Optimal Shortcuts for Balanced Search Trees - A Technical Report

Jeff Hartline
Cornell University
jhartlin@cs.cornell.edu

February 2, 2004

Abstract

We present an alternative to tree rebalancing for improving the expected search cost in weighted binary search trees. This alternative is based on the insertion of shortcut links between nodes in the search tree. We propose several shortcut models and give polynomial time algorithms to find the best shortcuts for two of these models.

1 The Problem

We are given a binary search tree T with weights associated with all leaf nodes. For each internal node v we are allowed to insert some number of links, or shortcuts, between v and descendents of v. The cost of the resulting shortcut search tree is the sum of the shortest distances between the root and all leaves weighted by their weights. Our goal is to find shortcuts that minimize tree cost.

We look at a restricted version of this problem in which tree T is a balanced binary tree of depth d and each node can be the source of at most one shortcut. We consider three shortcut models and give a detailed solution for two of these models. The first of these models is the *leafcut* model, in which all shortcuts are between internal nodes and leaf nodes. The *general shortcut* model allows shortcuts to any descendent node, leaf or internal. The last model, the *navigable shortcut* model, allows to any descendant node but does not allow shortcuts to cross each other.

This work is developed as an alternative to tree rebalancing schemes and other search techniques such as standard optimal binary search trees, skip lists (1), and dynamically rebalanced splay trees (2). The results presented in this paper deal only with the static solution of the shortcut problem but are motivated as a step towards using shortcuts in dynamic search scenarios where performing rotation operations on an existing tree structure is expensive or impossible. *Need much more extensive background.*

In the following sections, we present notation and algorithms for efficient solutions to the leafcut and general models. We begin in Section 2 by enumerating some basic notation. In Section 3 we propose a greedy algorithm for choosing optimal shortcuts and show that it gives non-optimal solutions. Next, we enumerate some preliminary results in Section 4. Section 5 contains algorithm descriptions and proofs for the dynamic programming solutions to the optimal shortcuts problem for the leafcut and general models. The techniques used in these solutions can be applied to obtain a solution for the navigable shortcut model. Finally, we conclude in Section 6 by describing some potential topics for further research on shortcut search trees.

2 Basic Notation

The following is notation we will use in reference to the shortcut search tree problem:

- T is a balanced binary search tree
- $root(T)$ is the root of T

- I is set of internal nodes of T
- L is set of leaf nodes of T
- n is the number of nodes in T
- $d = \log(n)$ is the depth of T
- $d(t)$ is the depth of node $t \in T$
- $w(t)$ is the weight of leaf $t \in L$
- $\mathit{left}(t)$ is the left child of $t \in I$
- $\mathit{right}(t)$ is the right child of $t \in I$
- $\mathit{parent}(t)$ is the parent of $t \in T - \mathit{root}(T)$
- S_T, or just S, is a set of feasible shortcuts (a mapping of nodes to descendent nodes)
- $d_S(t)$ is the shortest distance between $\mathit{root}(T)$ and node $t \in T$ in tree T with shortcuts S
- $\mathit{COST}_S(T) = \Sigma_{t \in T} w(t) \cdot d_S(t)$ is the cost of feasible shortcut solution S

3 Greedy Shortcut Search Trees

In this section we introduce greedy algorithms for finding low cost shortcuts in the leafcut, general, and navigable shortcut models. The greedy algorithms are shown to be sub-optimal, hence motivating the need for the more complicated dynamic programming technique described in Section 5.

The greedy principle for finding good shortcuts is as follows: add shortcuts top down (from the root to the leaves) and at each node we consider all shortcuts allowed by the shortcut model we are using. We pick which shortcut to add to maximize immediate improvement in cost.

To be more formal, recall that for a set of shortcuts S, the cost of the search tree T with shortcuts S is $\mathit{COST}_S(T)$. For some node r and some set of shortcuts that all originate from ancestors of r, denote $s_r = \mathit{GREED}(r, S)$ as the greedy shortcut that originates from r chosen from the set of legal shortcuts to minimize $\mathit{COST}_{S \cup s_r}(T)$. Greedy shortcuts for children of r are determined recursively: $\mathit{GREED}(\mathit{left}(r), S \cup s_r)$ and $\mathit{GREED}(\mathit{right}(r), S \cup s_r)$ for $\mathit{left}(r)$ and $\mathit{right}(r)$, respectively. The greedy algorithm starts the recursion at $r = \mathit{root}(T)$ with $S = \phi$. The final set of shortcuts S_{GREED} is the union of all greedy shortcuts s_r computed by this algorithm.

