Approximate Binary Trees in External Memory Models

by

Oliver Grant

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2016

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

We examine the classic binary search tree problem of Knuth [**?**]. First, we quickly re-examine a solution of Güttler, Melhorn and Schneider [**?**] which was shown to have a worst case bound of $c * H + 2$ where $c \geq 1/H(1/3, 2/3) \approx 1.08$. We give a new worst case bound on the heuristic of $H + 4$. In the remainder of the work, we examine the problem under various models of external memory. First, we use the Hierarchical Memory Model (HMM) of Aggarwal et al. [**?**] and propose several approximate solutions. We find that the expected cost of search is at most $\frac{W_{HMM}(P)}{\lceil lg(s_{min})\rceil} * (H+1)$ where $W_{HMM}(P)$ is a well-defined function of the probability distribution, $s_{min} = min(min_{i\in\{1,\ldots,n\}}(p_i), min_{j\in\{0,\ldots,n\}}(q_j)))$, and H is the entropy of the distribution. Using this, we improve a bound given in Thite's 2001 thesis for the HMM_2 model in the approximate setting. We also examine the problem in the Hierarchical Memory with Block Transfer Model [**?**] and find approximate solutions. Similarly, we find the expected cost is at most $\frac{W_{HMBTM}(P)}{\lceil lg(s_{min})\rceil} * (H + 1)$ where $W_{HMBTM}(P)$ is a well-defined function of the probability distribution and s_{min} and H are as before.

Acknowledgements

I would like to thank all the little people who made this possible.

Dedication

This is dedicated to the one I love.

Table of Contents

List of Tables viii

List of Figures ix

1 Introduction 1

1.1 Binary Search Trees . . . 1
1.2 The Optimum Binary Search Tree Problem . . . 1
1.3 Why Study Binary Search Trees . . . 2
1.4 Overview . . . 2

2 Background and Related Work 4

2.1 Preliminaries . . . 4
2.2 Binary Search Trees . . . 4
2.3 Alphabetic Trees . . . 5
2.4 Multiway Trees . . . 6
2.5 Memory Models . . . 7

3 An Improved Bound for the Modified Minimum Entropy Heuristic 8

3.1 The Modified Minimum Entropy Heuristic . . . 8
3.2 MME is within 4 of Entropy . . . 8

4 Approximate Binary Search in the Hierarchical Memory Model **11**

4.1 The Hierarchical Memory Model 11

4.2 Thite's Optimum Binary Search Trees on the HMM Model 12

4.3 Efficient Near-Optimal Multiway Trees of Bose and Douïeb 12

4.4 Algorithm ApproxMWPaging 13

4.5 Expected Cost ApproxMWPaging 15

4.6 Approximate Binary Search Trees of De Prisco and De Santis with Extensions by Bose and Douïeb 18

4.7 Algorithm ApproxBSTPaging 19

4.8 Expected Cost ApproxBSTPaging 20

4.9 Improvements over Thite in the HMM_2 Model 21

5 Approximate Binary Search in the Hierarchical Memory with Block Transfer Model **24**

5.1 Hierarchical Memory with Block Transfer Model 24

5.2 ApproxMWPaging with BT 25

5.3 ApproxMWPaging with BT Running Time 27

5.4 Search with ApproxMWPaging with BT 30

5.5 Expected Cost ApproxMWPaging with BT 31

5.6 ApproxBSTPaging with BT 31

5.7 Expected Cost ApproxBSTPaging with BT 31

6 Conclusion and Open Problems **32**

6.1 Discussion 32

6.2 Conclusion 32

APPENDICES **33**

A PDF Plots From Matlab **34**

References **35**

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Binary Search Trees

A binary search tree is simple structure used to store key-value pairs. It was invented in the late 1950s and early 1960s and is generally attributed to the combined efforts of Windley, Booth, Colin and Hibbard [**?**] [**?**] [**?**]. In general, the trees allow for quick binary searches through the data in search of a specific key. Each node has a key over which there is a total-ordering (a number, a string, etc.), as well as some value (generally the important information). Each tree node has at most two children generally labelled as the *left* and *right*. All nodes in the subtree of the *left* child of a specific node have a key strictly less than the key of the node in question. Similarly, nodes in the subtree of the *right* child of a specific node have a key strictly greater than the key of the node in question. A pointer is typically stored to a root node. Search begins from this root node and is done by recursively searching in either the *left* or *right* child of root node, or stopping if the root being searched has the correct key.

1.2 The Optimum Binary Search Tree Problem

Knuth first proposed the optimum binary search tree problem in 1971 [**?**]. We are given a set of n words $B_1, B_2, ..., B_n$ and $2n + 1$ frequencies, $p_1, p_2, ..., p_n$, $q_0, q_1, ..., q_n$ representing the probabilities of searching for each given word and the probabilities of searching for strings between (and outside of) these words. We have that $q_0 + \sum_{i=1}^{n} p_i + q_i = 1$. We also

assume that without loss of generality $q_i + p_i + q_{i+1} \neq 0$ for any i. The words (and gaps between) are used as keys and the lexicographical ordering of them provides our order over the keys. Our goal is to construct a binary tree such that the expected cost of search is minimized. The names make up the leaves of the tree while, gaps make up the internal nodes. The weighted path length of the tree is:
$P = \sum_{i=1}^{n} p_i(b_i + 1) + \sum_{j=1}^{n} q_j(a_j)$
Where b_i and a_j represent the depth of nodes representing the i^{th} word and j^{th} gap respectively. The optimal solution of Knuth requires $O(n^2)$ time, and $O(n^2)$ space. This solution is both time and space intensive. We will later examine an approximate solution to this problem of Güttler, Mehlhorn and Schneider which uses $O(n^2)$ time but $O(n)$ space and improve its worst-case bound. However, these problems were examined under the RAM model which is an inadequate model for many situations. We examine the problem in more realistic models and look at approximate solutions under these settings.

1.3 Why Study Binary Search Trees

Binary search trees are ubiquitous throughout computer science with numerous applications. The basic binary search tree has been built upon in many ways. AVL trees (named after creators AdelsonVelskii and Landis) were the first form of self-balancing binary search trees introduced [**?**]. The trees invented by the pair in 1963 and maintain a height of $O(lg(n))$ where n is the number of nodes in the tree during insertions and deletions (both of which take $O(lg(n))$ time). Improved self-balancing binary search trees followed in the form of red-black trees by Bayer in 1972 and splay trees by Sleator and Tarjan in 1985 [**?**] [**?**]. Tango trees were invented in 2007 by Demaine et al. and provided the first $O(lglgn)$ competitive binary tree [**?**]. Here, $O(lglgn)$ competitive means that the tango tree does at most $O(lglgn)$ times more work (pointer movements and rotations) than an optimal offline tree. B trees are among the most commonly used binary tree variant and were invented in 1970 by Bayer and McCeight [**?**].

Binary space partition Binary Tries in routers Hash Trees Heaps - PQ Huffman Code - compression Syntax tree expression evaluation

1.4 Overview

In Chapter 2 I review previous work done in the areas of binary search trees, multiway trees, alphabetic trees and various models of external memory. In Chapter 3, I re-examine

the Modified Minimum Entropy (MME) heuristic of Güttler, Mehlhorn and Schneider [**?**]. This is a $O(n)$ time algorithm for approximating the optimum binary search tree problem in the RAM model. The method worked very well in practice, and the group had great experimental results, but unfortunately they could not bound the worst case expected cost of the as they would have hoped. While simpler solutions like the *Min-max* and *Weight Balanced* techniques of Bayer [**?**] had worst case costs of at most $H + 2$, the trio's MME technique had a worst case expected search cost of $c*H+2$ where $c \approx 1.08$. I provide a new argument of its worst case expected search cost and show that it is within a constant of entropy: at worst $H + 4$. In Chapter 4, I move on to external memory models, examining the optimum binary search tree problem under the hierarchical memory model as was done in Thite's thesis [**?**]. I provide TWO? $O(n*lg(m_1))$ time algorithm which has a worst case expected cost of TODO FIX $C \leq (\lceil lg(m_1) \rceil * (\lfloor log_{m_1}(\frac{2}{min_{p,q}}) \rfloor + 1)) * W * (H + 1)$ WHERE TODO. The solution provided also gives a direct improvement over the solution Thite provided in the same work for HMM_2. In Chapter 5, I extend my solutions to examine a more realistic model for external memory, the hierarchical memory with block transfers model (HMBTM). I use similar algorithms which run in TODO and give worst case expected search costs of TODO. In Chapter 6 I summarize my findings and discuss several problems which remain open.

