, we suspect that under higher entropy distributions, the heuristic will need to be applied more often to have an impact.

One final thing to note is that p affects the *adaptability* of the structure to varying input distributions. Again, we have not run enough simulations to provide results on this, but we did notice – and it does make sense – that the more rarely we apply the heuristic, the more slowly our structure converges towards the optimal. Intuitively, p should be large enough to match the variations of the input distribution.

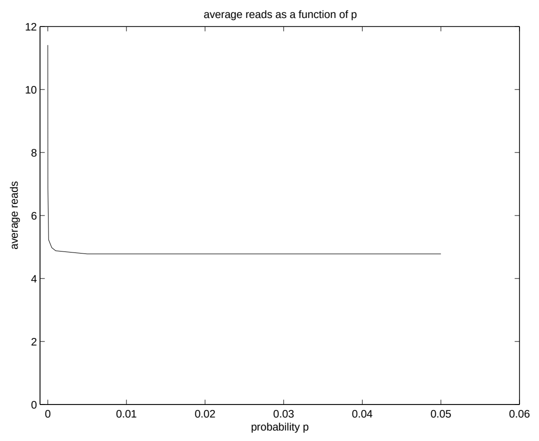

Figure 6: Average reads per lookup

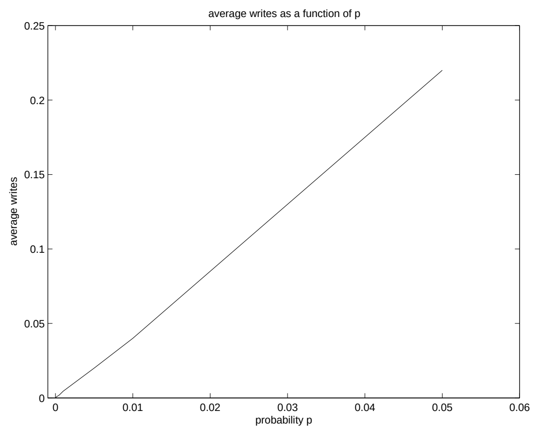

Figure 7: Average writes per lookup

7.3.5 Counters can nail you

Another surprising result is that move-to-root slightly outperforms move-up with counters: MTR and MTRR have a slightly smaller average number of reads per lookup than MUP. Intuitively, this should not be the case. Hit counters only prevent a less frequently accessed node from being rotated above its more frequently accessed parent. How can that be bad?

That's right, it isn't. In fact, MUP is very close to optimal in terms of average number of reads per lookup. However, it has a small defect: It arranges nodes optimally *per branch*. Consider the BST of Figure 8. Under MUP, this BST cannot evolve at all. Nodes c and d are stuck at positions lower than e and f, which are less frequently accessed. In short, MUP has a problem with unbalanced branches and may end up "nailing" frequently accessed nodes at the wrong positions.

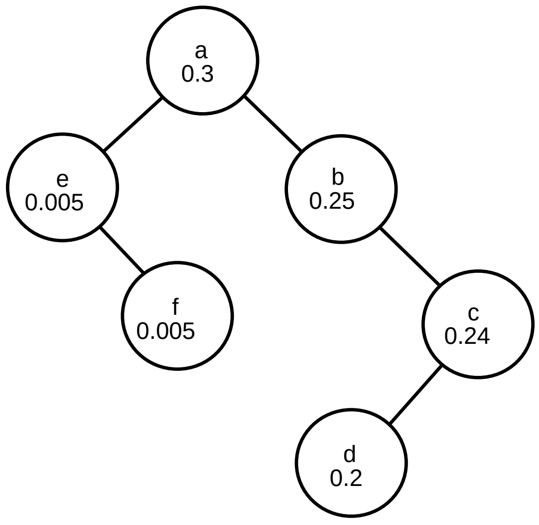

Figure 8: Non-optimal MUP final state

7.3.6 Is worst-case performance really bad?

Yes, it certainly is. Even MTRR and MUP, which have excellent average performance, have double worst-case number of reads compared to the optimal BST. Not to mention that worst-case number of writes is comparable to worst-case number of reads. This is due to the fact that all heuristics tend to imbalance the tree by moving nodes up and down.