Theorem 1. *For all shortcut models, greedy can be sub-optimal.*

Proof. This fact is easily shown by the example in Figure 1. Greedy gives the same solution for the leafcut, general, and navigable shortcut models. This solution has $\mathit{COST}_{S_{GREED}} = 6x + O(\epsilon)$. This is clearly sub-optimal because adjacent to the greedy solution is a cost $5x + O(\epsilon)$ solution. □

4 Preliminary Results

For the leafcut, general, and navigable shortcut models we construct an optimal search tree using a dynamic programming technique. This technique is described in detail in Section 5. First, however, we will go through some preliminary results that will be referenced repeatedly in our algorithm descriptions and proofs.

As stated in Section 1 this paper deals only with the case that each internal node is allowed to implement (or originate) at most one shortcut. It is convenient for us to assume that all nodes are also the target of at most one shortcut. Lemma 1 assures us that for all shortcut models there exists some optimal solution in which no node is the target of multiple shortcuts. Therefore, without loss of generality, we restrict ourselves to solutions in which all internal nodes are both the source and target of at most one shortcut.

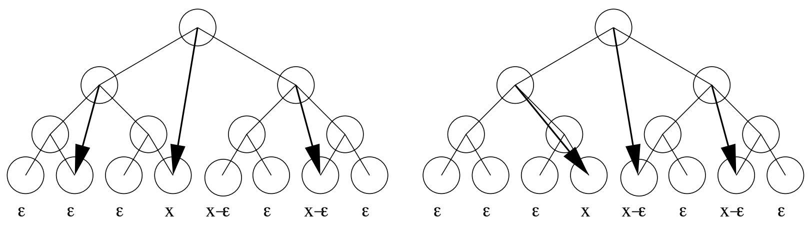


Figure 1: On the left is the greedy solution for the depth 3 tree with leaf weights shown below each leaf. Shortcuts are shown as bold arrows (observe that the greedy solution is the same for all shortcut models). The cost of the greedy solution is $6x + O(\epsilon)$. On the right is a solution with cost $5x + O(\epsilon)$, showing that the set of greedy shortcuts is not optimal.

Lemma 1. *For all shortcut models, an optimal solution exists in which no node is the target of more than one shortcut.*

Proof. Consider an optimal solution S in which node t is the target of two or more shortcuts. For each such t, modify S by removing all shortcuts to t except for one shortcut whose source v has minimal $d_S(v)$. The resulting solution cannot be more costly because the distance from the root to t was not increased. $\square$

Next, our dynamic programming algorithm makes extensive use of vectors and vector sets. Also, the efficiency of our solutions relies on polynomial bounds on the size of certain sets. The following is some notation for length d integer vectors and some important cardinality proofs:

- $|X|$ is the number of elements in a vector set X
- $|parts(X)|$ is the number of distinct *partitions* over all elements of a vector set X, where a *partition* of $x \in X$ is a tuple $(x_1 \in X, x_2 \in X)$ for which $x = x_1 + x_2$.
- $x(i)$ is the i^{th} element of vector x where we number elements $1 \ldots d$
- $|x|_1$ is the $1 - norm$, or sum of element magnitudes, of a vector x
- e_0 is all zeroes and e_i for $1 \leq i \leq d$ is defined by $e_i(i) = 1$ and $e_i(j \neq i) = 0$
- $V = \{0,1\}^d$ is the set of length d binary vectors (note $|V| = 2^d$)
- $W = \{w \mid w \in \mathbb{N}^d \wedge |w|_1 \leq d\}$

Lemma 2. *For large d, $\binom{kd}{d} \approx \sqrt{\frac{k}{2\pi d(k-1)}} \times (\frac{k^k}{(k-1)^{k-1}})^d$. Note that for $k = 2$ and $k = 3$ we obtain $\binom{2d}{d} \approx \sqrt{\frac{1}{\pi d}} \times 4^d$ and $\binom{3d}{d} \approx \sqrt{\frac{3}{4\pi d}} \times (\frac{27}{4})^d$, respectively.*

