Searching Large Indexes on Tiny Devices: Optimizing Binary Search with Character Pinning

Guy Shani
Microsoft Research
One Microsoft Way
WA, USA
guyshani@microsoft.com

Christopher Meek
Microsoft Research
One Microsoft Way
WA, USA
meek@microsoft.com

Tim Paek
Microsoft Research
One Microsoft Way
WA, USA
timpaek@microsoft.com

Bo Thiesson
Microsoft Research
One Microsoft Way
WA, USA
thiesson@microsoft.com

Gina Venolia
Microsoft Research
One Microsoft Way
WA, USA
ginav@microsoft.com

ABSTRACT
The small physical size of mobile devices imposes dramatic restrictions on the user interface (UI). With the ever increasing capacity of these devices as well as access to large online stores it becomes increasingly important to help the user select a particular item efficiently. Thus, we propose binary search with character pinning, where users can constrain their search to match selected prefix characters while making simple binary decisions about the position of their intended item in the lexicographic order. The underlying index for our method is based on a ternary search tree that is optimal under certain user-oriented constraints. To better scale to larger indexes, we analyze several heuristics that rapidly construct good trees. A user study demonstrates that our method helps users conduct rapid searches, using less keystrokes, compared to other methods.

Author Keywords
Binary Search, Optimal Binary Search Tree

ACM Classification Keywords
H.3.3 INFORMATION STORAGE AND RETRIEVAL: Information Search and Retrieval—*Retrieval models*

INTRODUCTION

Mobile devices continue to shrink in size with ever greater capacity. Portable media player devices exemplify this trend. According to market research, the size of consumers' digital music collections is steadily increasing with 18% of users in 2006 carrying more than 1000 songs on their portable devices [4]. Furthermore, future devices will likely offer anytime access to online music stores [3], where users will be able to browse huge catalogs of recorded music. In order to address the market demand for tiny media players, different companies have introduced a wide range of small form factor products, with and without a small graphical user display (e.g. Figure 1). However, the small physical size of such devices only allows for a limited user interface (UI), typically a small number of buttons and a single line display. For example, the iPod Shuffle[1] has a *directional pad* (d-pad) with one center and four direction buttons, and no display unit.

Given such a limited UI, selecting a song or artist from a large list (which could potentially be all the music available in an online store), becomes challenging. Simple approaches, such as iterating through the index item by item, fail to scale to large indexes.

Figure 1. Examples of limited UI in MP3 Players.

This paper addresses the problem of rapidly retrieving an item from a large sorted index using just a d-pad and a single line display for text input or browsing. Given that some items are more popular than others, our method takes as input a probability distribution representing the likelihood of accessing each item in the list. In particular, we propose a new information retrieval technique called *ternary search*, which augments an optimal binary search tree (OBST [6]) with character pinning [7].

To augment an OBST with character pinning, we transform

[1] www.apple.com/ipodshuffle

IUI'09, February 8 - 11, 2009, Sanibel Island, Florida, USA.

the binary tree into ternary form, and impose some user-oriented constraints on its structure. We present a dynamic programming (DP) [2] algorithm for the creation of the ternary tree, equivalent to the OBST construction method. As this algorithm scales poorly to large lists, we also develop several fast heuristics for the tree construction.

To validate our approach we conduct both an analytical evaluation as well as a real user study. In the analytical evaluation, we experimentally compare the expected search cost of a variety of models, ranging from optimal binary search trees to our various heuristics for ternary search trees, on two data sets. Our method significantly reduced expected search cost. In the user study, we compare our method to not only binary search but also a typical spelling interface on a search task involving only 5 actions and a single line display. The results are promising, showing our method to be superior both in terms of search time and the number of keystrokes. Users also preferred our method.

This paper is organized as follows; First, we provide background on search interfaces for small devices as well as optimal binary search tree construction. Second, we explain how a binary search tree can be extended into a ternary search tree to support pinning. We elaborate on the difficulties and suggest several heuristics for rapid tree construction. Then, we evaluate our method both analytically as well as with a user study, discussing the implications of our findings. We conclude with a discussion of related work and with an outline of future research extensions.

BACKGROUND

Searching for Item in Lists

Perhaps the simplest way to search for items is via linear search of an ordered list. The user is forced to scan each item in the order of appearance (typically lexicographic) until he finds the *search item*. Most mobile devices employ linear search for lists, such as a contact list on cell-phones. Some devices enhance linear search by allowing users to scan quickly in larger steps by pressing and holding a key.

Another simple search method is spelling. On devices with a large interface for text input, such as PDAs, this method is efficient [5]. Some users even manage to rapidly write text on numeric cell-phones with only 12 buttons available. However, on smaller devices with only 5 buttons, such as a d-pad, text entry becomes burdensome. Spelling is typically accomplished using the up and down keys to scroll through a list of letters. Once the desired letter has been reached, the user selects (*pin*) it and move to the next letter. This process can be enhanced by stopping once the prefix matches only a single item in the list.

