Abstract

We propose a randomization scheme for splay trees with the same time complexity as deterministic splay trees, but lower constants. We also present experimental results.

1 Introduction

Binary search trees (BSTs) are a hierarchical data structure for dynamically storing and retrieving data. Traditionally, operations on BSTs and other data structures have been studied using the RAM model of computation. However, it is often convenient to define a new model of computation based on BSTs. This model of computation allows the following unit-cost operations:

1. Rotation - Perform a rotation between a node and its parent. By checking if the node is the right or left child of its parent, we can determine if we should do a right rotation or a left rotation. An example rotation is shown in figure 1.

2. Move - Follow a pointer from the current node to its parent or child. This allows us to search for a node (by following child pointers) and find the successor and predecessor of a given node.

3. Augment - Augment the current node to store some information. This encompasses inserting and deleting a child node by updating the current node’s child pointers.

Accessing a node in a BST takes $O(h)$ time, where $h$ is the height of the BST. To prevent the height from growing too large, most BSTs have a procedure for remaining balanced, leading them to take $O(\log(n))$ time for each operation. For any BST with $n$ nodes, there is always a node at height at least $\log(n)$, so this is the best we can hope to do for any single access.

If we look at a longer sequence of accesses, it seems that we should be able to do better by performing some rotations after we access some nodes. Unfortunately, intuition fails here. For any BST, we can always construct a sequence of $m$ accesses which will take $O(m \log(n))$ time. This implies that any balanced BST is asymptotically optimal.
Figure 1: Splay tree operations. Top left: Original tree before splaying. Top right: After splaying $x$ (zig). Lower left: After splaying $u$ (zig-zig). Lower right: After splaying $v$ (zig-zag).

<table>
<thead>
<tr>
<th>m</th>
<th>Number of accesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>Number of keys</td>
</tr>
<tr>
<td>q_i</td>
<td>Probability of accessing item $i$</td>
</tr>
<tr>
<td>h</td>
<td>Height of tree</td>
</tr>
<tr>
<td>d</td>
<td>Cost of a rotation (if following a pointer has unit-cost)</td>
</tr>
<tr>
<td>$w(u)$</td>
<td>Weight assigned to node $u$</td>
</tr>
<tr>
<td>$\rho(u)$</td>
<td>Log of sum of weights in subtree rooted at node $u$</td>
</tr>
<tr>
<td>${a_m}$</td>
<td>Sequence of accesses</td>
</tr>
<tr>
<td>$T_u$</td>
<td>Subtree rooted at node $u$</td>
</tr>
<tr>
<td>$T_i$</td>
<td>Tree after servicing access $i$ ($T_0$ is the original tree).</td>
</tr>
<tr>
<td>$\Phi(T)$</td>
<td>Sum of $\rho(u)$ for all $u$ in tree $T$.</td>
</tr>
<tr>
<td>$r$</td>
<td>Constant factor for potential function</td>
</tr>
<tr>
<td>$t(a_i)$</td>
<td>Time to find and splay access $i$</td>
</tr>
<tr>
<td>$t^a(a_i)$</td>
<td>Amortized time to find and splay access $i$</td>
</tr>
<tr>
<td>$x_p$</td>
<td>Parent of node $x$</td>
</tr>
<tr>
<td>$x_g$</td>
<td>Grandparent of node $x$</td>
</tr>
</tbody>
</table>

Table 1: Variables for describing BSTs
We can have a richer study of BSTs if we measure time in a different way. Instead of simply counting the maximum number of operations required for any sequence of \( m \) accesses, we will look at the time required by our BST \textit{relative} to the optimal BST when both are given the same sequence of accesses. Intuitively, some sequences of accesses are hard and \textit{all} BSTs will take a lot of time accesses these elements. This means that the \textit{relative} time complexity will remain small. Formally, we define the \textit{competitive ratio} of a BST to be the maximum ratio between the number of operations performed by the BST and the number of operations performed by the optimal BST. Note that the optimal BST requires at least 1 operation per access and balanced BSTs require at most \( O(\log(n)) \) operations per access, so any balanced BST achieves a competitive ratio of \( O(\log(n)) \). Thus, we are interested competitive ratios which are \( o(\log(n)) \). It is important to remember that there may not exist an algorithm for computing the optimal BST. Even if such an algorithm exists it may not be efficiently computable.