Proof. This formula is derived by applying Sterling's approximation for factorials $n! \approx \sqrt{2\pi n}(\frac{n}{e})^n$ to the definition of $\binom{kd}{d} = \frac{k!}{d!(kd-d)!}$. $\square$

Lemma 3. *Over all elements of V there are 3^d distinct vector partitions.*

Proof. A partition is defined as a triple $v_0 = v_1 + v_2$. We can represent any such triple for binary v_0 as single length d ternary vector t_v by letting $t_v(j) = 0$ if $v_0(j) = v_1(j) = v_2(j) = 0$, $t_v(j) = 1$ if $v_0(j) = v_1(j) = 1$ (which implies $v_2(j) = 0$), and $t_v(j) = 2$ if $v_0(j) = v_2(j) = 1$ (which implies $v_1(j) = 0$). There are 3^d such ternary vectors and so there are 3^d distinct vector partitions over all elements of V. $\square$

Lemma 4. *There are $\frac{4^d}{\sqrt{\pi d}}$ distinct elements of W.*

Proof. Recall $W = \{w \mid w \in \mathbb{N}^d \wedge |w|_1 \leq d\}$. We can use a balls and bins counting argument to determine the size of W. We start with d balls and d bins labeled $1 \ldots d$ plus a special 0 bin. A distribution of the d balls to the $d+1$ bins maps directly to an element of W. The number of balls in the j^{th} bin for positive j corresponds to the value of the j^{th} vector element. The 0 bin just holds extra balls in the case that the vector sum is less than d. The total number of distinct ball-to-bin assignments, and thus the number of distinct vectors in W, is given by $\binom{2d}{d}$. By Corollary 2, this is approximately $\frac{4^d}{\sqrt{\pi d}}$. $\square$

Lemma 5. *Over all elements of W there are approximately $\frac{\alpha^d}{\sqrt{\frac{4}{3}\pi d}}$ distinct vector partitions for $\alpha = \frac{27}{4}$.*

Proof. A partition is defined as a triple $w_0 = w_1 + w_2$. As in Lemma 4, imagine placing d balls into $d+1$ bins labelled $0, 1, \ldots d$. This is the number of possible vectors w_0. To consider possible partitions w_1 and w_2 we split bins $1 \ldots d$ in half giving us $2d+1$ bins labelled $0, 1_1, 1_2, \ldots d_1, d_2$. The number of balls in bin j_i corresponds to the value of $w_i(j)$ for $i \in \{1, 2\}$. The total number of balls in bins j_1 and j_2 is the value of $w_0(j)$. The 0 bin holds extra balls if the vector sum is less than d (observe it is not split into two because these balls are not partitioned between w_1 and w_2). The total number of ball-to-bin assignments, and this the number of distinct vector partitions of W, is given by $\binom{3d}{d}$. By Corollary 2, this is approximately $\frac{\alpha^d}{\sqrt{\frac{4}{3}\pi d}}$ for $\alpha = \frac{27}{4}$. $\square$

These results will be used to compute the space requirements and running time of our algorithms. Keeping them in mind, we can now proceed to a description of the dynamic programming algorithms for optimal shortcuts.

5 Optimal Shortcut Search Trees

5.1 Throughput Vectors

The success and efficiency of our algorithms for finding optimal shortcut trees relies strongly on the notion of *throughput* vectors for edges of balanced binary search tree T and the relationship between these vectors for adjacent edges. We consider a valid set of shortcuts S for tree T. Recall that $root(T)$ is the root of T, I is the set of internal nodes of T, L is the set of leaf nodes of T, and d is the depth of T. For a shortcut $s \in S$ with source $i \in I$ and target $t \in T$, we denote depth $d_S(s)$ as the shortest distance between $root(T)$ and t using shortcut s. It is not hard to see that $d_S(s) = d_S(i) + 1$: one more than the shortest distance between the root and the source of s. Furthermore, we say that shortcut s uses edge e if e is on the path between i and t. For some edge e, we consider all shortcuts in S that use e. We define the *throughput* vector of edge e as a representation of the shortcuts that use e. For $1 \leq j \leq d$ and $k \in \mathbb{N}$, $w_e(j) = k$ means that k depth j shortcuts use edge e. Throughput vectors are thus length d vectors of natural numbers.