Given the vast research on data structures for facilitating fast and efficient list search, we explore the possibility to leverage this work to build a fast and efficient 5-button interface. One popular data structure for list search is the *Binary Search Tree* (BST [6]), which can be ordered according to the lexicographic order of string items.

To search a list using a BST, the retrieval system first suggests the item stored in the current node, which we call the "suggested item". The user specifies whether the item he is looking for (i.e., the *search item*) occurs before (up key) or after (down key) the current item in the list. If the suggested item is the search item, the user can accept (center key) the suggested item. Otherwise, the system moves to one of the children of the current node, and suggests the item that is stored in that node.

All the search methods mentioned so far implicitly assume that the user is equally likely to search for any item. In reality, some items are more popular than others. For example, a user might prefer to listen to certain songs more often than others. Given a distribution over item popularity, the search process could be augmented. In the binary tree case, items that are closer to the root are reached using a smaller amount of suggestions. Intuitively, it could be beneficial to place popular items closer to the root and unpopular items in the leaves.

Furthermore, we could expedite binary search by allowing users to accept the beginning characters of a suggested item that match the search item, even if the entire suggested item itself does not. Characters that are accepted, or 'pinned' by the user, can be used to constrain the next suggested item.

Problem Definition

Let $K = \langle s_1, s_2, ..., s_n \rangle$ be a list of key strings, sorted in lexicographic order. Let $pr(s_i)$ be a probability distribution over K. Let $s \in K$ be a target search string. A search for s in the list uses the following process:
1. The system presents a suggested string $s' \in K$ to the user. Some (possibly empty) prefix of s' is marked as pinned, and the character immediately after the pinned prefix is called the *active character*.
2. The user selects an action a out of a restricted set of actions. Specifically, the user can accept the suggestion (i.e. $s = s'$), specify whether s comes before or after s' with respect to the active character, or pin the active character.
3. Following the user action, the system presents a new suggestion.
4. The search ends successfully when the application suggests s to the user, and fails if no item in the list matches the constraints specified by the user's actions.

For now, we assume a deterministic search. If s is searched for twice, the same list of suggestions will be presented in the same order. Alternatively, one could personalize the search method based on the user history [1], but we do not consider this extension here. Let $c(s)$ be the cost of searching for string s. A cost might be the number of suggested strings before the target string s is found, or the time it takes to search for an item. Our goal is to minimize the expected search cost over the entire list: $\sum_i c(s_i)pr(s_i)$.

Optimal Binary Search Trees

Let K be a set of fully ordered distinct items. A Binary Search Tree (BST) over K is a binary tree with items in the nodes, such that all the items in the left subtree are smaller

than the root item, and all the items in the right subtree are larger than the root, and both children are also binary search trees. The average search cost in a binary search tree is

$$\frac{sum_{k \in K} depth(k)}{|K|} \quad (1)$$

assuming a uniform cost for all the user decisions on the search process.

Building a binary search tree with the minimal average cost uses the following recursive algorithm:

1. Find the median item $k \in K$ — the item that is in the middle of the sorted list of items.
2. Create a node n with k as its item.
3. Build a tree over the range of items $\{k' \in K, k' < k\}$, and set it as the left child of n.
4. Build a tree over the range of items $\{k' \in K, k' > k\}$, and set it as the right child of n.

The average search cost of this binary search tree is $O(log_2(|K|))$ and the construction time of the tree is $O(|K|log_2(|K|))$.

While an average search cost of $O(log_2(|K|))$ seems low, we can do better using more information — a probability distribution representing the likelihood of accessing each item in the list — $pr(k)$. In this case we can construct an Optimal Binary Search Tree (OBST), that minimizes the expected cost of searching for a item. The optimal split node in a subtree is determined by the following property:

$$c(T_{i,j}) = \frac{\min\limits_{k \in [i,j]} \begin{cases} pr_{i,k-1}(1 + c(T_{i,k-1}))+ \\ pr(k)+ \\ pr_{k+1,j}(1 + c(T_{k+1,j})) \end{cases}}{pr_{i,j}} \quad (2)$$

where $T_{i,j}$ is the optimal binary search trees over items in K raging from i to j, $pr_{i,j} = \sum_{k \in [i,j]} pr(k)$ is the sum of probabilities of items in the range $[i, j]$, and $c(T)$ is the expected cost of the subtree T.

We can hence create a dynamic programming algorithm that computes the optimal binary search tree bottom up in increasing $j - i$ ranges. The straightforward recursive algorithm that at each step checks all the possible roots and subtrees to minimize Equation 2 constructs the tree in $O(|K|^3)$. Knuth [6] has shown that it is sufficient to limit the search for the optimal root in the range $[i, j]$, considering only items between the roots of $T_{i,j-1}$ and $T_{i+1,j}$. Hence, the cost of constructing the tree is reduced to $O(|K|^2)$.

