Automated search for randomized algorithms for recursive 3-MAJ and 5-MAJ

Casey Mihaloew

July 27, 2015

Abstract

A methodical search for an efficient randomized algorithm for recursive 3-MAJ and 5-MAJ is introduced and implemented, with a new complexity for the hard distribution for 5-MAJ given as $O(4.4555^n)$. The techniques provided are general enough to work on other recursive evaluation problems with given distributions.

1 Introduction

The goal of this paper is to develop a computerized search to find efficient randomized algorithms for certain recursive evaluation problems, specifically recursive 3-MAJ and recursive 5-MAJ. This is accomplished by defining a class of algorithms that can be represented in the program and using increasingly good heuristics based on a hard distribution to determine which algorithm to choose.

The type of algorithms that are searched over are those that recursively evaluate descendants of a node in some order, possibly affected by the results of the previous evaluations, until the evaluation of that node can be determined. The simplest of these algorithms is just to evaluate all of the children and compute the value from those results, but there are of course some quick optimizations to make. To restrict it to a space that can be clearly defined and searched over, only nodes up to a predetermined depth will be queried. The algorithms will be interpreted as a series of states that an algorithm can be in, represented by a tree of depth d with some of the nodes filled in (evaluated) and some nodes unknown. For each state, there is a choice of which node to evaluate and depending on the answer goes to one of the two resulting states. The randomness comes in by randomly permuting the children of any newly seen node to avoid inputs more adversarial than the hard distribution. For the rest of the paper, it will be assumed that the input is in the hard distribution, which more work would need to be done to verify that it works for all distributions.

By treating each state as a problem to be solved, the actual search over the algorithms can be accomplished by dynamic programming assuming one can compute for each state which descendent to query next given results for the states with more nodes evaluated. For this paper, the choice is made based on a heuristic taking into account the probabilities that a given node evaluates to a given answer in the hard distribution and an estimate for the cost of querying a given node, which may be different for nodes at different depths or with different descendents already evaluated. Once all of these choices are made, the complexity of the algorithm can be computed by solving a (possibly large) system of linear recurrences, which not only gives the complexity but also a better estimate for the heuristic, as will be seen later.

Much work has been done on finding algorithms for 3-MAJ, and the current best known is stated in [1] with complexity $O(2.64644^h)$. Most of the work done in this paper is based on results from that work.

2 Preliminaries

2.1 Description of the states

The states that the algorithm of depth d are trees where each node is of depth at most d and has a value of 0, 1, or unknown, and an interior node has the value 0 or 1 iff that value can be determined using the value of its children; for example, if it has two children that are 1 and one that is unknown, its value would be 1. Additionally, it is required that each unknown leaf be at depth d so that the tree fully represents all of the descendents that haven't been evaluated yet.

The space of states that an algorithm can be in can be very large, even for fairly small depths. A few steps can be taken to ameliorate this. The first basic rule is never evaluate a node that does not affect the result. For the case of 3-MAJ, this amounts to not evaluating the third child of a node if the other two are known to be the same. This is represented in the states by removing all of the children from a node whose value is known from the tree. There are also some states that are equivalent by symmetry given that the ordering of the children is arbitrary, which can give a large reduction in the number of states. Even with these reductions, the number of states is superexponential, leaving the search only feasible up to depths two or three.

2.2 Description of the subroutines

The key to how this algorithm works is the use of numerous mutually recursive subroutines. The subroutines can be identified by states of depth strictly less than d corresponding to subtrees of a state of depth d. It is important to note that each of these subtrees can be treated as a depth d tree with respect to evaluation by adding three unknown children to every unknown leaf until it is a tree of depth d. Since the search will calculate what subtree to evaluate for every state of depth d, the algorithms for each of these subroutines will be computed along the way.