We denote the throughput vector for edge e between nodes $parent(v)$ and v as w_v. A shortcut that uses this edge either terminates at v or also uses one of the edges between v and $left(v)$ or $right(v)$. The only shortcut that uses an edge between v and a child of v but not edge e is a shortcut that originates at v. By our assumptions and Lemma 1, at most one shortcut terminates at v and at most one shortcut originates at v. Therefore, the sum of the throughput vectors $w_{left(v)} + w_{right(v)}$ is related to a partitioning of w_v. Figure 2 shows a node v, shortcuts that pass through node v, and the throughput vectors for neighboring edges.

5.2 The Leafcut Model

Here we present an optimal solution to the shortcut search tree problem in the leafcut model. This solution requires $O(n^2)$ space and $O(n^{1+log(3)})$ time. The correctness and efficiency of our algorithm relies on the following lemma.

Lemma 6. *For a search tree T of depth d and any set of legal leafcuts S, the throughput vector of some edge e is an element of $V = \{0, 1\}^d$.*

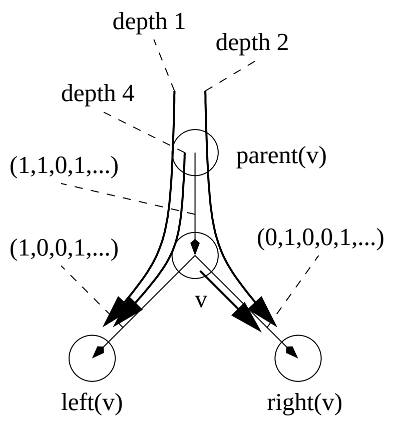


Figure 2: Here we show a node v, its parent $parent(v)$, and its two children $left(v)$ and $right(v)$. The parent of v is distance 3 from the root. Node v is therefore distance 4 from the root (because no shortcuts terminate at v). Shortcuts of depth 1 and 2 pass through $parent(v)$. Both $parent(v)$ and v also originate shortcuts. All edges are labelled with their throughput vectors, where we use "..." to indicate that the rest of the vector is zero. Observe that the sum of the throughput vectors below v equals the throughput vector above v plus $e_{d_S(v)+1}$, the contribution of the shortcut originated by v.

Proof. Leafcuts cannot change the distance between the root and any non-leaf nodes. All nodes v have at most one ancestor of any given depth and thus the edge incident to v is used by at most one shortcut of any given depth. Therefore $w_v \in \{0,1\}^d$. □

We compute our solution using a two dimensional dynamic programming table LCM with $LCM(r \in T, w \in V)$ denoting the optimal leafcut search cost in the subtree rooted at r with throughput vector $w_r = w$. The global optimal solution for T is therefore given by $LCM(root(T), e_0)$. We construct this optimal solution from the leaves upward using the following recurrence:

- $LCM(r \in L, e_0) = w(r) \cdot d$
- $LCM(r \in L, e_{1 \le x \le d}) = w(r) \cdot x$
- $LCM(r \in L, w \ne e_{0 \le x \le d}) = \infty$
- $LCM(r \in I, w \in V) = MIN(LCM_s(r,w), LCM_{\neg s}(r,w))$
- $LCM_s(r \in I, w \in V) = \underset{w_l + w_r = w + e_{d(r)+1}}{MIN} (LCM(left(r), w_l) + LCM(right(r), w_r))$
- $LCM_{\neg s}(r \in I, w \in V) = \underset{w_l + w_r = w}{MIN} (LCM(left(r), w_l) + LCM(right(r), w_r))$

The first three formulas are the base cases. If a leaf has no incoming shortcut, its cost is simply its weight times its depth. If a leaf has an incoming shortcut, its cost is its weight times the depth of the shortcut. A leaf cannot have multiple incoming shortcuts because leaves must terminate all incoming shortcuts and, by Lemma 1, at most one shortcut is allowed to terminate at any node. We thus assign this illegal situation infinite cost.