TERNARY SEARCH TREES

In evaluating lexicographic order, users have to focus on a particular character. As mentioned earlier, in some cases the suggested item and the search item share some prefix characters (e.g., "Mar" in the suggested item "Mariah Carey" and the search item "Mark Anthony"). We can therefore allow users to *pin* or lock down prefix characters to constrain the search. Pinning requires us to modify the binary tree data structure, making it into a ternary tree.

Pinning

When interacting with a binary search tree (either BST or OBST), the user actions are limited to accepting the suggested item k, moving up ("before k") or down ("after k") in the list. In the general case of generic items with some ordering over them, it is unclear whether we can do better. We are, however, interested in a special case, where the items are all strings, ordered by lexicographic order.

It is possible, therefore, that the suggested item s' and the search item s share some non-empty prefix. This allows us to reduce the search space to items that have that specific prefix only. As we will later show in our experiments, accepting a prefix is an easier task for a user than moving up or down the list, and can expedite retrieval.

We hence diverge from the standard binary search process by allowing the user to pin the active character [7]. With respect to the UI, pinning can be manifested in many ways. One way is to mark the pinned prefix. For example, instead of displaying the suggestion "Jennifer Lopez", we can display the suggestion "**Jen**nifer Lopez", denoting that the prefix "Jen" is already pinned and that "n" is the active letter. The user can now use the right-key to pin the active letter, use the left-key to un-pin the active letter, move up, down, or use the center-key to accept the suggestion. Pinning the active letter extends the pinned prefix to "Jenn".

Revising the Tree Structure

Since the user now has 4 possible navigational actions (and an "accept" action that ends the search), we will change the binary search tree to a ternary tree. The ternary tree will have three children, where the middle subtree consists of strings that agree on a larger prefix than the prefix shared by all items in this tree.

Formally, let s/i be the prefix of s consisting of i characters. Let $pf_{i,j} = s_i/(\max_l(s_i/l = s_j/l))$ be the maximal prefix that s_i and s_j share. Let:

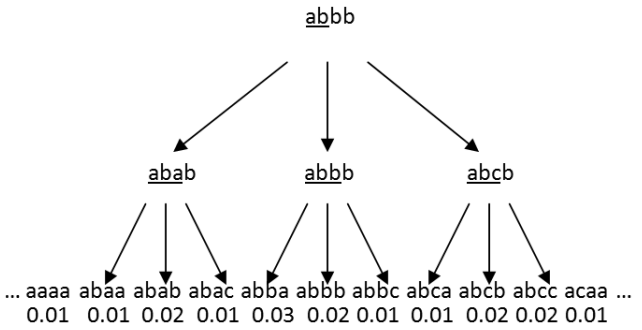
$$p^{min}_{i,j,k} = \min_{l \in [i,k]} (s_l/(|pf_{i,j}| + 1) = s_k/(|pf_{i,j}| + 1)) \quad (3)$$

$$p^{max}_{i,j,k} = \max_{l \in [i,k]} (s_l/(|pf_{i,j}| + 1) = s_k/(|pf_{i,j}| + 1)) \quad (4)$$

be the boundaries of the range of strings around k that agree on an extended prefix. We can now modify Equation 2 to the ternary case (see Example 1):

$$c(T_{i,j}) = \frac{\min\limits_{k \in [i,j]} \begin{cases} pr_{i,p^{min}_{i,j,k}-1}(1 + c(T_{i,p^{min}_{i,j,k}-1}))+ \\ pr(k)+ \\ pr_{p^{min}_{i,j,k},p^{max}_{i,j,k}}(1 + c(T_{p^{min}_{i,j,k},p^{max}_{i,j,k}}))+ \\ pr_{p^{max}_{i,j,k}+1,j}(1 + c(T_{p^{max}_{i,j,k}+1,j})) \end{cases}}{pr_{i,j}} \quad (5)$$

Example 1 Ternary tree construction



Let us consider an example where K is all the strings of length 4 over $\{a, b, c\}$, focusing on the range $s_i = abaa$ to $s_j = abcc$, with item probabilities as shown beneath the strings. The largest prefix shared by s_i and s_j is $pf_{i,j} = ab$. Let us look at $s_k = abbb$ as the candidate for the root of the subtree over the range $[s_i, s_j]$. Selecting s_k expands the prefix from ab to abb. The smallest string with prefix abb is $p^{min}_{i,j,k} = abba$ and the largest string is $p^{max}_{i,j,k} = abbc$. We will therefore need to look at the costs of the three subtrees $T_{abaa,abac}$, $T_{abba,abbc}$, and $T_{abca,abcc}$. The cost of the middle subtree $T_{abba,abbc}$ for example is

$$\begin{aligned} & c(T_{abba,abbc}) \\ & \quad = \frac{\begin{array}{c} pr_{abba} \cdot (1+1) + \\ pr(abbb) + \\ pr_{abbb} \cdot (1+1) + \\ pr_{abbc}(1+1) \end{array}}{pr_{abba,abbc}} \\ & \quad = \frac{0.03 \cdot 2 + 0.02 + 0.02 \cdot 2 + 0.01 \cdot 2}{0.06} \\ & \quad = 2.4 \end{aligned}$$