2.3 Probabilities of the states

For each state, the probability that it evaluates to 1 given the hard distribution can be calculated recursively from the probabilities of its children. If p_1, p_2, and p_3 are the probabilities for the children, then the probability for the state is

$$\frac{p_1p_2(1-p_3)+p_1(1-p_2)p_3+(1-p_1)p_2p_3}{1-p_1p_2p_3-(1-p_1)(1-p_2)(1-p_3)}$$

where the numerator is the probability that the tree evaluates to 1 and is in the hard distribution (two 1's and one 0) and the denominator is the probability that the node is in the hard distribution (not all three the same).

2.4 Probabilities of the transitions

When evaluating a subtree, it will be useful to know what the probability is that the subtree evaluates to a certain value given that the whole tree evaluates to a certain value. To get that formula, let T be the tree, S be the subtree, and T_i be the resulting tree after S is evaluated to i, so the probability we need to calculate is $\Pr[S = j|T = i]$. By Bayes' Theorem, $\Pr[S = j|T = i] = \Pr[T = i|S = j] \Pr[S = j]/\Pr[T = i] = \Pr[T_j = i] \Pr[S = j]/\Pr[T = i]$. By letting $i = 1$ in the previous equation and summing over j,

$\Pr[T=1,S=0]+\Pr[T=1,S=1]=\Pr[T_0=1]\Pr[S=0]+\Pr[T_1=1]\Pr[S=1]$ or $\Pr[T=1]=\Pr[T_0=1](1-\Pr[S=1])+\Pr[T_1=1]\Pr[S=1]$. Solving for $\Pr[S=1]$ gives

$$\Pr[S=1]=\frac{\Pr[T=1]-\Pr[T_0=1]}{\Pr[T_1=1]-\Pr[T_0=1]}$$

Therefore,

$$\Pr[S=j|T=i]=\frac{\Pr[T_j=i](\Pr[T=1]-\Pr[T_0=1])}{\Pr[T=i](\Pr[T_1=1]-\Pr[T_0=1])}$$

3 Algorithm

For a given depth d, calculate all possible states, call this set S_d. For each state in S_d, the goal is to calculate three things: an estimate for how long it would take to evaluate given that the answer is zero, an estimate for how long it would take to evaluate given that the answer is one, and an array containing the expected number of times each other subroutine is called. These values are calculated by choosing the best subtree to evaluate according to some heuristic and are just weighted sums of the values from the two possible resulting trees plus the adjustment for the evaluation at the current state, where the weightings are from the probabilities given in section 2.4. In order to take advantage of dynamic programming, the order in which these states are considered can either be as needing using memoization or in order of closest to being computed back to the initial state where nothing is known. Once these values are computed, the system of linear recurrences can be extracted from the arrays and solved to give a value for the base of the exponent in the running time of the algorithm and to give new estimates for the constant for each subroutine. These new values can be fed back into the program and repeated as many times as desired or until convergence.

3.1 Heuristic

The heuristic for choosing which subtree to evaluate is which one has the smallest expected value given the hard distribution and approximations for how long it would take to evaluate said subtree. To accomplish this, each tree will have an expected time to complete the evaluation given that the answer is 0 or 1 and the approximations will be predetermined before every iteration of the algorithm. To get the formula to do so, let T be the tree, S be the subtree, and T_i be the resulting tree after S is evaluated to i. As notation, let $E_{X,i}$ be the expected time to evaluate tree X to i. The expected time is

$$\sum_{i,j\in\{0,1\}\times\{0,1\}}\Pr[T=i,S=j](E_{S,j}+E_{T_j,i})$$

The values of $E_{T_j,i}$ are known from dynamic programming and $E_{S,j}$ is from the approximations. The other probability can be calculated using section 2.4 as $\Pr[T=i,S=j]=\Pr[S=j|T=i]\Pr[T=i]$ It is also necessary to calculate $E_{T,i}$ using

$$E_{T,i}=\sum_{j\in\{0,1\}}\Pr[S=j|T=i](E_{S,j}+E_{T_j,i})$$

One important thing to note about these equations is that they are all linear in the approximate terms, $E_{S,j}$, so multiplying all of these approximations by a constant will make no difference in the choices made by the algorithm, so it doesn't matter how deep the tree is in terms of designing the algorithm.