Chapter 2

Background and Related Work

2.1 Preliminaries

I MAY NOT EVEN NEED THIS SECTION (ESPECIALLY NOT WHAT IS CURRENTLY IN IT).

$H = \sum_{i=1}^{n} p_i * log_2(1/p_i) + \sum_{j=0}^{n} q_i * log_2(1/q_j)$ is the entropy of the probability distribution. We will also use $H(x_1, x_2, ..., x_n)$ to describe the entropy of various other probability distributions. We let $E_t = H(P_{L_t}(i), p_i, P_{R_t}(i))$ be the local entropy of a sub tree t rooted at B_i. $P_{L_t}(p_i) = \frac{P_L(p_i)}{p_t}$ and $P_{R_t}(p_i) = \frac{P_R(p_i)}{p_t}$ describe the normalized probabilities of searching for a forward before or after (respectively) word B_i. Similarly, $P_{L_t}(q_j)$ and $P_{R_t}(q_j)$ describe the normalized probabilities of searching for a forward before or after (respectively) the gap between words B_j and B_{j+1}.

2.2 Binary Search Trees

After Knuth's initial examination of the optimum binary search tree problem in 1971 [**?**], several others have examined the approximate version of the problem. The optimal solution requires $O(n^2)$ time and space which is too costly in many situations. While unable to bound an approximate algorithm within a constant of the optimal solution, many authors have been able to bound the cost based on the entropy of the distribution of probabilities H. In 1975 Bayer showed that $H - log_2 H - (log_2 e - 1) \leq C_{Opt}$, $C_{Opt} \leq C_{WB} H + 2$ and $C_{Opt} \leq C_{MM} \leq H + 2$ where C_{Opt}, C_{WB}, and C_{MM} are optimal, weight-balanced and min-max costs [**?**]. Weight-balanced and min-max costs heuristics are greedy and require both

$O(n)$ time and $O(n)$ space to run. In 1980, Güttler, Mehlhorn AND Schneider presented a new heuristic, the modified minimum entropy (MME) heuristic [**?**] which built upon the ideas of Horibe [**?**]. Güttler, Mehlhorn AND Schneider gave empircal evidence that the heuristic out-performed others [**?**]. While the heuristic took $O(n^2)$ time, it only required $O(n)$ space, a huge saving over the optimal solution. However, they were unable to prove that $C_{MME} \leq H + 2$ (like previous heuristics of weight-balanced and min-max) and settled with $C_{MME} \leq c_1 * H + 2$
where $c_1 = 1/H(1/3, 2/3) \approx 1.08$. In 1993, De Prisco and De Santis presented a new heuristic for constructing a near-optimum binary search tree [**?**]. The method is more specifically explained later in this work (Section TODO) and has an upper bounded cost of at most $H + 1 - q_0 - q_n + q_{max}$ where q_{max} is the maximum weight leaf node. This method was later updated by and Douïeb to have a worst case cost of [**?**]

$$\frac{H}{lg(2k-1)} \leq P_{OPT} \leq P_T \leq \frac{H}{lgk} + 1 + \sum_{i=0}^{n} q_i - q_0 - q_n - \sum_{i=0}^{m} q_{rank[i]}$$

Here, H is the entropy of the probability distribution, P_{OPT} is the average path-length in the optimal tree, P_T is the average path length of the tree built using their algorithm and $m = max(n - 3P, P) - 1 \geq \frac{n}{4} - 1$. P is the number of increasing or decreasing sequences in a left-to-right read of the access probabilities of the leaves. Moreover, $q_{rank[i]}$ is the i^{th} smallest access probability among all leaves except q_0 and q_n.

2.3 Alphabetic Trees

Given a set of n keys with various probabilities, the problem is to build a binary search tree where every internal node has two children, leaves have no children, and the n keys described are the leaves with minimum expected search cost. Here, the expected search cost is $\sum p_i * l_i *$ where p_i is the probability of searching for leaf/key i and l_i is the leaf's level in the tree. The alphabetic ordering of the leaves must be maintained. This is the same as the binary search tree problem with all internal node weights zero.

In 1952, Huffman famously developed the Huffman tree, which solved the same problem without a left-to-right ordering constraint on leaves [**?**]. Gilbert and Moore examined the problem with the added alphabetic constraint and developed a $O(n^3)$ algorithm which solved the problem optimally [**?**]. Hu and Tucker gave a $O(n^2)$ time and space algorithm in 1971 [**?**] which was improved by Knuth to take only $O(nlgn)$ time and $O(n)$ space in 1973 [**?**]. The original proof of Hu and Tucker was extremely complicated, but was fortunately

later simplified by Hu [?] and Hu et al. [?]. Garsia and Wachs gave an independent $O(nlgn)$ time, $O(n)$ space algorithm in 1977 [?]. This was shown to be equivalent to the Hu and Tucker algorithm in 1982 by Hu [?] and also went through a proof simplification [?] by Kingston in 1988.

In 1991, Yeung proposed an approximate solution which solved the problem in $O(n)$ time and space [?]. The algorithm produced a tree with worst case cost $H + 2 - q_1 - q_n$. This was later imporved by De Prisco and De Santis who created an $O(n)$ time algorithm which had a worst case cost of $H + 1 - q_0 - q_n + q_{max}$ [?]. The method was improved one more time by Bose and Douïeb who improved upon Yeung's method by decreasing the bound by $\sum_{i=0}^{m} q_{rank[i]}$ where $m = max(n - 3P, P) - 1 \geq \frac{n}{4} - 1$, P is the number of increasing or decreasing sequences in a left-to-right read of the access probabilities of the leaves and $q_{rank[i]}$ is the i^{th} smallest access probability among all leaves except q_0 and q_n [?]. De Prisco and De Santis used Yeung's method within their own, so this improvement to Yeung's method gave an overall tighter bound of $H + 1 - q_0 - q_n + q_{max} - \sum_{i=0}^{m} q_{rank[i]}$. CHECK THIS OUT TODO

2.4 Multiway Trees

This static k-ary or multiway search tree problem is similar to optimum binary search tree problem with the added constraint that up to k keys can be placed into a single node, and cost of search within a node is constant. Multiway search trees maintain an ordering property similar to that of traditional binary search trees. Each key in every page in the subtree rooted at a specific location in a page g (i.e. between two keys k and l) must have its key lie between k and l. Each internal node of the k-ary tree contains at least one and at most $k - 1$ keys while a leaf node contains no keys. Successful searches end in an internal while unsuccessful searches end in one of the $n + 1$ leaves of the tree. The cost of search is the average path depth is defined as :

$$\sum_{i=1}^{n} p_i(d_T(x_i) + 1) + \sum_{j=0}^{n} q_j(d_T(x_{i-1}, x_i))$$

where x_i's represent successful search keys, pairs (x_{i-1}, x_i) represent unsuccessful search "keys" and $d_T(x_i)$ or $d_T(x_{i-1}, x_i)$ represent the depth of a specific successful or unsuccessful search respectively.

Vishnavi et al. [?], and Gotlieb [?] in 1980 and 1981 respectively independently solved the problem optimally in $O(k * n^3)$ time. In a slightly modified B-tree model (every leaf

has same depth, every internal node is at least half full), Becker's 1994 work gave a $O(kn^{\alpha})$ time algorithm where $\alpha = 2 + log_k 2$ [**?**]. Moreover, in 1997 Becker propsed an $O(Dkn)$ time algorithm where D is the height of the resulting tree[**?**]. The algorithm did not produce an optimal tree but was thought to be empirically close despite having no strong upper bound. In 2009, Bose and Douïeb gave both an upper and lower bound on the optimal search tree in terms of the entropy of the probability distribution as well as an $O(n)$ time algorithm to build a near-optimal tree [**?**]. There bounds of:

$$\frac{H}{lg(2k-1)} \leq P_{OPT} \leq P_T \leq \frac{H}{lgk} + 1 + \sum_{i=0}^{n} q_i - q_0 - q_n - \sum_{i=0}^{m} q_{rank[i]}$$

will be discussed in more detail in section BLAH TODO of this paper.