There are a couple of different versions of this problem. For \textit{static} optimality, the BST is fixed \textit{a priori}. For \textit{dynamic} optimality, the BST can be altered when accessing a node. In the \textit{offline} version of the problem, we are given access to the entire sequence of queries ahead of time; the \textit{online} version only sees the queries as they arrive in real time.

Note that the online static problem is not interesting. The BST is fixed ahead of time, so an adversary could always construct a sequence of accesses which take \( O(\log(n)) \) time. This means that \textit{any} balanced BST is statically optimal.

## 2 Previous Work

### 2.1 Deterministic Algorithms

Knuth \cite{Knuth1971} gave a deterministic algorithm which uses dynamic programming to solve the offline, static optimality problem.

For the offline, dynamic optimality problem, we cannot use Knuth’s method. Currently, the best known method is to simply simulate all possible operations after every access. We can bound the number of operations by the number of operations required for any BST: \( O(m \log(n)) \). This gives an exponential (yet deterministic) algorithm for computing the best offline dynamic BST.

Tarjan and Sleator \cite{TarjanSleator1985} presented splay trees in 1985. The key to splay trees is the splay operation, which is executed after every Find operation. Splaying a node means to perform a sequence of rotations to the node and its parents such that the node is moved to the root of the BST. The order of rotations depends on the orientation of the node, its parent, and its grandparent.

1. (zig) If a node does not have a grandparent, rotate the node.
2. (zig-zig) If a node its parent are both left children or both right children of their respective parents, then rotate the parent and then rotate the node.
3. (zig-zag) Otherwise, the node and its parent appear on opposite sides of their respective parents. Rotate the node, which brings it up one level, and then rotate it again.
The intuition behind splay trees is that frequently accessed items will be moved towards the root of the tree. This means that subsequent accesses will take less time.

Tarjan and Sleator showed that if accesses are generated from a fixed probability distribution, splay trees will be within a constant factor of the optimal static BST. However, they were unable to show that splay trees were dynamically optimal, leaving it as an open problem (the “Dynamic Optimality Conjecture”) which they conjectured to be true.

In 2004, [ED04] invented Tango Trees, which come very close to being being dynamically optimal. Tango Trees are a $O(\log(\log(n)))$ competitive, meaning for any sequence of accesses on a tree with $n$ nodes, Tango Trees will do at most a factor of $\log(\log(n))$ more work than the optimal BST on the same sequence. While this is certainly not constant ($O(1)$), in practice $\log(\log(n))$ is less than 10 unless the number of keys is many orders of magnitude larger than the number of atoms in the universe. This is the only BST which provably beats the trivial $O(\log(n))$ competitive ratio.

As of 2015, the Dynamic Optimality Conjecture remains open, and Tango Trees have the best known competitive ratio.

### 2.2 Randomized Algorithms

In Tarjan and Sleator’s original paper, they proposed one randomization scheme: stop splaying after a certain number of accesses. Specifically, after each access, with probability $p = 1/m$ decide to stop splaying for every subsequent access. We will call this randomization scheme $\text{RandSplay1}$. Once they have stopped splaying the expected average time per access is $O(n \log(n)/m + \sum_{i=1}^{n} q_i \log(1/q_i))$, where $q_i$ is the probability of accessing item $i$. Note that this does not include the initial splays, which costs $O((m + n) \log(n))$.

In 2002, Albers and Karpinski [AK02] proposed another randomization scheme. Instead of setting a cutoff point for when to stop splaying, they decide to splay after every access with some probability $p$, independently. We call this approach $\text{RandSplay2}$. By examining the ratio $d$ between the cost of a rotation and the cost of following a parent pointer, Albers and Karpinski showed that $\text{RandSplay2}$ has an expected total access time of $O(1 + pd)(3m \log(n) + m) + (1/p + d)n \log(n))$.

### 3 Approach

We analyze two alternative randomization schemes. Both start splaying after each access, but do not splay the accessed node all the way to the root.

$\text{RandSplay3}$ - Starting at the accessed node, with probability $p$ perform the usual zig, zig-zig, or zig-zag step. Otherwise, we do nothing and move to the node’s parent.

---

1. Before Tarjan and Sleator invented splay trees, two similar strategies using this idea were presented:

   1. Move To Root - Keep rotating the accessed node until it arrives at the root (always zig)
   2. Exchange - Rotate the accessed node once.

Unfortunately, [AM78] showed that there is a sequence of accesses of arbitrary length which requires $O(n)$ time per access.
RandSplay4 - Starting at the accessed node, start splaying as usual. However, after each zig, zig-zig, and zig-zag step, flip a coin with bias $p$. If the coin lands on heads, continue splaying; otherwise, we stop splaying and immediately return.