3.2 System of linear recurrences

To calculate the expected running time of the generated algorithm, each subroutine can be treated as two functions from the height of the original tree to the running time, one for each value the whole tree can evaluate to, and by keeping track of how many times each subroutine is expected to be called at a smaller height for each subroutine, a system of linear recurrences can be generated. The solution to the system of recurrences is simplified by the fact that we only need an asymptotic answer, so only the largest eigenvalue of the matrix for the recurrences needs to be considered. The corresponding constant terms for the asymptotic solution are the values in the eigenvector multiplied by some global constant, but since these values are only used in the heuristic for the next run of the program and the heuristic is independent of multiplication by a global constant, a normalized eigenvector suffices.

4 Results

The algorithm was implemented and run for both 3-MAJ and 5-MAJ. In the 3-MAJ case, depths of 2 and 3 were the only feasable value. Depth 2 gave the same algorithm and analysis as was previously known. Somewhat surprisingly, depth 3 gave no improvement. While this does not mean that there is definitely no better algorithm of this type at this depth or higher, it certainly indicates that the known algorithm is a rather good algorithm. 5-MAJ was run on depth 2 and gave a running time of about 4.4555^h, better than the basic depth 1 algorithm, randomly picking children until you get three of the same value, with running time 4.5^h. None of these took more than a few seconds to run and converged in no more than four iterations.

5 Conclusions

While no new bound was found for 3-MAJ, an upper bound for 5-MAJ was given that is new as far as I could tell. Additionally, most of the techniques mentioned are general enough that efficient algorithms could be found for other recursive evaluation problems given a hard distribution and some amount of symmetry. Additional work could be done to try and remove the reliance on the knowledge of a hard distribution by calculating it on the fly and it is possible that greater depths could be searched by expoiting some other pruning mechanism besides basic symmetry. Ideally, this line of thought could provide examples of good algorithms in order to find a pattern that could lead to a very good or optimal algorithm for one of these problems that doesn't necessarily follow the same template.

6 References

[1] Magniez, P., Nayak, A., Santha, M., Sherman, J., Tardos, G., and Xiao, D. 2013 Improved bounds for the randomized decision tree complexity of recursive majority, arXiv:1309.7565v1