The costs of the other two subtrees $T_{abaa,abac}$ and $T_{abca,abcc}$ are 2.5 and 2.4 respectively. Therefore, the cost of the entire tree $T_{abaa,abcc}$ is

$$\begin{aligned} & c(T_{abba,abbc}) \\ & \quad = \frac{\begin{array}{c} pr_{abaa,abac} \cdot (c(T_{abaa,abac}) + 1) + \\ pr(abbb) + \\ pr_{abca,abcc} \cdot (c(T_{abba,abbc}) + 1) + \\ pr_{abca,abcc}(c(T_{abca,abcc}) + 1) \end{array}}{pr_{abaa,abcc}} \\ & \quad = \frac{0.04 \cdot 3.5 + 0.02 + 0.06 \cdot 3.4 + 0.05 \cdot 3.4}{0.15} \\ & \quad = 3.56 \end{aligned}$$

The figure above shows the resulting tree. We underline the shared prefix $pf_{i,j}$ at the internal nodes. As we can see, the string $abbb$ is repeated 3 times in this subtree, making it suboptimal. That is, a ternary tree where $abbb$ appears only at its root will have a lower expected cost, but may be confusing for users.

Unfortunately, this definition is not optimal, since the subtree $T_{p^{min}_{i,j,k}, p^{max}_{i,j,k}}$ contains the item k. Therefore, items selected for roots will appear in more than a single node in the tree. An optimal solution can be computed by deleting ancestor items from the subset covered by the subtree, as opposed to continuous ranges of items. However we speculate that users might find it difficult to grasp non-continuous sets. Constructing the ternary tree with the minimal expected cost, where only continuous items subsets are allowed, can be done in $O(2^{|K|})$.

Also, in Equation 5 the root of the tree can be outside the range of the roots of $T_{i,j-1}$ and $T_{i+1,j}$. Therefore, in contrast to the $O(|K|^2)$ construction time for an Optimal Binary Search Tree, the Optimal Ternary Search Tree construction time is $O(|K|^3)$. For long lists, this complexity will result in a very long computation time. For example, constructing a tree with about $21,000$ possible items took more than 10 hours on an Intel Xeon 3 GHz CPU machine with 16 GB of RAM, running Windows 2003 Server.

However, we suggest a number of heuristic methods to build such trees. All of the methods that we describe create the tree top-down, first selecting the root and then constructing its three subtrees. This contrasts with the DP technique that builds the tree bottom-up, starting at the leaves and joining already computed subtrees to create new, larger trees. All the heuristic methods have a average case complexity of $O(|K|log(|K|))$ and a worst case complexity of $O(|K|^2))$. In our experiments with a $21,000$ items dataset, the construction of the trees using the heuristics were accomplished in only a few seconds.

Median: the first heuristic method we suggest ignores the probability distribution and the string structure of items. We select the median item k_m of the sorted list to be the root. We now compute p^{min}_{i,j,k_m} and p^{max}_{i,j,k_m} and construct 3 subtrees over $\{i, ..., p^{min}_{i,j,k_m} - 1\}$ (left subtree), $\{p^{min}_{i,j,k_m}, p^{max}_{i,j,k_m}\}$ (center subtree), and $\{p^{max}_{i,j,k_m} + 1, ..., j\}$ (right subtree). The subtrees are recursively computed using the same technique. This method is an extension of a regular binary search tree to the ternary case.

Popularity: the second heuristic method we suggest is based on the assumption that we should place popular items as close to the root as possible. At each step we select the most popular item k in the current item set K to be placed at the root. The subtrees ranges are computed as in the Median heuristic, recursively employing the popularity approach over each of the subtrees. This method uses the probability distribution, but not the string structure.

Balanced: the third heuristic we suggest attempts to balance the probabilities between the three subtrees. We select k such that

$$max(p_{i,p^{min}_{i,j,k}-1}, p_{p^{min}_{i,j,k}, p^{max}_{i,j,k}}, p_{p^{max}_{i,j,k}+1,j}) - \quad (6)$$

$$min(p_{i,p^{min}_{i,j,k}-1}, p_{p^{min}_{i,j,k}, p^{max}_{i,j,k}}, p_{p^{max}_{i,j,k}+1,j}) \quad (7)$$

is minimized, taking into account the probability distribution

and the string structure of the domain. When multiple strings have the same balance, we break ties by taking the median balanced string.