Intuitively, these strategies are attractive. Like ordinary splay trees, they bring frequently accessed elements to the top of the tree. However, they do not do as much work restructuring the tree. When nodes which are accessed with low frequency are accessed, ordinary splay trees bring these nodes to the root, which is a bad idea because these nodes probably will not be accessed again soon. Using our strategy, frequently accessed nodes will eventually rise to the top of the tree, but infrequently accessed nodes will only move part of the way up the tree. A second benefit is that the variance in access time decreases. Specifically, when we access a very deep node, it is very unlikely that we will splay that node all the way to the root.

4 Theoretical Results

4.1 Review of Deterministic Splay Trees

Sleator and Tarjan proved a number of results about the performance of deterministic splay trees using a powerful potential function for trees. Their results were all amortized. We first present include Sleator and Tarjan’s analysis of deterministic splay trees and then adapt it for randomized splay trees.

First, we define some notation. Let $w(\cdot)$ be a function that assigns weights to vertices. Now, let $\rho(u)$ indicate the (log) total weight of the subtree rooted at $u$:

$$\rho(u) = \log \sum_{v \in T_u} w(v)$$

In a couple places we will abuse notation to let $\rho'(u)$ be the value of $\rho(u)$ after a zig-zig/zig-zag/zig and let $\rho_\alpha(u)$ be the value of $\rho(u)$ after access $a_\alpha$. 

Figure 2: Reference BST used for analysis
Finally, define the potential of tree $T$ as

$$\Phi(T) = \sum_{v \in T} \rho(v)$$

This potential function has two nice properties:

1. Intuitively, our potential function is a measure of how balanced our tree is. The more balanced the tree is, the higher the potential function will be.

2. The weights $w$ are arbitrary. Picking different weight functions $w$ will allow us to prove different properties about the splay tree.

We can use our potential function to bound the total amortized time spent on $m$ operations. Given a sequence of "search" operations $a_1, a_2, ..., a_m$, let $t(a_i)$ denote the time spent on the operation. Let $T_i$ denote the tree after our $i^{th}$ operation with $T_0$ denoting the original tree. We define $t^a(a_i)$ to be the amortized time to find and splay access $a_i$:

$$t^a(a_i) = t(a_i) + r(\Phi(T_i) - \Phi(T_{i-1}))$$

The value $r$ is some constant that can be thought of a scaling factor. While it seems like a minor detail, accurately choosing $r$ is crucial to our analysis since a clever choice for $r$ will allow for a particularly elegant bound of $t^a(a_i)$.

Therefore, our total running time can be determined from the total amortized time:

$$\sum_i t(a_i) = \sum_i t^a(a_i) + r(\Phi(T_0) - \Phi(T_m))$$

Note that $\Phi(T)$ only depends on that locations of the $n$ nodes in the tree, not on the number of accesses. This means that as $m$ grows to be much larger than $n$, the difference in potentials becomes negligible, and $t^a(\cdot)$ converges to $t(\cdot)$. We will show that while $t$ can have wild values (if $T_0$ is initially a tree of depth $O(n)$, then $t$ can be really large), $t^a$ has tamer behavior.

To compute the total splaying time, we will examine the amortized costs of zig, zig-zig, and zig-zag operations and then sum them over all such operations. Although we could have $O(n)$ many such operations, the sum of the costs of all the zigs, zig-zigs, and zig-zags telescopes, resulting in a small net amortized cost. To prove this, we show the following three inequalities hold for some constant $r$:

$$t^a(\text{zig}) \leq 2 + d + 3r(\rho'(x) - \rho(x))$$
$$t^a(\text{zig-zig}) \leq 3r(\rho'(x) - \rho(x))$$
$$t^a(\text{zig-zag}) \leq 3r(\rho'(x) - \rho(x))$$