A Java source code for 3-MAJ

```java
public static final int H=3; // Depth to search
public static final Trie[] trs=new Trie[H]; // Ids for subroutines
public static final int[] total=new int[H+1]; // Number by depth
public static int S=0;
public static final Tree FALSE=new Tree(0, 1), TRUE=new Tree(1, 2);

public static void main(String[] args) {
        // Set up depth zero case
        List<Tree> l0=new ArrayList<Tree>();
        l0.add(new Tree(0.5, 0)); l0.add(FALSE); l0.add(TRUE);
        List<List<Tree>> store=new LinkedList<List<Tree>>();
        total[0]=3;
        // Set up the rest of the trees
        for(int i=0; i<H; i++) {
                S+=l0.size()-2;
                trs[i]=new Trie(l0.size());
                store.add(l0);
                l0=expand(l0, trs[i]);
                total[i+1]=l0.size(); }
        Tree root=l0.get(0);
        // Initialize costs arbitrarily
        double[] cost=new double[2*S];
        for(int i=0; i<2*S; i++)
                cost[i]=Math.random();
        double[][] mat=new double[2*S][2*S];
        // Give unique identifiers to each subroutine at each detph
        int add=total[H-1]-2;
        for(int i=H-2; i>=0; i--) {
                for(Tree t: store.get(i)) {
                        int id=t.id;
                        if(id==1||id==2) continue;
                        if(id>2) id-=2;
                        id+=add;
                        Tree help=t.getCorrespondingTree(i);
                        int id2=help.id;
                        if(id2>2) id2-=2;
                        id2+=add-total[i+1]+2;
                        mat[2*id][2*id2]=1; mat[2*id+1][2*id2+1]=1; }
                add+=total[i]-2; }
        // Iterate the search some number of times
        for(int times=0; times<20; times++) {
                root.setUpRecurse(cost); add=0;
                for(Tree t: store.get(H-1)) {
                        int id=t.id;
                        if(id==1||id==2) continue;
                        if(id>2) id-=2;
                        id+=add;
                        Tree help=t.getCorrespondingTree(H-1);
                        mat[2*id]=help.recurse0;
                        mat[2*id+1]=help.recurse1; }
                // Solve the recurrence
                double e=bigEigenvalue(mat, cost);
                System.out.println(e);
                for(Tree t: l0) {
                        t.recurse0=null; t.recurse1=null; }
        }
}

// From trees at depth i, calculate trees at depth i+1
public static List<Tree> expand(List<Tree> st, Trie tr) {
        List<Tree> ans=new ArrayList<Tree>();
        ans.add(new Tree(st.get(0), st.get(0), st.get(0), 0));
        tr.add(ans.get(0), 0, 0, 0);
        ans.add(st.get(1));
        ans.add(st.get(2));
        int on=3;
        // Iterate over triples, taken in nondecreasing order
        // to take advantage of symmetry
        for(int i=0; i<st.size(); i++) {
        for(int j=i; j<st.size(); j++) {
        for(int k=j; k<st.size(); k++) {
                // States already added
                if(i==0&&j==0&&k==0) continue;
                if(j==1&&k==1) tr.add(FALSE, i, j, k);
                else if(j==2&&k==2) tr.add(TRUE, i, j, k);
                else if(i==1&&j==1) tr.add(FALSE, i, j, k);
                else if(i==2&&j==2) tr.add(TRUE, i, j, k);
                else { // A new state
                        Tree temp;
                        ans.add(temp=new Tree(st.get(i), st.get(j),
                                st.get(k), on));
                        tr.add(temp, i, j, k); on++; }
        }}}
        return ans;
}

public static class Tree {
        Tree a, b, c; double p; int id;
        // Expected number of times each subroutine is run
        double[] recurse0, recurse1;
        // Values for the heuristic
        double expect0=0, expect1=0;
        Move best;

        public Tree (double d, int id_) {
```

```
        p=d; id=id_; }

public Tree (Tree a_, Tree b_, Tree c_, int id_) {
        a=a_; b=b_; c=c_;
        // Calculate probability
        double s1=a.p+b.p+c.p;
        double s2=a.p*b.p+a.p*c.p+b.p*c.p;
        double s3=a.p*b.p*c.p;
        p=(s2-3*s3)/(s1-s2); id=id_; }

public void setUpRecurse(double[] cost) {
        if(recurse0!=null) return; // Return if already computed
        recurse0=new double[2*S]; recurse1=new double[2*S];
        if(id==1||id==2) return;
        List<Move> moves=getMoves(H);
        best=null; double bestE=Double.POSITIVE_INFINITY;
        // Loop over all possible moves to find the best one
        for(Move m: moves) {
                m.target0.setUpRecurse(cost);
                m.target1.setUpRecurse(cost);
                double ps=(m.target0.p-p)/(m.target0.p-m.target1.p);
                double p0=ps*(1-m.target1.p)/(1-p);
                double p1=ps*(m.target1.p)/(p);