Example 2 Ternary tree heuristics

... aaaa	abaa	abab	abac	abba	abbb	abbc	abca	abcb	abcc	acaa ...
0.01	0.01	0.02	0.01	0.03	0.02	0.01	0.01	0.02	0.02	0.01

Consider again the set of strings and their probabilities from Example 1, focusing on the range $abaa$ to $abcc$. The median heuristic will select $abbb$ as the root because it has exactly 4 items to the left and 4 items to the right. The popularity heuristic will select $abba$ as the root because it is the string with the maximal probability (0.03). The balanced heuristic can select any of the keys between $abba$ and $abbc$. We therefore use the median tie breaker setting $abbb$ as the root of the tree.

Restricting the Ternary Tree

The optimal ternary tree has a behavior that might seem strange to users. As the tree has the option to "accept" a suggested string, there is no need to present a suggestion more than once. Therefore, the optimal tree will not contain any repeated items. When the user pins the current active letter, thus moving to the middle subtree, the tree suggests the root of that subtree. If items appear only once in the tree, the root of this subtree will contain a different item. Users will therefore observe the following behavior — when accepting the current letter, they implicitly reject the suggestion.

This behavior might be difficult to comprehend, because users may expect that continuing to accept letters will be equivalent to accepting the item. When the correct item is suggested, a user might want to hold down the 'right arrow' key, accepting letters until the entire suggestion is accepted.

Therefore, we restrict the trees such that the item of the middle child must be identical to the item of the parent. The left and right subtrees are still selected optimally given that constraint. This tree structure is not optimal in terms of search cost, but is more appropriate for the user interface we have developed.

The optimization equation (Equation 5) is therefore refined into:

$$c(T_{i,j}) = \frac{\min_{k\in[i,j]} c(T_{i,j,k})}{pr_{i,j}} \tag{8}$$

where:

$$c(T_{i,j,k}) = \begin{cases} pr_{i,p^{min}_{i,j,k}-1}(1 + c(T_{i,p^{min}_{i,j,k}-1}))+ \\ pr(k) + pr_{p^{min}_{i,j,k},p^{max}_{i,j,k}}(1 + c(T_{p^{min}_{i,j,k},p^{max}_{i,j,k},k}))+ \\ pr_{p^{max}_{i,j,k}+1,j}(1 + c(T_{p^{max}_{i,j,k}+1,j})) \end{cases} \tag{9}$$

is the cost of the tree over the range $[i, j]$ that has $i \leq k \leq j$ as its root.

EVALUATION

In this section, we report two types of evaluations: an experimental evaluation of the tree construction methods we have suggested, as well as a usability study of the ternary search interface.

Tree Construction Evaluation

In order to evaluate the expected search costs of the various trees we have suggested above, we conduct an experiment that compares the expected search cost of each method. We used two datasets — a list of movies and a list of artists (singers and bands). We used movies and their popularity from the NetFlix dataset[2], resulting in a list of about $17,000$ items. Artist popularity was computed using ratings from MSN Music[3]. The dataset contained $10,000$ artists, but we enriched the data by adding some suffixes of artist names so that a user can search by surname. For example, for the band "the red hot chilli peppers", we also added the suffixes "red hot chilli peppers", "hot chilli peppers", "chilli peppers" and "peppers". When a suffix appears more than once, such as all the artists whose surname is "Smith", we summed the probabilities of all the original items ending with that suffix. After that, we normalized the probabilities. The list including both complete artist names and suffixes contained about $21,000$ items.

We first compare the trees resulting from the different construction methods. We compute the expected cost of searching for an item over the following 6 methods — an Optimal Binary Search Tree (OBST — Equation 2), the Optimal Ternary Search Tree (OTST — Equation 5), the Restricted Optimal Ternary Search Tree (ROTST — Equation 8), and the three Ternary Search Tree heuristics — Popularity, Balanced, and Median. The expected cost of searching for a string s is the length of the shortest path in the tree from the root to s, multiplied by the probability of s, although our method could use different cost functions. To compute the expected search cost over the entire tree, we sum the expected search costs of all the items.

Table 1 shows the expected cost of searching over items in the lists using the above methods. Even though the Optimal Ternary Search Tree (OTST) is always best, the Balanced heuristic provides a reasonable alternative. The OBST method performs very well compared to the ROTST, mainly because of the constraints we put on the structure of the ROTST. When these constraints are removed, the OTST ternary trees are better. However, as we explain below, the OBST method that does not support pinning of letters, and as such is very unsatisfying for real users. Also, as we later argue the action costs of a binary tree operation are much higher than the action costs of a ternary tree operation.

As expected, the Median method, which ignores string structure and the probabilities, performs the worst. The Popularity method, which takes only the probabilities but not the string structure into account, also does worse than the Balanced heuristic.