However, there is one thing to note here: As opposed to what happens in a static structure (like the optimal BST), in a dynamic structure no single node suffers worst-case performance alone – unless it has a significantly smaller probability of access than the rest of the nodes. This becomes obvious from the worst per node average performance results, where MTRR does better than the optimal.

7.4 Bounding the Tree Depth

In this subsection, we show some simulation results for the MTRB (move-to-root-bounded) structure and compare them with corresponding results for MTRR and the optimal BST.

7.4.1 Scenario 1

	OPT	MTRR	MTRB
Average Reads	4.56	4.95	6.3
Worst-case Reads	12	21	16
Worst-case Writes	0	21	16
Worst per node Reads	12	11.81	16

7.4.2 Scenario 2

	OPT	MTRR	MTRB
Average Reads	6.79	7.34	8.32
Worst-case Reads	10	22	16
Worst-case Writes	0	22	16
Worst per node Reads	12	11	16

7.4.3 Does it cost?

These results clearly show that bounding the tree depth costs: The average number of reads per lookup increases significantly. Another thing to note is that worst per-node average performance is the same with overall worst-case performance.

These results are not surprising: Once a branch reaches maximum depth, MTRB simply "locks" it at that position and does not allow it to evolve any more. This affects average performance, because frequently accessed nodes may not be allowed to move up. It also affects worst per node average performance, because any unlucky node that gets "pinned" near the bottom, stays there forever.

8 Conclusions

We considered the problem of building a routing table structure, which would adjust itself according to the distribution of of the probabilities of accessing the different rules. To do this, we started with a simple Binary Search Tree (BST) structure, whose keys are the disjoint intervals that make up the routing table prefixes. Using this as a base, we experimented with a number of simple heuristics.

Our first results show that the single-rotation heuristic does not work well. On the contrary, the move-to-root heuristic approximates very well the optimal BST in terms of average number of reads per lookup. The problem is that move-to-root costs a lot in terms of average number of writes per lookup. Fortunately, randomization works well: applying the heuristic very infrequently keeps the average number of reads close to optimal, while reducing significantly the average number of writes.

We also touched upon the issue of *when* to apply the move-to-root heuristic: As a function of the accessed node depth? With a fixed probability p? What should that be? Although we do not have complete results on this, we conclude that with a very small probability p, it is still possible to achieve an average number of reads close to optimal. However, applying the heuristic very infrequently affects the adaptability of the structure to changing input distributions.

Finally, we built a self-adjusting structure with bounded depth. However, our simulation results show that bounding the tree depth in a naive way has a significant negative impact on average performance. Thus, the next step will be to bound tree depth in a smarter way, which will allow the tree to evolve, while maintaining a minimal balance.

References

[1] Pankaj Gupta et. al., *Near-Optimal Routing Lookups with Bounded Worst Case Performance* , , [INFOCOMM]

[2] G. Cheung et. al., *Optimal Routing Table Design for IP Address Lookup under Memory Constraints* , [INFOCOMM 1999]

[3] B. Allen, I. Munro *Self-Organizing Binary Search Trees*, [ACM Journal, 1978]

[4] J.R. Bitner, *Heuristics that Dynamically Organize Data Structures*, [SIAM Journal on Computing, Vol. 8, No. 1, pp 82-110, 1979]

[5] D. Sleator, R. Tarjan, *Self-Adjusting Binary Search Trees* , [ACM, 1985]

[6] C. Martinez, S. Roura, *Randomized Binary Search Trees*, [1997]

[7] D.E. Knuth, *Sorting and Searching*, [The Art of Computer Programming, Vol. 3, 1973]

[8] T. Cormen, C. Leiserson, R. Rivest, *Red-Black Trees, Rotations*, [Introduction to Algorithms, 1st Edition, pp 262, 1990]

[9] http://klamath.stanford.edu/tools/PALAC/

[10] B. Lampson, V. Srinivansan, G. Varghese, *IP Lookups using Multiway and Multicolumn Search*

[11] T. Cover, J. Thomas *Elements of Information Theory*, [Wiley-Interscience, 1991]

[12] S. Resnick *Adventures in Stochastic Processes*, [Birkhaeuser, 1992]