**Zig operation:** Let $x$ be the current node, and let $p$ be the parent of $x$. Assume that following a parent or child pointer cost 1 unit of time and that rotations cost $d$ units of time. Finally, let $r$ be some constant. Because we zig only if a node has depth at most 2, the cost of a zig is $2 + d$. To get the amortized cost, we must add the change in potential. Let $\rho'(\cdot)$ refer to updated $\rho(\cdot)$ values after the tree is changed by some operation. This yields an amortized cost of

$$t^a(\text{zig}) = 2 + d + r(\rho'(x) + \rho'(x_p) - \rho(x) - \rho(x_p))$$
The key idea here is that the only vertices who’s \( \rho \) value change are \( p \) and \( x \) which means summing these changes gives us the net change in potential function. Now, \( x \) increases in potential while \( x_p \) decreases in potential, so \( \rho(x_p) - \rho'(x_p) > 0 \). Now,

\[
t^a(zig) = 2 + d + r(\rho'(x) + \rho'(x_p) - \rho(x) - \rho(x_p))
\]

\[
\leq 2 + d + r(\rho'(x) - \rho(x))
\]

\[
\leq 2 + d + 3r(\rho'(x) - \rho(x))
\]

**Zig Zig operation:** Let \( x \) be the current node, \( x_p \) be the parent of \( x \), and \( x_g \) be the grandparent of \( x \). We now make the substitution that \( r = 2 + d \), allowing us to absorb the time cost into our potential function:

\[
t^a(zig-zig) = 4 + 2d + r(\rho'(x) + \rho'(x_p) + \rho'(x_g) - \rho(x) - \rho(x_p) - \rho(x_g))
\]

\[
= (2 + d)(2 + \rho'(x) + \rho'(x_p) + \rho'(x_g) - \rho(x) - \rho(x_p) - \rho(x_g))
\]

The final step is to show the following claim:

\[
2 + \rho'(x) + \rho'(x_p) + \rho'(x_g) - \rho(x) - \rho(x_p) - \rho(x_g) \leq 3\rho'(x) - 3\rho(x)
\]

Proving this is not too difficult, and we omit the proof. The main idea behind it is to explicitly write out \( \rho(x), \rho(x_g), \rho'(x), \rho'(x_g) \) in terms of \( w \) and then use the fact that \( \log \) is a concave function. Interested readers can find a more detailed proof in [ST85].

Using this claim, our amortized time becomes

\[
t^a(zig-zig) \leq (2 + d)3(\rho'(x) - \rho(x))
\]

**Zig-Zag:** The proof is nearly identical to the one for zig-zig, and can be found in [ST85].

**Total:** We can finally bound the total amortized time \( t^a(a_i) \). Summing up all the zig-zigs, zig-zags, and zigs gives the total amortized time, which telescopes to

\[
t^a(a_i) \leq 2 + d + 3(2 + d)(\rho_i(a_i) - \rho_{i-1}(a_i))
\]

At this point, we’ve done all the grunt work for analyzing splay trees. All that’s left to do is choose a weight function \( w \). Here is a quick example.

**Theorem 1.** Executing \( m \) find operations on a splay tree of \( n \) keys takes time at most

\[
(2 + d)m + 3(2 + d)m \log n + n \log n
\]

This means that for \( m \geq n \), our average time per query is at most

\[
2 + d + 3(2 + d) \log(n) + \log(n)
\]

**Proof.** We have that \( t^a(a_i) \leq 2 + d + 3(2 + d)(\rho'(a_i) - \rho(a_i)) \). Let \( w(a_i) = 1 \) for all \( a_i \). Then \( \rho(x) \leq \log(n) \) for all \( x \) meaning that

\[
t^a(a_i) \leq 2 + d + 3(2 + d) \log(n)
\]
Substituting this in yields that

\[ \sum_{1}^{m} t^a(a_i) = (2 + d)m + 3(t + d)\log(n) \]

so to get \( \sum_{1}^{m} t(a_i) \) we just add in

\[ \Phi(0) - \Phi(m) \leq \Phi(0) \leq n\log(n) \]

Adding this in proves the theorem \( \Box \)

Like this, many other interesting (and more significant) results about splay trees can be shown. All that really changes are what we pick for the weights \( w(x) \).

4.2 Analysis for Random Splay Trees

We now adapt the above analysis to randomized splay trees. In this section, we will only analyze RandSplay3. RandSplay4 has a similar analysis.

Similar to above, we define \( t(a_i) \) to be the (random variable) representing the amount of time querying \( a_i \) takes and \( t^a(a_i) \) to be the random variable representing the amortized time spent querying \( a_i \). They are related by the same equation as their deterministic counterparts:

\[ t^a(a_i) = t(a_i) + r(\Phi(T_i) - \Phi(T_{i-1})) \]

This equation is identical to the one before, except this time all the variables (including \( \Phi(T_i), \Phi(T_{i-1}) \)) are now random variables. Here, \( r \) is still a constant, but different from the one used in the deterministic case.