                double temp0=p0*(m.target1.expect0+cost[m.id*2+1])+
                        (1-p0)*(m.target0.expect0+cost[m.id*2]);
                double temp1=p1*(m.target1.expect1+cost[m.id*2+1])+
                        (1-p1)*(m.target0.expect1+cost[m.id*2]);
                double temp=p*temp1+(1-p)*temp0;
                if(temp<bestE) {
                        bestE=temp; expect0=temp0;
                        expect1=temp1; best=m; } }
        // Update the values with the best move
        double ps=(best.target0.p-p)/(best.target0.p-best.target1.p);
        double p0=1-ps*(1-best.target1.p)/(1-p);
        double p1=ps*(best.target1.p)/(p);
        for(int i=0; i<2*S; i++) {
                recurse0[i]=(1-p0)*best.target1.recurse0[i]+p0
                                *best.target0.recurse0[i];
                recurse1[i]=p1*best.target1.recurse1[i]+(1-p1)
                                *best.target0.recurse1[i]; }
        recurse0[best.id*2+1]+=1-p0; recurse0[best.id*2]+=p0;
        recurse1[best.id*2+1]+=p1; recurse1[best.id*2]+=1-p1;
}

private List<Move> getMoves(int h) { // Get possible moves
        if(a==null) { // Only for depth 0 unknown
```

```
                        List<Move> ans=new LinkedList<Move>();
                        if(id==0) ans.add(new Move(0, FALSE, TRUE));
                        return ans; }
                List<Move> temp=new LinkedList<Move>();
                // Add an evaluation only if this is a subtree
                if(h<H&&id!=1&&id!=2)
                        temp.add(new Move((id==0?0:id-2), FALSE, TRUE));
                // Add the moves for all three children
                List<Move> temp2=a.getMoves(h-1);
                for(Move m: temp2) {
                        m.target0=trs[h-1].get(m.target0.id, b.id, c.id);
                        m.target1=trs[h-1].get(m.target1.id, b.id, c.id);
                        if(h<H) m.id+=total[h]-2; }
                temp.addAll(temp2); temp2=b.getMoves(h-1);
                for(Move m: temp2) {
                        m.target0=trs[h-1].get(m.target0.id, a.id, c.id);
                        m.target1=trs[h-1].get(m.target1.id, a.id, c.id);
                        if(h<H) m.id+=total[h-1]-2; }
                temp.addAll(temp2); temp2=c.getMoves(h-1);
                for(Move m: temp2) {
                        m.target0=trs[h-1].get(m.target0.id, b.id, a.id);
                        m.target1=trs[h-1].get(m.target1.id, b.id, a.id);
                        if(h<H) m.id+=total[h-1]-2; }
                temp.addAll(temp2);
                return temp; }

        public Tree getCorrespondingTree(int h) {
                if(a==null) {
                        if(id==0) return trs[h].get(0, 0, 0);
                        else return this; }
                Tree ta=a.getCorrespondingTree(h-1);
                Tree tb=b.getCorrespondingTree(h-1);
                Tree tc=c.getCorrespondingTree(h-1);
                return trs[h].get(ta.id, tb.id, tc.id); }
}

// Stores information for a single transition
public static class Move {
        int id; Tree target0, target1;

        public Move (int a, Tree b, Tree c) {
                id=a; target0=b; target1=c; } }

public static class Trie {
        int n; Tree ans; Trie[] next;
        public Trie (int n_) { n=n_; }

        public void add(Tree v, int... key) { addH(0, v, key); }

        private void addH(int on, Tree v, int[] key) {
                if(on==key.length) {
                        ans=v; return; }
                if(next==null) next=new Trie[n];
                if(next[key[on]]==null) next[key[on]]=new Trie(n);
                next[key[on]].addH(on+1, v, key); }

        public Tree get(int... key) {
                Arrays.sort(key); return getH(0, key); }

        private Tree getH(int on, int[] key) {
                if(on==key.length) return ans;
                return next[key[on]].getH(on+1, key); }
}

// Uses power iteration to solve the recurrence
public static double bigEigenvalue(double[][] m, double[] w) {
        double[] t=new double[w.length];
        double ans=0;
        for(int times=0; times<100; times++) {
                matHit(m, w, t); ans=0;
                for(int i=0; i<w.length; i++) ans+=t[i]*t[i];
                ans=Math.sqrt(ans);
                for(int i=0; i<w.length; i++)  dw[i]=t[i]/ans;
        }
        matHit(m, w, t); double num=0; double den=0;
        for(int i=0; i<w.length; i++) {
                num+=t[i]*w[i]; den+=w[i]*w[i]; }
        return num/den; }

public static void matHit(double[][] m, double[] in, double[] out) {
        for(int i=0; i<out.length; i++) out[i]=0;
        for(int i=0; i<out.length; i++)
                for(int j=0; j<out.length; j++)
                        out[i]+=m[i][j]*in[j];
        }
```