2.5 Memory Models

HMM HMM BT Other Stuff Cache Oblivious Look at what Thite wrote about TODO - LOOK @ SURVEY ON MEMORY MODELS

Chapter 3

An Improved Bound for the Modified Minimum Entropy Heuristic

3.1 The Modified Minimum Entropy Heuristic

As described in [**?**], the heuristic greedily chooses the word B_i as the root such that $H(P_{L_t}(p_i), p_i/p_t, P_{R_t}(p_i))$ is maximized (local information gain) where $P_{L_t}(p_i)$ and $P_{R_t}(p_i)$ are the probabilities of searching for a word to the left and to the right of p_i (normalized for the sub tree in question whose total probability is p_t). There are two exceptions to this rule. Firstly, if there exists p_i such that $\frac{p_i}{p_t} > max(P_{L_t}(p_i), P_{R_t}(p_i))$ we always select B_i as the root. Moreover, if there exists q_j such that $\frac{q_j}{p_t} > max(P_{L_t}(q_j), P_{R_t}(q_j))$ then we select the root from among B_j and B_{j+1}. B_j is chosen if $P_{L_t}(q_j) > P_{R_t}(q_j)$ and B_{j+1} is chosen otherwise. Since it takes linear time and constant space to find the all of the optimal local entropy splits in each level of the tree, our algorithm takes $O(n^2)$ time and linear space.

3.2 MME is within 4 of Entropy

Lemma 3.2.1. *When using the MME heuristic to choose the root of a binary search, one of the following three cases must occur:*
1) $E_t \geq 1 - 2\frac{p_r}{p_t}$
2) There exists q_i *such that* $\frac{q_i}{p_t} > max(P_{L_t}(q_i), P_{R_t}(q_i))$

3) $max(P_L(p_r), P_R(p_r)) < \frac{4}{5}p_t$

Proof. Suppose there exists some p_i such that $\frac{p_i}{p_t} > max(P_{L_t}(p_i), P_{R_t}(p_i))$. By the MME heuristic, it must be selected as the root and thus $p_r = p_i$. As shown in [**?**] $H(x, 1-x) \geq 2x$ when $x < 1/2$. Thus we have:
$E_t \geq H(max(P_{L_t}(p_i), P_{R_t}(p_i)), 1 - max(P_{L_t}(p_i), P_{R_t}(p_i))$
$\geq 2 * max(P_{L_t}(p_i), P_{R_t}(p_i))$
$\geq 1 - \frac{p_i}{p_t}$
$\geq 1 - 2\frac{p_i}{p_t} \geq 1 - 2\frac{p_r}{p_t}$
as required.

If we have a p_i across the middle of the set (i.e. $P_{L_t}(p_i) < \frac{1}{2}p_t$ and $P_{R_t}(p_i) < \frac{1}{2}p_t$) then we have:
$E_t \geq H(x, y, 1 - x - y)$ where $0 < x < 0.5$ and $0 < y < 0.5$ and we know that
$H(x, y, 1 - x - y) \geq H(1/2, 1/2) = 1$ in this case. Thus, $E_t \geq 1 - 2p_r$ as required.

Otherwise, we must have a q_i spanning the middle of the data set (i.e. $P_{L_t}(q_i) < \frac{1}{2}p_t$ and $P_{R_t}(q_i) < \frac{1}{2}p_t$). Now supposed that case 2) does not occur: there does not exist a q_i such that $\frac{q_i}{p_t} > max(P_{L_t}(q_i), P_{R_t}(q_i))$. Thus, we have that
$max(P_{L_t}(p_i), P_{R_t}(p_i)) \geq min(P_{L_t}(p_i), P_{R_t}(p_i)$ and
$max(P_{L_t}(p_i), P_{R_t}(p_i)) \geq q_i$
Thus, $max(P_{L_t}(p_i), P_{R_t}(p_i)) \geq 1/3$ and by our assumption $max(P_{L_t}(p_i), P_{R_t}(p_i)) < 1/2$.
So, as in the proof of table 3 (5.3) in [**?**]
$E_t \geq H(1/3, 2/3) \approx 0.92$.
Since we do not have case 1), we know that
$E_t < 1 - 2\frac{p_r}{q_t}$
$\implies \frac{p_r}{q_t} < \frac{1-H(1/3,2/3)}{2} \approx 0.04$
Suppose that $max(P_L(p_r), P_R(p_r)) \geq \frac{4}{5}p_t$ then we have
$E_t \leq H(\frac{4}{5}, \frac{1-H(1/3,2/3)}{2}, \frac{1}{5} - \frac{1-H(1/3,2/3)}{2}) \approx 0.87 < H(1/3, 2/3) \geq E_t$
a contradiction.
Thus, if we do not have case 1) or 2), we must have case 3).

□

Theorem 3.2.2. $C_{MME} \leq H + 4$

Proof. This is very similar to the proof of theorem 4.4 in [**?**]. We will first bound each E_t on a case by case basis using the cases of Lemma 1.
If case 1 occurs, we obviously have that
$E_t \geq 1 - 2\frac{p_r}{p_t}$
Note that this can only happen once for each word as a word can only be the root once.

Let q_m be the middle gap when case two or three occurs. When case 2 occurs we have that
$E_t \geq H(\frac{1}{2} - \frac{1}{2}\frac{q_m}{p_t}, \frac{1}{2} + \frac{1}{2}\frac{q_m}{p_t}) \geq 2(\frac{1}{2} - \frac{1}{2}\frac{q_m}{p_t}) = 1 - \frac{q_m}{p_t}$.
Note that this can only happen twice for each gap (by the definition of the MME heuristic).

When case 3 occurs we have that
$E_t \geq H(\frac{1}{2} - \frac{1}{2}\frac{q_m}{p_t}, \frac{1}{2} + \frac{1}{2}\frac{q_m}{p_t}) \geq 1 - \frac{4}{5}(\frac{q_m}{p_t})^2$ when $\frac{q_m}{p_t} < \frac{1}{2}$.

As in [**?**] we define a b_t for each subtree as follows:
let $b_t = 2p_r$ for case 1.
let $b_t = 2q_m$ for case 2.
let $b_t = \frac{q_m^2}{p_t}$ for case 3.
Thus, we have [**?**]
$H = \sum_{t \in S_T} P_t E_t \geq \sum P_t - \sum b_t = C - \sum b_t$
$\implies C \leq H + \sum b_t$
Where S_T is the set of all subtrees our our tree.