The last three formulas represent the inductive case for non-leaves. We break the inductive case into two sub-cases and take the better of the two solutions. $LCM_s(r,w)$ is the case that r originates a shortcut, and $LCM_{\neg s}(r,w)$ is the case that r does not originate a shortcut. For each of these sub-cases we consider all possible partitions of throughput vector w or $w + e_{d(r)+1}$, recursively compute the optimal value for the subtree rooted at the children of r, and return the value corresponding to the best partitioning.

We compute the optimal solution by first computing the LCM function for leaves over all possible throughput vectors and then iterating up the tree towards the root over all possible throughput vectors. Once we have determined the optimal leafcut search tree cost we determine a set of leafcuts S_{LCM} that gives rise to this optimal cost in a top down fashion: at each node determine which shortcut choices yielded the optimal solution.

Theorem 2. *The LCM dynamic programming approach produces a leafcut tree with minimal cost.*

Proof. Suppose that S_{LCM}, the solution produced by the LCM algorithm, does not have minimal cost. There is some other selection of leafcuts S_{OPT} with strictly smaller cost. By Lemma 1 we assume that in S_{OPT} at most one shortcut terminates at any node.

Associated with the set of leafcuts S_{OPT} are throughput vectors w_r for each node r. By Lemma 6 $w_r \in V$. For internal r, at most one shortcut terminates at r and at most one shortcut originates at r. Therefore, either $w_r + e_{d(r)} = w_{left(r)} + w_{right(r)}$ (if r originates a shortcut) or $w_r = w_{left(r)} + w_{right(r)}$ (if r does not originate a shortcut). For leaves r, at most one shortcut terminates at r, and the cost of this leaf is simply $w(r)$ times the depth of this shortcut or times d if there is no such shortcut.

S_{OPT} is thus in the space of solutions scanned by the LCM algorithm. The LCM algorithm clearly computes a solution of minimal cost within the space of solutions it considers (it builds its solution from the bottom of T upwards by taking the best of all possible partitions). Therefore the cost of the solution returned by LCM is at most the cost of S_{OPT}, contradicting our assumption that S_{LCM} did not have minimal cost. □

Theorem 3. *The LCM dynamic program requires $O(n^2)$ space and $O(n^{1+log(3)})$ time.*

Proof. There are n nodes in the tree and, from Lemma 6, at most $|V| = n$ possible throughput vectors per node. For each node and each throughput vector we must store the value of the optimal solution. Hence, $O(n^2)$ space is used by the LCM dynamic program.

For each node, we consider all possible throughput vectors and all possible partitions of the throughput vector v and the modified vector $v + e_{d(r)}$. Over all throughput vectors in V, Lemma 3 states that there are 3^d possible partitions for the modified throughput vector. Thus, the LCM dynamic program uses $O(n^{1+log(3)})$ running time. □

Unfortunately, the leafcut model is not a very powerful model as an alternative to tree rebalancing for improving search cost. In fact, as Theorem 4 shows, the the search cost of an optimal leafcut search tree can be exponentially worse than the cost of a rebalanced tree. This drawback of the leafcut model is what motivates the general shortcut model, addressed in detail in Section 5.3.

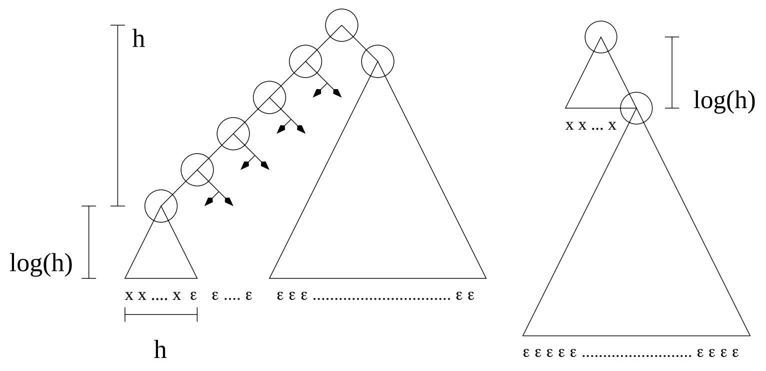
Theorem 4. *The cost of an optimal leafcut search tree can be exponentially worse than the cost of a rebalanced tree.*