[2]www.netflixprize.com
[3]music.msn.com

NetFlix dataset						
Items	OBST	OTST	ROTST	Median	Popularity	Balanced
1000	8.454	7.654	7.943	8.558	8.933	8.22
5000	9.642	8.849	9.109	10.532	10.302	9.761
10000	9.876	9.235	9.557	11.421	10.687	10.268
17770	9.959	9.477	9.767	12.289	10.896	10.534
MSN Music dataset						
Items	OBST	OTST	ROTST	Median	Popularity	Balanced
1000	7.369	6.542	6.815	8.041	7.528	7.154
5000	8.262	7.544	7.741	10.236	8.628	8.191
10000	8.475	7.787	7.988	11.206	8.96	8.541
20501	9.306	8.298	8.574	11.736	9.686	9.201

Table 1. Expected search costs of different methods.

User Study

Although the experiments comparing expected search costs in the previous section demonstrated the analytical advantage of OTST, we conducted a user study in order to assess whether that advantage would be evident in terms of real user experience with a limited UI.

Methodology

In the study, we simulated a limited UI on the desktop — the user could only use 5 keys (up, down, left, right, center) with a single line display. At each step, the user was presented with one random item (item name) from the list of items, and was asked to search for it using one of the three methods below. In all three methods, the user can undo an action by moving left.

Spelling: the user uses the up-down arrows to find the current letter, and clicks right to accept (pin) it. When the prefix matches only a single item in the list, the item is displayed. The user then clicks the center button to accept the item.
Binary search: the user is presented with a suggestion and can either choose to accept the item (center) or specify that the item he is searching for is before (up) or after (down) the suggested item. In the suggested string characters that are already fixed (the prefix that the entire subtree agrees upon) are colored in black, and the rest is colored in red. We used an Optimal Binary Search Tree for providing the suggestions.
Ternary search: the user is displayed with a suggestion and can either accept it (center), accept the current active letter (right), specify that the search item is before (up) or after (down) the suggestion. Again, the accepted prefix is colored in black and the suggestion suffix is colored in red. We used a ternary tree created by the Balanced method for providing the suggestions.

In all of the methods above, the 'left' key was used to retrace and undo an action. Thus, users can fix mistakes that were made during the search process.

Procedure

Prior to the experiment, users went through a training session where they read an explanation of the different search methods and then had to search for a string. During the search, when users selected a wrong action, such as pinning a wrong letter or pressing 'up' instead of 'down', the application notified them of the mistake and they were asked to fix the error. The training session ended only when the search string was successfully found.

After training, users were asked to search for 3 strings per method. A search ended whenever users pressed the 'center' button, whether the string was successfully found or not. During the session, users were not notified if they pressed a wrong key. We kept track of all user actions, including all keystrokes and the elapsed time between keystrokes.

The participants in the study were Microsoft employees who were recruited via an email solicitation. 75 participants conducted 619 searches with the hope of winning a Zune music device.

Results

Quantitative Results

The results from the study were encouraging. As Figure 2 shows, users found the search strings significantly faster when using the ternary search method. Indeed, the average time for a successful search using a ternary tree was 19 seconds, as opposed to 41 seconds for spelling and 47 seconds for binary search.

Method	Ternary	Binary	Spelling
Search time	19.3	46.9 (0.002)	41.3 (0.0008)
Keystrokes	20	31(0.028)	110(0.0001)
Errors	6.20	13.16(0.05)	0.33(0.003)

Table 2. Average measurements in the user study. We report in parentheses pairwise p-values computed using a t-test compared to the Ternary Search.

Table 2 summarizes the measurements that were collected during the user study, including the average number of keystrokes and the average number of errors during a search. The Ternary Tree had the fastest search time and the lowest number of keystrokes. However, the Spelling method had the lowest