When we analyzed \( t^a(a_i) \) in the deterministic case, we broke it up into a bunch of individual pieces each of which corresponded to a splay operation. In the randomized case, we can do the same thing. Recall that in RandSplay3, we either splay the node \( x \) (using zig, zig-zag, or zig-zig) with probability \( p \), or we simply move our pointer to its grandparent \( x_g \) with probability \( (1 - p) \). We define \( t^a_j(a_i) \) to be the amortized cost incurred at depths \( j \) and \( j + 1 \). Note that each coin tosses determining whether to zig-zag/zig-zig or follow pointers to the node’s grandparent is independent. This implies that given \( a_i \), each \( t^a_j(a_i) \) is independent.

We can now write the expected amortized time as

\[ E[t^a(a_i)] = E[\sum_{j=1}^{1/2\text{depth}(a_i)} t^a_j(a_i)] = \sum_{j=1}^{1/2\text{depth}(a_i)} E[t^a_j(a_i)] \]

We now consider the three types of splay operations.

**Zig:** With probability \( (1 - p) \), we do not splay and just go to the root. With probability \( p \) we do the entire splay operation, so we just plug in the value from our analysis in the deterministic case:

\[ E[t^a_j(a_i)] = (1 - p)(2) + p(2 + d + r(\rho'(x) - \rho(x))) \leq 2 + p + pd + 3r(\rho'(x) - \rho(x)) \]

Note that the node which gets zigged, \( x \), is not deterministic, but the values \( \rho(x) \) and \( \rho'(x) \) are. This is an important observation. To see why this holds, the subtree
rooted at $x$ contains the same nodes no matter what our previous operations are. This is because $x$ will always be the node that is a distance of $2j$ from the original position of $a_i$. Therefore, we can leave our expression as is and not be concerned with $x$ (in fact we can pretend that $x$ behaves exactly as it does in the deterministic case).

**Zig Zig:** With probability $(1 - p)$ we do not splay, and the cost is 4: 2 for going to the grandparent, and 2 for the search down to $x$. If we do splay, splaying costs exactly as it did in the deterministic case. Therefore,

$$E[t^a_j(a_i)] = (1 - p)(4) + p(4 + 2d + r(\rho'(x) + \rho'(x_p) + \rho'(x_g) - \rho(x) - \rho(x_p) - \rho(x_g))$$

In order to show the same upper bound we did in the deterministic case, we need to select $r$ so that the constant on the inside is 2 again. In particular, we desire

$$\frac{4 + 2pd}{rp} = 2,$$

which means $r = d + \frac{2}{p}$. Substituting this and substituting

$$\rho'(x) + \rho'(x_p) + \rho'(x_g) - \rho(x) - \rho(x_p) - \rho(x_g) + 2 \leq 3(\rho'(x) - \rho(x))$$

yields that

$$E[t^a_j(a_i)] \leq (2 + dp)3(\rho'(x) - \rho(x))$$

As before, $x$ is not deterministic as it varies depending on what happened in the previous splay operations. However, the same argument from the zig operation applies for $x$, so $\rho(x)$ is deterministic (i.e. dependent only on $a_i$ and $T_{i-1}$ and independent of all the splay operations before $t^a_j(a_i)$).

**Zig Zag:** Just as before, this case is analogous to the zig-zig case, yielding

$$E[t^a_j(a_i)] \leq (2 + dp)3(\rho'(x) - \rho(x))$$

**Total:** Since the values of $x$ are independent of our coin tosses, we can assume without loss of generality that $x = a_i$ and that the values $\rho(x)$ are the values that would arise if we deterministically splayed. Doing this allows us to have the same telescoping effect giving us that

$$E[t^a(a_i)] = \sum_j E[t^a_j(a_i)] \leq (2 + p + pd) + 3r(\rho_i(a_i) - \rho_{i-1}(a_i))$$

as desired. This bound is the same form as the deterministic bound, but with different constant factors (depending on $p$). Therefore, any bound shown for deterministic splay trees by using this potential method also applies to randomized splay trees. For example, we have the following adaptation of Theorem 1.

**Theorem 2.** Accessing $m$ items in a randomized splay tree of $n$ keys takes time at most

$$(2 + p(1 + d))m + 3(2 + pd)m \log n + (d + \frac{2}{p})n \log n$$

This means that for $m \geq n$, our average time per query is at most

$$2 + p(1 + d) + 3(2 + pd) \log(n) + (d + \frac{2}{p})n \log n / m$$

9
5 Experimental Results

We implemented Splay Trees with deterministic splaying and four randomization variants. The code was written in C++, where we represented the tree as doubly-linked struct nodes. In our implementation, rotations were a factor of 6 more expensive than following parent/child pointers ($d = 6$).