As mentioned above, cases 1 and 2 can only occur once and twice respectively. Case 3 however, can occur many times, but each time it occurs, $\frac{q_m}{p_t}$ must decrease by a factor of 5/4. Let S_m be the set of all subtrees t for which q_m is the middle gap. We have that
$C \leq H + \sum b_t = H + 2\sum_{r=1}^{n} p_r + 2\sum_{m=0}^{n} q_m + \sum_{m=0}^{n} \sum_{t \in S_m} \frac{4}{5}\frac{q_m^2}{p_t}$
$\implies C \leq= H + 2 + \sum_{m=0}^{n} \frac{4}{5} q_m \sum_{x=0}^{\infty} \frac{1}{2} * (\frac{4}{5})^x$
$\implies C \leq= H + 2 + \sum_{m=0}^{n} \frac{4}{5} q_m \frac{5}{2}$
$\implies C \leq= H + 4$

□

Chapter 4

Approximate Binary Search in the Hierarchical Memory Model

4.1 The Hierarchical Memory Model

The Hierarchical Memory Model (HMM) was proposed in 1987 by Aggarwal et al. as an alternative to the classic RAM model [**?**]. It was intended to better model the multiple levels of the memory hierarchy. The model has an unlimited number of registers, $R_1, R_2, ...$ each with its own location in memory (positive integer). In the first version of the model, accessing a register at memory location x_i takes $\lceil lg(x_i) \rceil$ time. Thus, computing $f(a_1, a_2, ..., a_n)$ takes time $\sum_{i=1}^{n} \lceil lg(location(a_i)) \rceil$. The original paper also considered arbitrary cost functions $f(x)$. We will use the cost function as was explained in Thite's thesis [**?**]. Here, $\mu(a)$ is the cost of accessing memory location a. We have a series of memory sizes $m_1, m_2, ..., m_h$ where m_l has infinite size each with an associated cost $c_1, c_2, ..., c_h$. We assume that $c_1 < c_2 < ... < c_h$.

$$\mu(a) = c_i if \sum_{j=1}^{i-1} m_j < a \leq \sum_{j=1}^{i} m_j.$$

While Thite notes that typical memory hierarchies have increasing sizes for slower memory levels, we explicitly assume that successive memory sizes divide one another evenly:

$$m_1 < m_2 < ... < m_h \text{ and}$$

$$\forall i \in \{1, 2, ..., h\},\ m_i \mid m_{i+1}.$$

4.2 Thite's Optimum Binary Search Trees on the HMM Model

Thite's thesis provided solutions to several problems in the HMM and a related model [**?**]. He first provided an optimal solution to the following problem (known as **Problem 5** in the work):

Problem [Optimum BST Under HMM]. Suppose we are given a set of n ordered keys $x_1, x_2, ..., x_2$ with associated probabilities of search $p_1, p_2, ..., p_n$, as well as $n+1$ ranges $(-\infty, x_1), (x_1, x_2), ..., (x_{n-1}, x_n), (x_n, \infty)$ with associated probabilities of search $q_0, q_1, ..., q_n$. The problem is to construct a binary search tree T over the set of keys and compute a memory assignment function $\phi : V(T) \rightarrow 1, 2, ..., n$ that assigns nodes of T to memory locations such that the expected cost of a search is minimized under the HMM model.

He provided three separate optimum solutions; **Parts**, **Trunks**, and **Split**. These algorithms have various use cases, running in times $O(\frac{2^{h-1}}{(h-1)!} * n^{2h+1})$, $O(\frac{2^{n-m_h} * (n-m_h+h)^{n-m_h} * n^3}{(h-2)!})$ and $O(2^n)$ respectively. In the following sections, I will provide an approximate solution to this problem that runs in time $O(n * lg(m_1))$ and an upper bound on its expected search cost.

Thite also considered the same problem under the related HMM_2 model. This model assumes there are simply two levels of memory of size m_1 and m_2 with costs of access c_1 and c_2 where $c_1 < c_2$. Thite provided an optimal solution to this problem (named **TwoLevel**) which ran in time $O(n^5)$, $o(n^5)$ if $m_1 \in o(n)$, and $O(n^4)$ if $m_1 \in O(1)$. He also gave an $O(nlgn)$ time approximate solution with an upper bounded expected search cost of $c_2(H + 1)$. The solution I provide which approximates the optimum tree under the HMM model also provides a strict improvement over Thite's approximate algorithm in both running time and expected cost under the HMM_2 model.

4.3 Efficient Near-Optimal Multiway Trees of Bose and Douïeb

In 2009, Bose and Douïeb's 2009 work provided a new construction method with linear running time (independent of the size of a node in tree) with the best expected cost to

date [**?**]. The group was able to prove that:

$$\frac{H}{lg(2k-1)} \leq P_{OPT} \leq P_T \leq \frac{H}{lgk} + 1 + \sum_{i=0}^{n} q_i - q_0 - q_n - \sum_{i=0}^{m} q_{rank[i]}$$

Here, H is the entropy of the probability distribution, P_{OPT} is the average path-length in the optimal tree, P_T is the average path length of the tree built using their algorithm and $m = max(n - 3P, P) - 1 \geq \frac{n}{4} - 1$. P is the number of increasing or decreasing sequences in a left-to-right read of the access probabilities of the leaves. Moreover, $q_{rank[i]}$ is the i^{th} smallest access probability among all leaves except q_0 and q_n.
As described in the subsection TODO, we are given n ordered keys with weights $p_0, ..., p_n$ as well as $n + 1$ weights of unsuccessful searches $q_0, ..., q_n$. We often refer to "keys" representing the gaps as (x_{i-1}, x_i) to be mean the "key" associated with a search between key x_{i-1} and x_i.

The algorithm occurs in three steps. First, a new distribution is created by distributing all leaf weight to internal nodes. The peaks and valleys of the probability distribution of the leaf weights is used to do this assignment. After this, the new probability distribution is made into a k-ary tree using a greedy recursive algorithm. The algorithm essentially chooses the $l \leq k$ elements to go in the root node such that each child's subtree will have probability of access of at most $1/k$. They then reattach leaves to this internal key tree they have created. Their algorithm's design allows them to bound the depth of keys and leaves which ultimately allows them to achieve the bounds they have described.

4.4 Algorithm ApproxMWPaging

First, we need a lemma about converting a multiway search tree to a binary search tree. For the sake of clarity, we will call what are typically known as *nodes* of the multiway tree *pages*. This represents how various items of our search tree will fit onto pages. We maintain the notion of calling individual items *keys*.

Lemma 4.4.1. *Given a multiway tree T' with page size k and n keys, where keys are associated with a probability distribution of successful and unsuccessful searches as in Knuth's original optimum binary search tree problem, we can create a BST T where each key in a given page g of T' forms a connected component in T in $O(nlg(k))$ time.*

Proof. For each page g, we sort its keys and create a complete BST B over the keys. We create an ordering over all potential locations where keys could be added to the tree from

left to right. All keys in all descendant pages of a page in a specific subtree rooted at a child of g will lie in a specific range. There are at most $k+1$ of these ranges (since our page has at most k keys). These ranges will precisely correspond to the at most $k+1$ locations where a new child key could be added to B. We note that we cannot add new children to keys which correspond to unsuccessful searches. We order these locations from left to right and attach root keys from the newly created BST's of each of the ordered (left to right) children of g. These will all be valid connections since each child of g has keys in these correct ranges, and combining BST's in this fashion produces a valid BST. We perform a sort of $O(k)$ items (in time $O(klg(k))$) $O(n/k)$ times, and make $O(n)$ new parent child connections, giving us total time $O(nlg(k)$. □

In order to approximately solve the optimum BST problem under the HMM model, we will do the following:

1) First, we create a Multiway Tree T' using the algorithm of Bose and Douïeb. This takes $O(n)$ time with our node size equal to m_1, the smallest level of our memory hierarchy [**?**].

2) Inside each page (node of the multiway tree), we create a balanced binary search tree (ignoring weights). We call each of these T'_k for $k \in 1, ..., \lceil n/m_1 \rceil$. This takes $O(m_1)$ time per page, of which there are at most $O(n/m_1)$ giving us $O(n)$ time total.

3) In order to make this into a proper binary search tree, we must connect the $O(n/m_1)$ BST's we have made as described in the previous Lemma. This takes $O(nlg(m_1))$ time.

4) We pack things into memory in a breadth first search order in T starting from the root. This takes $O(n)$ time.

THIS IS THE OLD WAY. I THINK ITS NEEDED FOR TH BETTER MODEL BUT NOT FOR THIS. We pack things into memory in a breadth first search order in T' starting from the root. We do not leave gaps if pages are not full. The individual items of the pages of size at most m_1 are stored in breadth first search order based on the trees made in step 2. This takes $O(n/m_1)$ for the breadth first search of T', and $O(m_1)$ per page, of which there are $O(n/m_1)$, giving us $O(n)$ time.

We are left with a binary search tree which has been properly packed into our memory in total time $O(nlg(m_1))$.

4.5 Expected Cost ApproxMWPaging

First, we bound the depth of nodes in our BST T. The depth of a key x_i (denoted (x_{i-1}, x_i) for unsuccessful searches) is defined as $d_T(x_i)$ ($d_T(x_{i-1}, x_i)$).