Proof. We prove this result by example. Figure 3 shows an input tree of depth $d = h + \log(h)$ for which the optimal leafcut search tree has cost $\frac{h}{2} + O(\epsilon)$ while a rebalanced tree easily achieves cost $\log(h) + O(\epsilon)$. □

5.3 The General Shortcut Model

Here we present an optimal solution to the general shortcut search tree. This solution requires $O(n^3)$ space and $O(n^{1+log(\alpha)})$ time for $\alpha = \frac{27}{4}$. We adapt this solution from the solution to the leafcut model in Section 5.2, adjusted to allow for the termination of shortcuts at internal nodes and the larger space of throughput vectors. The correctness and efficiency of our adaption relies on the following lemma.

Lemma 7. *For a search tree T of depth d and any set of legal shortcuts S, the throughput vector of some edge e is an element of $W = \{w \mid w \in \mathbb{N}^d \wedge |w|_1 \leq d\}$.*



h
log(h)
x x .... x ε ε .... ε ε ε ε .... ε ε
h
log(h)
x x ... x
ε ε ε ε ε .... ε ε ε ε

Figure 3: We have $h \cdot 2^h$ leaves in a depth $d = h + \log(h)$ binary search tree. The $h-1$ leftmost leaves have uniform cost $x = \frac{1}{h-1} - O(\epsilon)$ and all the other leaves have cost ϵ. On the left we see that the optimal leafcut solution can do no better than adding leafcuts between the chain of $h-1$ nodes to reach the $h-1$ significant leaves in $1, 2, \ldots, h-1$ steps, for a total cost of $\frac{h}{2} + O(\epsilon)$. As shown on the right, however, if we are allowed to rebalance the tree we can reach all $h-1$ nodes in $\log(h)$ steps for a total cost of $\log(h) + O(\epsilon)$.

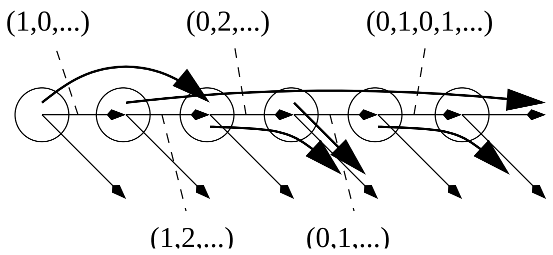


Figure 4: Shortcuts to internal nodes can result in non-binary throughput vectors. Here we show a section of a search tree T with shortcuts and the associated throughput vectors for a number of edges. We use "..." to indicate that the rest of a throughput vector is zero.

Proof. All nodes v have at most d ancestors and therefore the edge incident to v is used by at most d shortcuts. Therefore $|w_v|_1 \leq d$ and thus $w_v \in W$. Figure 4 shows an example in which a shortcut to an internal node can result in a non-binary throughput vector. (In actuality, throughput vectors in the general shortcut model are elements of a strict subset of W with cardinality $C_{d+1} = \frac{1}{d+2} \cdot \binom{2d+2}{d+1}$, the $(d+1)^{th}$ Catalan number. As evident from Lemma 4, however, the size of this subset is not significantly smaller than the size of W. We are therefore comfortable using the simpler set W.)

□

We compute our solution using a three dimension dynamic programming table GSM with $GSM(r \in T, w \in W, d' \in \{0, 1, \ldots, d\})$ denoting the optimal general shortcut search cost in the subtree rooted at r a distance $d_S(r) = d'$ from the root and with throughput vector $w_r = w$. The GSM function differs from the LCM function in three ways: throughput vector w now has a larger domain, the new parameter d' allows for $d_S(r)$ to be smaller than $d(r)$ because of internal shortcuts, and the GSM recurrence allows for shortcuts to terminate at internal nodes. The global optimal solution is given by $GSM(root(T), e_0, 0)$. We construct the optimal solution from the leaves upward using the following recurrence:

- $GSM(r \in L, e_0, d' \in \{0, 1, \ldots, d\}) = w(r) \cdot d'$
- $GSM(r \in L, e_{1 \leq x \leq d}, d' \in \{0, 1, \ldots, d\}) = w(r) \cdot min(x, d')$
- $GSM(r \in L, w \neq e_{0 \leq x \leq d}, d' \in \{0, 1, \ldots, d\}) = \infty$

- $GSM(r \in I, w \in W, d' \in \{0, 1, \dots, d\}) =$
  $MIN(GSM_{s \wedge \neg t}(r, w, d'), GSM_{\neg s \wedge \neg t}(r, w, d'), GSM_{s \wedge t}(r, w, d'), GSM_{\neg s \wedge t}(r, w, d'))$

- $GSM_{s \wedge \neg t}(r \in I, w \in W, d' \in \{0, 1, \dots, d\}) =$
  $\underset{w_l + w_r = w + e_{d'+1}}{MIN}(GSM(\mathit{left}(r), w_l, d' + 1) + GSM(\mathit{right}(r), w_r, d' + 1))$

- $GSM_{\neg s \wedge \neg t}(r \in I, w \in W, d' \in \{0, 1, \dots, d\}) =$
  $\underset{w_l + w_r = w}{MIN}(GSM(\mathit{left}(r), w_l, d' + 1) + GSM(\mathit{right}(r), w_r, d' + 1))$

- $GSM_{s \wedge t}(r \in I, w \in W, d' \in \{0, 1, \dots, d\}) =$
  $\underset{e_x + w_l + w_r = w + e_{x+1}}{MIN}(GSM(\mathit{left}(r), w_l, x + 1) + GSM(\mathit{right}(r), w_r, x + 1))$

- $GSM_{\neg s \wedge t}(r \in I, w \in W, d' \in \{0, 1, \dots, d\}) =$
  $\underset{e_x + w_l + w_r = w}{MIN}(GSM(\mathit{left}(r), w_l, x + 1) + GSM(\mathit{right}(r), w_r, x + 1))$

The first three formulas are once again the base case for leaves. For non-leaves we consider four subcases based on whether or not node r is the source of a shortcut and whether or not node r is the target of a shortcut. The optimal solution the the GSM algorithm is computed the same way it is computed for the LCM algorithm in Section 5.2. The following two theorems we present without proof. The proofs are very similar to the proofs of Theorem 4 and Theorem 3 given Lemmas 4, 5, and 7.

Theorem 5. *The GSM dynamic programming approach produces a shortcut tree with minimal cost.*

Theorem 6. *The GSM dynamic program requires $O(n^3 \cdot \sqrt{log(n)})$ space and $O(n^{1+log(\alpha)} \cdot log(n))$ time for $\alpha = \frac{27}{4}$.*

While allowing general shortcuts can substantially improve search cost it can make search much more difficult. Particularly, optimal search in a tree with general shortcuts cannot be done decentrally. The navigable shortcut model lets shortcuts target non-leaf nodes but in a fashion consistent with decentralized search (namely, shortcuts cannot cross each other). A dynamic program very similar to the LCM and GSM solutions finds a set of optimal navigable shortcuts but due to space constraints does not appear in this version of this paper.

6 Conclusions and Future Directions

Theorem 4 shows that optimal leafcut search trees can be exponentially worse than rebalanced binary search trees. A similar Theorem is needed to compare optimal general and navigable shortcut trees with rebalanced search trees. Also, it is worth studying the expected performance of optimal leafcut trees for random orderings of the leaf nodes.

Next, this paper considered only balanced binary trees. Efficient algorithms need to be found for arbitrary input search trees. Ideally, a technique is needed to find the optimal search tree for an ordered set of input weights where we are allowed to rebalance the tree and insert shortcuts.

Lastly, we have solved only the static version of the shortcut search tree problem. Now that this problem has a known efficient solution, we should explore the dynamic use of shortcuts in a search space with unknown or changing weights.

7 Acknowledgements

I would like to thank Dexter Kozen and Cornell University.

References

[1] William Pugh, *A Probabalistic Alternative to Balanced Trees, in* Workshop on Algorithms and Data Structures, 1990.

[2] D. D. Sleator and R. E. Tarjan, *Self-adjusting binary search trees, in* Journal of the ACM, 32:652-686, 1985.