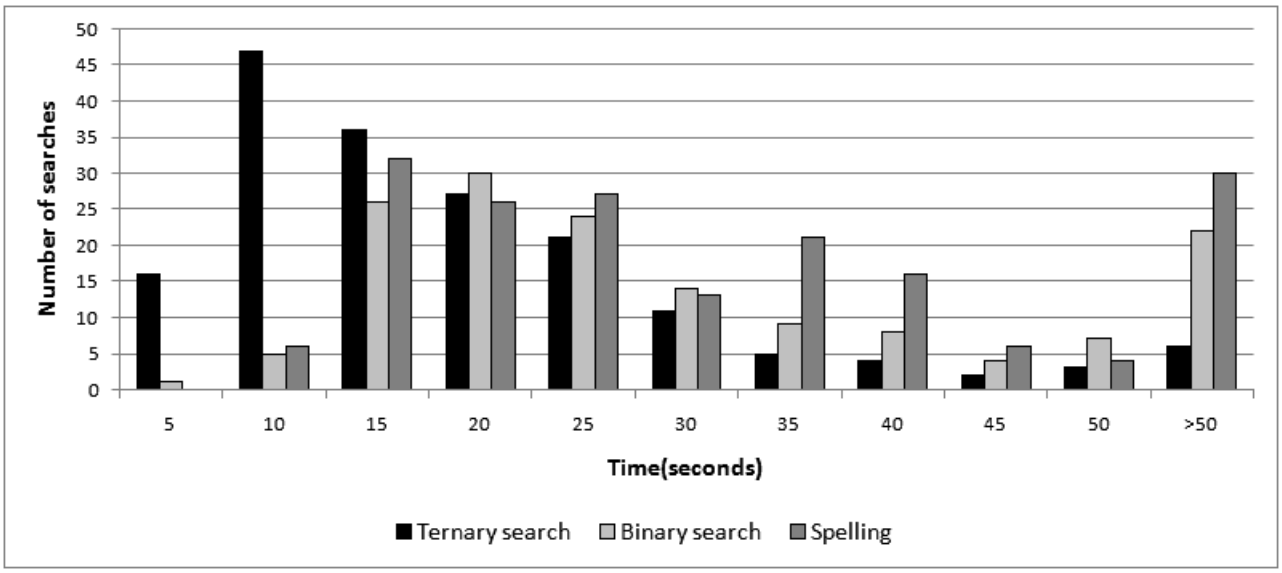


Figure 2. Histogram of the time span of a successful search.

number of errors, due perhaps to its simplicity. For the Spelling method, only pinning the wrong character is considered an error. The user can go up and down many times, passing through the correct character without incrementing the error count.

Qualitative Results

Once all searches were finished, users were asked to fill out a short questionnaire (not all users provided answers). The users answered 3 questions:

1. Which method seemed the fastest for searching?
2. Which method did you prefer?
3. Which method do you think the average user (i.e., non-IT professional) will prefer?

As Table 3 shows, most users thought that the Ternary Search method was the fastest, and said that they preferred this method. Nevertheless, most users thought that the average user would prefer Spelling, perhaps due its simplicity, or perhaps due to familiarity. Which search method actual non-IT professionals will prefer remains as future research.

Method	Ternary	Binary	Spelling
Fastest	39	8	18
Preferred	35	9	21
Average user	18	9	38

Table 3. Answers to questions after the user study.

Measuring Search Costs

Finally, we seek to better understand the cost of searching for a string using our methods. During the construction of the trees we assumed that every decision the user makes has the same cost (1). However, it is possible that different decisions impose different costs. For example, deciding that a letter should be pinned may be an easier task than determining whether the search item appears before or after the suggested item in a lexicographic order. These two decisions may require different cognitive effort. We measured the average time between keystrokes as a proxy for cognitive effort, shown in Table 4. Although it is evident that the average time for choosing to pin the current letter in the Ternary Search method is lower than the average time of going up or down, the difference is not large. However, the average time of going up or down in the Binary Search method is considerably higher than the average time of going up or down in the Ternary Search method. The reason may be due to the fact that in the Ternary Search method, the user effectively needs to look only at the current active letter (the first unpinned letter). In the Binary Search method the user must observe the entire unpinned string to decide whether to go up or down. As deciding to go up or down becomes easier, the reduced effort in deciding to pin becomes less significant. Therefore, we should not expect considerable changes to the tree structure when changing the cost model to reflect average keystroke times.

Method	Ternary	Binary	Spelling
Up	0.892	1.348	0.315
Down	0.970	1.260	0.250
Right	0.849	N/A	0.680
Center	1.475	1.616	1.475

Table 4. Average time for a single keystroke. We computed p-values comparing the average time of up and down to the right action using a pairwise t-test. All the differences were found to be significant at the 0.01 level.

We can observe that the required effort for each keystroke in the Binary Search method is 30% to 50% higher than in the Ternary Search method. If we add 30% to the expected cost of the OBST method in Table 1, the results in favor of the ternary search methods are more impressive.

Finally, it is evident in Table 4 that the Spelling method displayed the shortest average keystroke times. Users achieved this with the simple strategy of pressing and holding the up or down keys until they got near the vicinity of the desired character. While this strategy seemed to have required less cognitive effort, it still resulted in longer search completion times and more keystrokes, as shown in Table 2. Furthermore, the fact that users seemed very comfortable moving up and down in the Spelling method intimates that deciding the proper order of two characters — the active character and the current search item character, is probably a low cost operation. This observation further supports our pinning modifications to the binary search method. We recognize that our average keystroke times may reflect the participant population and hence, consider evaluation of search costs for non-IT professionals an important future work.

RELATED WORK

Mobile search interfaces that require users to make simple lexicographic decisions about a desired query in relation to accepted characters have already hit the marketplace. For example, *Thumbtacts*[4] for the iPhone allows users to search over their contacts "using one hand". The interface features a compass in the center of the screen which can be dragged to one of four directions using a thumb stroke. If the user sees the letter that begins the query they are seeking in one of the four directions, they can drag the compass to that letter. Otherwise, they hit the center of the screen to receive four new choices. After the user drags the compass to a letter, that letter is pinned and the four directions show new choices that all begin with the pinned characters. Thumbtacts does not appear to utilize any kind of figure of merit, such as popularity, to rank-order their choices. It is likely that utilizing our ternary search tree as the underlying data structure would improve the performance of Thumbtacts.