Lemma 4.5.1. *For a key x_i,*

$$d_T(x_i) \leq \lceil lg(m_1) \rceil * \lfloor log_{m_1}(\tfrac{1}{p_i}) \rfloor.$$

For a key (x_{i-1}, x_i),

$$d_T(x_{i_1}, x_i) \leq \lceil lg(m_1) \rceil * (\lfloor log_{m_1}(\tfrac{2}{q_i}) \rfloor + 1).$$

Proof. First we note that in the tree T' we build using Bose and Douïeb's multiway tree algorithm, the maximum depth of keys (call this $d'_T(x_i), d'_T(x_{i-1}, x_i)$) for a page size m_1 is [**?**]:

$$d_T(x_i) \leq \lfloor log_{m_1}(\tfrac{1}{p_i}) \rfloor.$$

$$d_T(x_{i-1}, x_i) \leq \lfloor log_{m_1}(\tfrac{2}{q_i}) \rfloor + 1.$$

As explained in the paper, these follow from Lemmas 1 and 2 of Bose and Douïeb [**?**].

Within a page, we make a balanced (ignoring weight) BST, so each key has a depth *within* a page of at most $\lceil lg(m_1) \rceil$.

Since our algorithm always connects the root of the BST made for page to a key in the BST made for the page's parent, a key x_i has a depth of at most $\lfloor log_{m_1}(\frac{1}{p_i}) \rfloor$ in terms of pages accessed ($\lfloor log_{m_1}(\frac{2}{q_i}) \rfloor + 1$ pages for keys $((x_{i-11}, x_i))$. Within a page, we will examine at most $\lceil lg(m_1) \rceil$ keys. Thus, a key's depth is at most the bound described. □

We now describe the cost of searching for a key located at deepest part of tree T: W. Let $m'_j = \sum_{k \leq j} m_j$. We define $m'_0 = 0$. Let l be the smallest j such that $m'_j > n$. l represents how many levels of memory will be required for storing our tree. Let $H(T)$ represent the height of our tree.

Lemma 4.5.2. $W \leq \sum_{k=1}^{l-1}(\lfloor lg(m'_i + 1)\rfloor - \lfloor lg(m'_{i-1} + 1)\rfloor) * c_i +$
$(H(T) - \lfloor lg(m'_{l-1} + 1)\rfloor) * c_l$

Proof. Consider the accessing each key along the path from the root to the deepest key in T. Note that we will have to examine $H(T)$ keys in order to access the deepest leaf in the tree. We will access one key a depth 0, one key at depth 1 and so on. Because the tree is packed into memory in BFS order, a key a depth i will be at memory location at most $2^i - 1$. Now, consider how many levels of the binary search tree T will fit inside of m_1, the fastest memory. In order for all keys of depth i (and higher) to be in m_1 we need:
$2^i - 1 \leq m_1 \implies i \leq lg(m_1 + 1)$.
Thus, at least $\lfloor lg(m_1 + 1)\rfloor$ levels of T fit on m_1. Next we examine how many fit on m_j for $1 < j < l$. The last level to completely fit on m_j or higher memories is the maximum i such that:
$2^{i+1} - 1 \leq m'_2 \implies i \leq lg(m'_j + 1)$.
Thus, at least $\lfloor lg(m'_j + 1)\rfloor$ levels of T fit on m_j or higher levels of memory.
Thus, on our search for the deepest key in the tree, we will make at least $\lfloor lg(m_1 + 1)\rfloor$ checks for elements located at memory m_1. This will cost a total of

$$\lfloor lg(m_1 + 1)\rfloor * c_1.$$

For each memory level m_j for $1 < j < l$, we will make at least $\lfloor lg(m'_j + 1)\rfloor$ checks in m_j or higher memories. Of these checks, at least $\lfloor lg(m'_{j-1} + 1)\rfloor$ will be in memory levels strictly higher up in the memory hierarchy than j. Since $c_j > c_i$ for $j > i$, an upper bound on the cost of searching for all elements in memory level j on the path from the root to the deepest element of the tree is:

$$(\lfloor lg(m_1 + 1)\rfloor - \lfloor lg(m'_{j-1} + 1)\rfloor) * c_j.$$

Finally, we can upper bound the cost by assuming that all remaining searches take place at c_l. The searches at level l will cost at most:

$$(lg(H(T) - \lfloor lg(m'_{l-1} + 1)\rfloor) * c_l.$$

Combining the above three equations gives the desired bound.

□

Next, we note that the cost of search for a key at some depth i is at most $\frac{i}{H(T)} * W$.

Lemma 4.5.3. *The cost of searching for a keys x_i and (x_{i-1}, x_i) ($C(x_i)$ and $C(x_{i-1}, x_i)$ respectively)can be bounded as follows:*

$$C(x_i) \leq \frac{\lceil lg(m_1) \rceil * \frac{lg(\frac{1}{p_i})}{lg(m_1)}}{\lceil lg(2n+2) \rceil} * W$$

$$C(x_{i-1}, x_i) \leq \frac{\lceil lg(m_1) \rceil * \lfloor log_{m_1}(\frac{2}{q_i}) \rfloor}{\lceil lg(2n+2) \rceil} * W$$

Proof. From Lemma 4.5.1 we have a bound on the depth of keys x_i and (x_{i-1}, x_i). Moreover, we know that since there are $2n + 1$ keys in our tree, our tree must have a height of at least:

$$\lceil lg(2n + 2) \rceil.$$

Note that since our tree is stored in BFS order in memory, whenever we examine a key's child, it will be at a memory location of at least the same, if not higher cost (by being in the same or a deeper page). Thus, the path from the root to a specific key can be upper bounded by the cost of searching for the deepest element in the tree, multiplied by the fraction of the way down the tree x_i or (x_{i-1}, x_i) are. Note that when searching for keys, we must search along the entire path from root to the key in question, while we need only examine the path from the root to the parent of a key for unsuccessful (x_{i-1}, x_i) searches. Combining 4.5.1 and 4.5.4 with the minimum height of our tree gives:

$$C(x_i) \leq \frac{\lceil lg(m_1) \rceil * \lfloor log_{m_1}(\frac{1}{p_i}) \rfloor}{\lceil lg(2n+2) \rceil} * W$$
$$\implies C(x_i) \leq \frac{\lceil lg(m_1) \rceil * \frac{lg(\frac{1}{p_i})}{lg(m_1)}}{\lceil lg(2n+2) \rceil} * W$$

Similarly,
$$C(x_{i-1}, x_i) \leq \frac{\lceil lg(m_1) \rceil * \lfloor log_{m_1}(\frac{2}{q_i} + 1 - 1) \rfloor}{\lceil lg(2n+2) \rceil} * W$$
$$C(x_{i-1}, x_i) \leq \frac{\lceil lg(m_1) \rceil * \frac{lg(\frac{2}{q_i})}{lg(m_1)}}{\lceil lg(2n+2) \rceil} * W.$$

This completes the proof.

□

We can now bound the expected cost of search using the bounds for each key.