We tested all 5 splaying strategies on a sequence of inputs. Each test was initialized with a perfect binary tree of height $h = 32$ with $n = 2^{h+1} - 1$ unique integer keys $1, 2, ..., n$. The sequence of accesses had $32n$ items, randomly drawn from a probability distribution over the keys. For the randomized schemes, we tested many values for $p$. Every test was repeated 100 times, and the average time was reported.

We choose to sample from Zipf’s Distribution because it closely models the distribution of accesses in the real world. George Kingsley Zipf first studied these distributions with regards to natural languages. Since then, it has been shown that many other phenomenon follow this distribution: census data, patterns in music, and sizes of companies. Zipf’s Distribution assigns probability to item $i$ inversely proportional to the rank of item $i$ in the sorted list of accesses (see fig. 3).

Our strategy showed a significant improvement over deterministic splaying, but did not beat the two simpler randomization schemes ($\text{RandSplay1}$ and $\text{RandSplay2}$). We believe that this is because additional overhead required by our schemes outweighed the benefits of randomization at a finer scale. Additionally, we believe that $\text{RandSplay4}$ does worse than $\text{RandSplay3}$ because it splays too little. Using $\text{RandSplay4}$, the probability of continuing to splay a given access decreases exponential as that node is rotated closer to the root. This forms a geometric series, so the expected number of zigs, zig-zigs, and zig-zags is constant ($O(1)$). In contrast, $\text{RandSplay3}$ does $\frac{1}{2}hp$ splay operations in expectation.

6 Conclusion

Splay Trees are conjectured to be dynamically optimal. Two coarse randomization schemes proposed by [ST85] and [AK02] were shown to decrease the average access
Table 2: Average total access time for different splaying strategies. The first column corresponds to never splaying, while the second column corresponds to always splaying; neither depends on $p$, so all entries are identical. Bold entries correspond to the best performance for each scheme.

<table>
<thead>
<tr>
<th>$p$</th>
<th>Splay</th>
<th>RandSplay1</th>
<th>RandSplay2</th>
<th>RandSplay3</th>
<th>RandSplay4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{-1}$</td>
<td></td>
<td>67.8157</td>
<td>144.7972</td>
<td>174.1983</td>
<td>139.4108</td>
</tr>
<tr>
<td>$2^{-2}$</td>
<td>70.5716</td>
<td>111.0207</td>
<td>140.6240</td>
<td>190.4780</td>
<td></td>
</tr>
<tr>
<td>$2^{-3}$</td>
<td>71.1186</td>
<td>91.6139</td>
<td>117.7125</td>
<td>215.6632</td>
<td></td>
</tr>
<tr>
<td>$2^{-4}$</td>
<td>71.5175</td>
<td>88.3024</td>
<td>108.2934</td>
<td>216.5675</td>
<td></td>
</tr>
<tr>
<td>$2^{-5}$</td>
<td>71.1962</td>
<td>77.4494</td>
<td>102.8874</td>
<td>196.7062</td>
<td></td>
</tr>
<tr>
<td>$2^{-6}$</td>
<td>71.1582</td>
<td>75.6040</td>
<td>100.8970</td>
<td>195.0340</td>
<td></td>
</tr>
<tr>
<td>$2^{-7}$</td>
<td>71.3919</td>
<td>74.8931</td>
<td>97.1101</td>
<td>190.0399</td>
<td></td>
</tr>
<tr>
<td>$2^{-8}$</td>
<td>74.3924</td>
<td>73.1489</td>
<td>96.6507</td>
<td>180.7443</td>
<td></td>
</tr>
<tr>
<td>$2^{-9}$</td>
<td>71.1868</td>
<td>74.9089</td>
<td>93.9916</td>
<td>184.8084</td>
<td></td>
</tr>
<tr>
<td>$2^{-10}$</td>
<td>69.1606</td>
<td><strong>69.9218</strong></td>
<td><strong>92.5387</strong></td>
<td>185.6226</td>
<td></td>
</tr>
</tbody>
</table>

time by a constant factor. We showed that a finer randomization scheme also gives a constant factor speedup. However, our randomization scheme did not give better performance than the coarse randomization schemes.

References