In the research literature, Lehikoinen and Roykkee [7] examined using binary search with pinning in their BinScroll algorithm for mobile search. However, they did not consider how to optimize the method. Therefore, BinScroll relies on a traditional, non-probabilistic, binary search over a sorted list of items, except that pinning constrains the search to elements with the pinned prefix. Similar to our Median heuristic, a move up or down the list from a current suggested item changes the focus to the median element in the new restricted range. In BinScroll, the current suggested item defines a new boundary for that range. Our optimal ternary tree, as well as our Popularity and Balanced heuristics, utilize the additional information over item distribution to reduce the search costs compared to BinScroll.

Although Lehikoinen and Roykkee evaluated BinScroll with a user study showing it to be convenient for users, they did not compare it to other approaches. The closest related work was by Chittaro and De Marco [5], in which they evaluated BinScroll-like methods on a PDA device, splitting the range of items into k regions. Comparing this method to a spelling interface, they concluded that spelling is faster and reduces the number of errors. This is not surprising, because the cognitive effort of understanding which region to choose is considerable. Also, a PDA can display a relatively large virtual keyboard that makes spelling easier. Chittaro and De Marco also did not leverage a probability distribution over items nor did they consider optimizing search cost.

Efficient methods for string search were also previously discussed in other contexts, such as machine translation, where it is critical to quickly find a set of possible suffixes for a translated prefix [11, 8], and in other natural language applications [9]. Our techniques may be useful for researchers in these domains too.

FUTURE WORK

Our evaluation was performed on a desktop, simulating the behavior of a small device. Our next step will be to replicate our results on mobile devices. Furthermore, the above method was suggested for devices that have a single line display, but can be extended to devices that have a small number of display lines, and to devices that have no display at all, such as the Apple Shuffle. In the case of a multi-line display, where multiple suggestions can be shown, it may be possible to direct the user towards the right action by displaying, for instance, suggestions from the two subtrees. In the case of no display, the suggestion can be spoken to the user to offer "eyes-free" interaction with a potentially even smaller device.

Our methods use a static tree that needs to be reconstructed every time the item distribution changes. However, we can maintain statistics over user searches and reconstruct the tree after sufficient statistics are gathered. Other data structures automatically adjust to changes in user preferences. For example, splay trees [10] rearrange their structure so that recently searched items are closer to the root. We can hence originally construct a tree using the methods we suggest, and then use splaying to rearrange the tree given the items the user searches for. The resulting personalized search tree is incrementally improved to better fit the user history. Splay trees are binary and a variation conforming to our restrictive ternary tree structure needs to be developed.

CONCLUSION

In this paper we suggested a method for searching for items within a large catalog of items ranked by probabilities, using a limited user interface. We explained how an Optimal Binary Search Tree can be augmented with character pinning in the form of a Ternary Search Tree. As the cost for creating an optimal ternary search tree is unrealistic for large indexes, we suggested several heuristics that create good ternary trees.

Finally, we evaluated the Ternary Search method both analytically with respect to expected search cost and empirically with respect to a user study. The user study confirmed that minimizing expected search cost can result in reducing both real search time and keystrokes. Users also preferred our method.

[4]http://www.crimsonresearch.net/thumbtacts/

REFERENCES

1. R. S. Amant, T. E. Horton, and F. E. Ritter. Model-based evaluation of cell phone menu interaction. In *Conference on Human Factors in Computing Systems archive*, pages 343 – 350, 2004.
2. R. Bellman. *Dynamic Programming*. 1957.
3. M. Breene. US portable music device forecast, 2007 to 2012. Technical report, Jupiter Research, 2007.
4. D. Card. US portable music device forecast, 2006 to 2011. Technical report, Jupiter Research, 2006.
5. L. Chittaro and L. D. Marco. Evaluating the effectiveness of ”effective view navigation” for very long ordered lists on mobile devices. In *Interact 2005: 10th IFIP International Conference on Human-Computer Interaction*, pages 482–495, 2005.
6. D. E. Knuth. Optimum binary search trees. *Acta Informatica*, 1(1):14–25, 1971.
7. J. Lehikoinen and M. Roykkee. An empirical and theoretical evaluation of binscroll: A rapid selection technique for alphanumeric lists. *Personal Ubiquitous Computing*, 6(2):141–150, 2002.
8. A. Lopez. Hierarchical phrase-based translation with suffix arrays. In *Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, page 976985, 2007.
9. T. Matsumoto, D. M. W. Powers, and G.Jarrad. Application of search algorithms to natural language processi. In *Australasian Language Technology Workshop*, 2003.
10. D. Sleator and R. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985.
11. R. Zens and H. Ney. Efficient search for interactive statistical machine translation. In *In EACL 03: Proceedings of the tenth conference on European chapter of the Association for Computational Linguistics*, pages 387–393, 2003.