Theorem 4.5.4. $C \leq (\frac{\lceil lg(m_1) \rceil}{\lceil lg(2n+2) \rceil * lg(m_1)} * W) * (H+1)$

Proof. The total expected cost of search is simply the sum of the weighted cost of search for all keys. Given our last lemma, we have that:

$C \leq \sum_{i=1}^{n} p_i * C(x_i) + \sum_{j=1}^{n+1} q_i * C(x_{i-1}, x_i)$

$\implies C \leq \sum_{i=1}^{n} p_i * \frac{\lceil lg(m_1) \rceil * \frac{lg(\frac{1}{p_i})}{lg(m_1)}}{\lceil lg(2n+2) \rceil} * W + \sum_{j=1}^{n+1} q_i * \frac{\lceil lg(m_1) \rceil * \frac{lg(\frac{2}{q_i})}{lg(m_1)}}{\lceil lg(2n+2) \rceil} * W$

$\implies C \leq (\frac{\lceil lg(m_1) \rceil}{\lceil lg(2n+2) \rceil * lg(m_1)} * W) * (\sum_{i=1}^{n} p_i * lg(\frac{1}{p_i}) + \sum_{j=1}^{n+1} (q_i * lg(\frac{2}{q_i})))$

$\implies C \leq (\frac{\lceil lg(m_1) \rceil}{\lceil lg(2n+2) \rceil * lg(m_1)} * W) * (\sum_{i=1}^{n} p_i * lg(\frac{1}{p_i}) + \sum_{j=1}^{n+1} (q_i * lg(\frac{1}{q_i})) + 1$

$\implies C \leq (\frac{\lceil lg(m_1) \rceil}{\lceil lg(2n+2) \rceil * lg(m_1)} * W) * (H+1).$ □

4.6 Approximate Binary Search Trees of De Prisco and De Santis with Extensions by Bose and Douïeb

As in the classic Knuth problem, we are given a set of n probabilities of searching for words $(p_1, p_2, ..., p_n)$, as well as $n+1$ probabilities of unsuccessful searches $(q_0, q_1, ..., q_n)$. The duo provides an algorithm which constructs a binary search tree in $O(n)$ time with an expected cost of at most [**?**]

$$H + 1 - q_0 - q_n + q_{max}$$

where q_{max} is the maximum probability of an unsuccessful search. This was later modified by Bose and Douïeb (the same paper described in section TODO) to have an improved bound [**?**]

$$P_T \leq H + 1 - q_0 - q_n + q_{max} - \sum_{i=0}^{m'} pq_{rank[i]}$$

P_T is the average path length of the tree built using their algorithm and $m' = max(2n - 3P, P) - 1 \geq \frac{n}{2} - 1$. P is the number of increasing or decreasing sequences in a left-to-right read of the access probabilities of the leaves. Moreover, $pq_{rank[i]}$ is the i^{th} smallest access probability among all keys and leaves except q_0 and q_n.

First, I explain the algorithm of De Prisco and De Santis and then explain the extensions of Bose and Douïeb. De Prisco and De Santis' algorithm occurs in three phases.

In **Phase 1**, an auxiliary probability distribution is created using $2n$ zero probabilities, along with the $2n+1$ successful and unsuccessful search probabilities. Yeung's linear time alphabetic search tree algorithm is used with the $2n+1$ successful and unsuccessful search probabilities used as leaves of the new tree created. This is referred to as the *starting tree*.

In **Phase 2** what's known as the *redundant tree* is created by moving p_i keys up the *starting tree* to the lowest common ancestor of keys q_{i-1} and q_i. The keys which used to be called p_i are relabelled to $old.p_i$.

In **Phase 3** the *derived tree* is constructed from the *redundant tree* by removing redundant edges. Edges to and from nodes which represented zero probability keys are deleted. This derived tree is a binary search tree with the expected search cost described.

In Bose and Douïeb's work, they explain how they can substitute their algorithm for Yeung's linear time alphabetic search tree algorithm which results in a better bound (as described above). We use the updated version (by Bose and Douïeb) of De Prisco and De Santis' algorithm as a subroutine in the sections to follow.

4.7 Algorithm ApproxBSTPaging

Our second solution to create an approximately optimum BST under the HMM model works as follows:

1) First, we create an approximately optimal BST T using the algorithm of De Prisco and De Santis [**?**] (as updated by Bose and Douïeb [**?**]). This takes $O(n)$ time.

2) In a similar fashion to step 4) of *ApproxMWPaging*, we pack things into memory in a breadth first search order of T starting from the root. This relative simple traversal also takes $O(n)$ time.

We are left with a binary search tree which has been properly packed into our memory in total time $O(n)$.

4.8 Expected Cost ApproxBSTPaging

As explained in the Bose and Douïeb paper, the average path length search cost of the tree created by their algorithm is at most:
[**?**] $P_T \leq H + 1 - q_0 - q_n + q_{max} - \sum_{i=0}^{m'} pq_{rank[i]}$

Similar to the previous section, if we can bound the cost of search for a given path length, then we can form a bound on the average cost of search in the HMM model. Consider $min_{p,q}$ for our probability distribution. It is implicitly shown in the work of Bose and Douïeb that:

As before, we can describe the cost of searching for a key located at deepest part of tree T: W. Recall, $m'_j = \sum_{k \leq j} m_j$, $m'_0 = 0$ and let l be the smallest j such that $m'_j > n$.

Lemma 4.8.1.
$W \leq \sum_{k=1}^{l-1} (\lfloor lg(m'_i + 1) \rfloor - \lfloor lg(m'_{i-1} + 1) \rfloor) * c_i$
$+ (H(T) - \lfloor lg(m'_{l-1} + 1) \rfloor) * c_l$

Proof. Since we are simply putting things into memory in BFS order, and all we use is the height of the tree and the memory hierarchy, the proof is identical to that of **Lemma 4.5.3**. □

Next, we note that the cost of search for a key at some depth i is at most $\frac{i}{H(T)} * W$. This must be true because the cost of examining nodes can only increase as we move down the tree because of the increasing property of the cost of access of our various levels of memory and our BFS packing algorithm. Since, we also know the expected path length of our tree we can show that:

Theorem 4.8.2. $C \leq (\frac{W}{H(T)}) * (H + 1 - q_0 - q_n + q_{max} - \sum_{i=0}^{m'} pq_{rank[i]})$

Proof. Bose and Douïeb show that after using their algorithm for Phase 1 of De Prisco and De Santis algorithm algorithm, every leaf of the *starting tree* (all keys representing successful and unsuccessful searches) are at depth at most $\lfloor lg(\frac{1}{p}) \rfloor + 1$ for at least $\{2n - 3P, P\} - 1$ of $p \in \{\{p_1, p_2, ..., p_n\} \cup \{q_0, q_1, ..., q_n\}\}$ and $\lfloor lg(\frac{1}{p}) \rfloor + 2$ for all others. Recall that P is the number of peaks in the probability distribution $q_0, p_1, q_1, ..., p_n, q_n$. Moreover, after phases two and three of the algorithm, each key has its depth decrease by 2, and all leaves (except one) move up the tree by one. Now, let $C_d(i)$ represent the cost of accessing a node at depth i. As mentioned above, this is at most $\frac{i}{H(T)} * W$. Using this, we can derive the following equation an upper bound on the expected cost of search:

$$C \leq \sum_{i=1}^{n} (p_i * \frac{depth(p_i)}{H(T)} * W) + \sum_{i=0}^{n} (q_i * \frac{(depth(q_i)-1)}{H(T)} * W)$$

$$\implies C \leq \frac{W}{H(T)} (\sum_{i=1}^{n} (p_i * depth(p_i)) + \sum_{i=0}^{n} (q_i * (depth(q_i) - 1)))$$

$$\implies C \leq \frac{W}{H(T)} (\sum_{i=1}^{n} (p_i * depth(p_i)) + \sum_{i=0}^{n} (q_i * (depth(q_i) - 1)))$$

$$\implies C \leq \frac{W}{H(T)} (\sum_{i=1}^{n} (p_i * \lfloor lg(\frac{1}{p_i}) \rfloor + \sum_{i=0}^{n} (q_i * \lfloor lg(\frac{1}{q_i}) - 1 + 1 \rfloor))$$

Note that the expected cost is equal to the expected path length for each key, multiplied by the cost of search for each key. Since we have an upper bound on the cost of search for an element at some depth i, and an upper bound on the expected path length TODO □

IS IT DEPTH + 1 FOR THE KEYS? SHOULD WE BE DOING AN EXTRA ACCESS FOR LEAVES RELATIVE TO HOW THEY DID IT BEFORE.

4.9 Improvements over Thite in the HMM_2 Model

The HMM_2 model is the same as the general HMM model with the added constraint that there are only two types of memory (slow and fast). In Thite's thesis, he proposed both an optimal solution to the problem, as well as an approximate solution that runs in time $O(nlg(n))$ [**?**]. I show that my solution is at least as good, and may be better in certain situations. First I give an alternate bound to my expected cost which is similar to the one presented in Thite's work.

Lemma 4.9.1. *Let* $m'_j = \sum_{k \leq j} m_j$. *We define* $m'_0 = 0$. *Let* l *be the smallest* j *such that* $m'_j > n$. *Then the cost of search:*
$C \leq c_l * (H + 1 - q_0 - q_n + q_{max} - \sum_{i=0}^{m'} pq_{rank[i]})$.

Proof. By using the ApproxBSTPaging algorithm, we can simply assume all key accesses happen in the most expensive type of memory (m_l). Thus, our expected cost can be bounded by an upper bound on the average path length, multiplied by this upper bound on our cost of accessing each item. □

Moreover, for both ApproxMWPaging and ApproxBSTPaging:

Lemma 4.9.2. *Suppose* $\sum_{k=1}^{l-1}(c_{i+1} - c_i) * (\lfloor lg(m'_i + 1) \rfloor) > \lfloor lg(m'_1 + 1) \rfloor * c_1$. *Then* $W < c_l * H(T)$.

Proof. $W \leq \sum_{k=1}^{l-1}(\lfloor lg(m'_k + 1) \rfloor - \lfloor lg(m'_{k-1} + 1) \rfloor) * c_i$
$+(height(T) - 1 - \lfloor lg(m'_{k-1} + 1) - 1 \rfloor) * c_l$
Rearranging the formula gives:
$\implies W \leq H(T) * c_l + \lfloor lg(m'_1 + 1) \rfloor - \sum_{k=1}^{l-1}(c_{k+1} - c_k) \lfloor lg(m'_k + 1) \rfloor$
Thus, as long as $\sum_{k=1}^{l-1}(c_{k+1} - c_k) \lfloor lg(m'_k + 1) \rfloor > \lfloor lg(m'_1 + 1) \rfloor$ then we will get our desired result.

□

Note that our assumption will generally be true since $c_l > c_{l-1} > ... > c_1$ and $m_l > m_{l-1} > ... > m_1$. Finally, we can see that:

Theorem 4.9.3. *Suppose* $\sum_{k=1}^{l-1}(c_{i+1} - c_i) * (\lfloor lg(m'_i + 1) \rfloor) > \lfloor lg(m'_1 + 1) \rfloor * c_1$. *Then* $C < c_l * (H + 1 - q_0 - q_n + q_{max} - \sum_{i=0}^{m'} pq_{rank[i]})$

Proof. $C \leq (\frac{W}{H(T)}) * (H + 1 - q_0 - q_n + q_{max} - \sum_{i=0}^{m'} pq_{rank[i]})$ Assuming $\sum_{k=1}^{l-1}(c_{i+1} - c_i) * (\lfloor lg(m'_i + 1) \rfloor) > \lfloor lg(m'_1 + 1) \rfloor * c_1$ and using **Lemma 4.9.2** gives:
$\implies C < (\frac{c_l * H(T)}{H(T)}) * (H + 1 - q_0 - q_n + q_{max} - \sum_{i=0}^{m'} pq_{rank[i]})$
$\implies C < c_l * (H + 1 - q_0 - q_n + q_{max} - \sum_{i=0}^{m'} pq_{rank[i]})$ □

TODO UNDERSTAND WHY THIS ISN'T A DIRECT IMPROVEMENT. I'M CONFUSED

Subbing our assumption into the HMM_2 model gives: $(c_2 - c_1) * (\lfloor lg(m_1 + 1) \rfloor) > \lfloor lg(m_1 + 1) \rfloor * c_1$
$\implies c_2 > 2 * c_1$
which is a valid assumption in many types of memory hierarchies.

Corollary. $C < c_2 * (H + 1 - q_0 - q_n + q_{max} - \sum_{i=0}^{m'} pq_{rank[i]})$
whenever $c_2 > 2 * c_1$

In many types of memory hierarchies, this will provide an improvement over Thite's bound of:
$C < c_2 * (H + 1)$
especially in the case where $c_2 >> c_1$.

Chapter 5

Approximate Binary Search in the Hierarchical Memory with Block Transfer Model

5.1 Hierarchical Memory with Block Transfer Model

This model (HMBTM for short) was proposed by Aggarwal, Chandra, and Snir as an improvement over the HMM model [**?**] discussed previously [**?**]. The model with block transfer still has a memory hierarchy with increasing costs of access, but allows for any contiguous block of memory to be copied from one location to another with constant unit time per element after the initial access. This provides a better model for standard computers which can copy many words from one type of memory to another after accessing a single word. Like the HMM model, we have an unlimited number of registers (numbered 1, 2, , .. etc.) and we will still use the cost function as was explained in Thite's thesis [**?**]. Here, $\mu(a)$ is the cost of accessing memory location a. We have a series of memory sizes $m_1, m_2, ..., m_h$ where m_l has infinite size each with an associated cost $c_1, c_2, ..., c_h$. We assume that $c_1 < c_2 < ... < c_h$.

$$\mu(a) = c_i if \sum_{j=1}^{i-1} m_j < a \leq \sum_{j=1}^{i} m_j.$$

We explicitly assume that successive memory sizes divide one another evenly:

$$m_1 < m_2 < ... < m_h \text{ and}$$

$$\forall i \in \{1, 2, ..., h\},\ m_i \mid m_{i+1}.$$

In this new model, we are given the additional operational opportunity to move a block from one location to another. Specifically, a block copy operation is defined as:
$[x - l, x] \rightarrow [y - l, y]$.
The contents of $\mu(x - i)$ are copied to $\mu(y - i)$ for $i \in \{0, ..., l\}$. This is valid if the two intervals are disjoint. This copy operation costs $max(f(x, y)) + l$.

I provide fast but approximate solutions to the optimum BST problem under new model in the following sections.

5.2 ApproxMWPaging with BT

The first algorithm I provide is very similar to ApproxMWPaging. The first three steps of the algorithm are in fact identical. To review, we create a Multiway Tree T' using the algorithm of Bose and Douïeb, form balanced BST's within each page of T', and then connect these balanced BSTs from T' to form a valid BST T. This is done in $O(nlg(m_1))$ time.

To decide packing order, we will do a modified BFS of T'. We will always pack individual pages of T' into memory in a BFS of the balanced BST formed from the page.

BT–Range-Pack packs the tree T using r_q as roots from memory location loc. It packs recursively into pages of size m_i (or smaller if necessary) until m_{i+1} nodes have been packed or we run out of roots to pack from in $r\_q[]$. If $i = 0$ then we only pack amt nodes (so we can fill memory levels appropriately). Note that we only remove the top root from r_q if the root has been completely used i.e. it has been packed into a page of size $\leq m_1$. We always return a pair, the updated r_q which contains new roots to search packed from (added in BFS order from leaves of m_1 or smaller packed "pages") and the current location in memory where we should pack into next.

We call BT–Pack() in order to pack nodes into the tree. Essentially, this function attempts to pack each level of the memory hierarchy independently from m_1 to m_l. Within each memory level i, we attempt to pack it from a single source (the next available node in BFS order of our tree) in blocks of size m_{i-1} which are recursively packed in blocks of size m_{i-2}. If we cannot fill a memory level, or a specific block within the packing of a memory

level, we will pack whatever we can starting again from a new root (still in BFS order) passing into our BT–Range-Pack function the biggest i possible such that m_i will still fit in the page we are trying to pack.

We are left with a binary search tree which has been properly packed into our memory in total time $O(n * l)$.

Algorithm 1 ApproxPaging with BT Packing

```
1:  procedure BT–RANGE-PACK(r_q[], i, loc, amt)
2:      if i = 0 then
3:          Create largest BFS tree T possible from r_q[0] with at min(m_1, amt) nodes
4:          Push T in BFS order into memory starting from memory location loc
5:          loc+ = size(T)
6:          r_q.pop()
7:          for each leaf l in T in BFS order do
8:              Push all children of l onto r_q
9:          end for
10:         return (r_q[], loc)
11:     else
12:         total = 0
13:         while total < m_{i+1} and ¬(empty? r) do
14:             old_loc = loc
15:             i = max_{0,...,j+1}(i : m_i + total < m_{j+1})
16:             (loc, new_r_q) = AMWBT–Range-Pack (r_q[0, ..., 0], i − 1, loc, m_i)
17:             for each node r in new_r_q do
18:                 r_q.push(r)
19:             end for
20:             total+ = loc − old_loc
21:         end while
22:         return (r_q[], loc)
23:     end if
24: end procedure
```

Algorithm 2 ApproxPaging with BT Packing

```
1: procedure BT–Pack
2:     r_q = empty
3:     r_q.push_back(root)
4:     loc = 0
5:     for j = 0 to l − 1 do
6:         while loc < m_{j+1} and ¬(empty?r_q) do
7:             if loc + m_1 < m_{j+1} then
8:                 i = max_{0,...,j+1}(i : m_i + total < m_{j+1})
9:                 (loc, new_r_q) = AMWBT–Range-Pack ([r_q[0]], i − 1, loc, m_i)
10:            else
11:                (loc, new_r_q) = AMWBT–Range-Pack ([r_q[0]], 0, loc, m_i)
12:            end if
13:            r_q.pop()
14:            for each node r in new_r_q do
15:                r_q.push(r)
16:            end for
17:        end while
18:    end for
19: end procedure
```

5.3 ApproxMWPaging with BT Running Time

First, we examine the runtime of BT–Range-Pack($r\_q[], i, loc, amt$).

Lemma 5.3.1. *BT–Range-Pack($r\_q[], i, loc, amt$) returns an updated location, call it new_loc in time at most $O((i+1)*(new\_loc-loc))$ time and packs $(new\_loc-loc)$ nodes into memory.*

Proof. I will prove this by induction on i. In the base case ($i = 0$) we simply do a BFS from the given root, packing as many of $min(m_1, amt)$ into memory as possible, we return an updated loc which is exactly equal to the size of the BFS tree we have created. We also push all children of all leaves of this BFS tree onto our r_q. This all takes time $(new\_loc - loc) \in O(1 * (new\_loc - loc))$ as required.

Suppose BT–Range-Pack($r\_q[], i, loc, amt$) returns new_loc in time at most $O(l*(new\_loc-loc))$ time and packs $(new\_loc-loc)$ nodes into memory for all $i < k$. Consider BT–Range-Pack($r\_q[], k, loc, amt$). Since $i \neq 0$ we enter the else case on line 11. Now, we may call BT–Range-Pack a number of times, but the sum of loc updates will be at most m_{i+1}. By our induction hypothesis, we know each BT–Range-Pack($r\_q[], i, loc, amt$) for $i < k$ packs sum number $(new\_loc - loc)$ of nodes into memory in time $O((i+1) * (new\_loc - loc))$. Moreover, other than the recursive calls, all lines take $O(1)$ except line 15 which takes $O(i)$ and lines 17-18 which take $(new\_loc - loc)$ time since only a linear amount of children can be pushed onto r_q for each item packed into memory (and each child can only be pushed to r_q once, by its single parent). Let S be the set of all calls q BT–Range-Pack($r\_q[], q_i, loc, amt$) is called directly from our original BT–Range-Pack($r\_q[], k, loc, amt$) call and packs q_amt items into memory. Let $T_{RP}(i, new\_loc - loc)$ be the cost of BT–Range-Pack($r\_q[], i, loc, amt$) which packs $new\_loc - loc$ items into memory. The we have that:

$T_{RP}(k, new\_loc - loc) = O(new\_loc - loc) + O(i) + \sum_{q \in S} T_{RP}(q_i, q\_amt)$
By our induction hypothesis we have that:
$\implies T_{RP}(k, new\_loc - loc) \in O(new\_loc - loc) + O(i) + \sum_{q \in S} O(q_i * q\_amt)$
Since q_i is at most i we have:
$\implies T_{RP}(k, new\_loc - loc) \in O(new\_loc - loc) + O(i) + i * \sum_{q \in S} q\_amt$
Finally, since $new\_loc - loc = \sum_{q \in S} q\_amt$:
$\implies T_{RP}(k, new\_loc - loc) \in O(new\_loc - loc) + O(i) + i * (new\_loc - loc)$
$\implies T_{RP}(k, new\_loc - loc) \in O((new\_loc - loc) * (i+1))$ as required. □

This leads us to our runtime for BT–Pack().

Lemma 5.3.2. *BT–Pack packs all nodes in our tree T into memory in time $O(n * l)$.*

Proof. First note that, as in BT–Range-Pack, we always update loc to be the new location where should pack nodes into memory. Lines $5 - 18$ loop through each memory level. For a given level, we pack as much as can into it, using as large BT–Range-Pack's as possible. Specifically, line 8 takes $O(l)$ and lines $14 - 15$ take $(new\_loc - loc)$ time since only a linear amount of children can be pushed onto r_q for each item packed into memory (and each child can only be pushed to r_q once, by its single parent). Let S_h be the set of all calls q BT–Range-Pack($r\_q[], q_i, loc, amt$) called from our BT–Pack when $j = h$ (these each pack q_amt of nodes into memory). Let $packed\_amt$ be the amount packed by a call to BT–Range-Pack in lines 9 or 11. By our previous proof, these take time $O((packed\_amt) * (i))$. Since i is at most l and $sumpacked\_amt = n$ (we never pack

anything twice) we have that the time required for BT–Pack T_P is:
$T_P \in \sum_{j=0}^{l-1}) sum_{q \in S_j} O(q\_amt * j)$
$\implies T_P \in o(l) * \sum_{j=0}^{l-1}) sum_{q \in S_j} O(q\_amt)$
$\implies T_P \in O(n * l)$ as required.

□

5.4 Search with ApproxMWPaging with BT

Algorithm 3 ApproxMWPaging with BT Search

```
1: procedure AMWBT–Search(mem, key, prev_moved_s, offset_s)
2:     if (¬(empty?prev_moved_s) and (mem >= prev_moved_s.top())) or (mem ≥ m_1) then
3:         if (¬(empty?prev_moved_s) and (mem >= prev_moved_s.top())) then
4:             mem+ = offset_s.top()
5:             offset_s.pop()
6:             prev_moved_s.pop()
7:             if neq(empty?offset_s) then
8:                 mem− = offset_s.top()
9:             end if
10:        end if
11:        offset_s.push(mem)
12:        i = min_{m'_i : m'_i > mem}
13:        amt_move = m'_i − mem
14:        prev_moved_s.push(amt_move)
15:        [mem, mem + amt_move − 1] → [0, amt_move − 1]
16:        AMWBT–Search(0, key, prev_moved_s, offset_s)
17:    else
18:        offset = 0
19:        if ¬(empty?prev_s_moved) and (mem >= prev_moved_s.top()) then
20:            offset = offset_s.top()
21:        end if
22:        if key = mem.key then
23:            return mem.value
24:        else if key < mem.key then
25:            if mem has no left child then
26:                return mem
27:            else
28:                AMWBT–Search(mem.left_child − offset, key, prev_moved_s, offset_s)
29:            end if
30:        else
31:            if mem has no right child then
32:                return mem
33:            else
34:                AMWBT–Search(mem.right_child − offset, key, prev_moved_s, offset_s)
35:            end if
36:        end if
37:    end if
38: end procedure
```

To run we call AMWBT–Search$(0, key, empty, empty)$ since the memory location of the root should be 0.

5.5 Expected Cost ApproxMWPaging with BT

How a path has cost What cost is similar arg to ApproxMWPaging

5.6 ApproxBSTPaging with BT

Given what we have already shown, this algorithm is extremely simple to explain. We first create a BST T using the algorithm of De Prisco and De Santis [**?**] (as updated by Bose and Douïeb [**?**]) in $O(n)$ time (as was the first step in ApproxBSTPaging). We then simply call BT–Pack() in order to pack the tree into memory as described in the ApproxMWPaging with BT sections.

5.7 Expected Cost ApproxBSTPaging with BT

How a path has cost What cost is similar arg to ApproxMWPaging

Chapter 6

Conclusion and Open Problems

6.1 Discussion

6.2 Conclusion

APPENDICES

Appendix A

Matlab Code for Making a PDF Plot

References