Modern Decision Tree Optimization with Generalized Optimal Sparse Decision Trees

Duke Course Notes

Cynthia Rudin

CART and C4.5 were built for computers of the 1980’s and 1990’s, which are completely different than modern computers. We can do a lot more now, and we do not need to rely on top-down greedy tree induction methods like CART and C4.5.

I will be talking about the Generalized Optimal Sparse Decision Trees (GOSDT) algorithm of Lin et al. [2020], McTavish et al. [2022]. These algorithms are the current fastest algorithms for optimal and approximately optimal sparse decision tree optimization at the time of this writing. GOSDT is pronounced “ghost.” GOSDT uses dynamic programming with bounds that prevent it from exploring the whole search space of possible trees. I’ll present a simplified version of GOSDT here.

Before we start, I’ll point out that GOSDT is very powerful. Even for challenging datasets like the FICO Explainable Machine Learning Challenge Data [FICO et al., 2018], it yields single sparse trees about as accurate as boosted decision trees. An example of such a tree on this dataset is below.

```
<table>
<thead>
<tr>
<th>ExternalRiskEstimate ≤ 67.5</th>
<th>True</th>
<th>False</th>
</tr>
</thead>
<tbody>
<tr>
<td>class</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1  PercentTradesWBalance ≤ 73.5</td>
<td>class</td>
<td></td>
</tr>
<tr>
<td>AverageMinFile ≤ 63.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSinceMostRecentInqexcl7days ≤ 0.5</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>MSinceMostRecentInqexcl7days ≤ 70.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSinceMostRecentInqexcl7days ≤ -7.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AverageMinFile ≤ 63.5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Formulation

GOSDT minimizes an objective function:

\[ R(\text{tree}, X, Y) = \ell(\text{tree}, X, Y) + \lambda \cdot \#\text{leaves}(\text{tree}) \text{ such that depth}(\text{tree}) \leq D, \quad (1) \]
where loss is misclassification error:

\[ \ell(\text{tree}, \mathbf{X}, \mathbf{Y}) = \frac{1}{n} \sum_{i=1}^{n} \mathbb{I}_{y_i \neq \hat{y}_{i,\text{tree}}}. \]

\( R \) has both a hard constraint and a soft constraint on sparsity. Either \( \lambda \) or \( D \) will make the optimal tree sparse, but using \( D \) alone with \( \lambda = 0 \) won’t get to the smallest tree. This is because the optimal sparse tree might look like this:

![Diagram of a sparse tree]

If you used \( \lambda = 0 \), the algorithm would have produced a complete tree of depth \( D \) (which has 8 leaves) instead of this lovely sparse tree with only 5 leaves.

On the other hand, if you used \( \lambda > 0 \) alone (eliminating the constraint on \( D \)), then computation is much more expensive. Why is that? Let’s consider how the space of trees grows for a specific number of features. I did some combinatorics to calculate the number of possible trees for number of features \( p = 20 \):

\[
\begin{align*}
\text{The number of trees with depth} & \leq 3 \text{ is } 9.4 \times 10^8 \\
& \leq 4 \text{ is } 9.2 \times 10^{28} \\
& \leq 5 \text{ is } \text{“Inf”}.
\end{align*}
\]

Even if the optimal tree has depth 3, since you don’t know this in advance, you might need to search depths 4, 5, and 6 to prove your depth 3 tree is optimal. On the other hand if you had set \( D = 4 \), you would never need to search depth 5 trees at all.

That’s why it’s useful to have both the soft constraint using \( \lambda \) (to ensure a sparse tree), and the hard constraint using \( D \) (which saves computation if we can choose \( D \) correctly).
Binarization Preprocessing

We need to do some preprocessing to binarize the data. Essentially, we will enumerate the possible splits of a decision tree as binary features.

Our training set is \( \{ (x_i^{\text{orig}}, y_i) \}_{i=1}^n \). We transform each continuous feature into a collection of binary features. Say that \( x_{cts} \) has realized values in the dataset of 3.2, 4.1, 6.8, 7.0, and 8.4. In other words, there exist datapoints in our dataset where \( x_{cts} \) attains each of these values. Now we find the midpoints between any two of these values and create an indicator variable.

So we create variables: \( \mathbb{1}_{x_{cts} \leq 3.65}, \mathbb{1}_{x_{cts} \leq 5.45}, \mathbb{1}_{x_{cts} \leq 6.9}, \mathbb{1}_{x_{cts} \leq 7.7}, \) etc. We replace \( x_{cts} \) with these binary variables and use the binarized data as our data matrix \( X \).

That process does create a lot of binary variables, so an alternative is to run an algorithm like random forests or boosted decision trees, and only use splits that were used in these algorithms. For instance, it is possible that random forest never split on “age<25.” In that case, we can omit the age<25 variable from our preprocessed set of variables.

The Key Ingredients for GOSDT

GOSDT uses dynamic programming. In dynamic programming, we solve sub-problems, and those give us the solution to the overall problem. For decision trees, we need to find the optimal root split. But you don’t know what the optimal split for the root is until you find the optimal splits for the nodes just beneath it. And you don’t know the optimal splits for those until you do the ones beneath them! Luckily, this doesn’t go on forever, because you can often prove that the optimal solution is a leaf node when you have only a few data points.
Raw Ingredient 1: Represent each subproblem by its data, as a bit-vector

Each subproblem is indexed by a bit-vector describing the data we need to use for the subproblem. The master problem is:

\[ s = [1, 1, 1, 1, \ldots, 1] \leftarrow \text{master problem involving all data} \]

This is a bit-vector of size \( n \), and each element is 1 because it is involved in the master problem at the root of the tree. When we solve the master problem, we are done with GOSDT. Here is another subproblem. Perhaps it arose from splitting the master problem on a variable, which produces two subproblems, and this is the data from one of them.

\[ s = [0, 1, 1, 0, 1, 1, \ldots] \leftarrow \text{problem using data points 2, 3, 5, 7, 8,\ldots} \]

Choosing to index subproblems by their bit-vector representations has a lot of advantages: bit-vector comparisons are really fast. This allows us to quickly see if two subproblems are identical. The bit-vector representation also helps us because it gives a unique way to identify a subproblem that doesn’t depend on how we arrived at it. We can easily arrive at the same subproblem in different ways. For instance, a subproblem could have arisen by splitting on the tenth feature and then the twelfth, but it could also have arisen by splitting by the twelfth feature and then the tenth. Or, if the tenth and eleventh features are similar, it’s possible that splitting on the tenth feature instead of the eleventh feature somewhere in the tree could yield the same subproblem. With the bit-vector indexing, if you solve the subproblem once, you can use its solution every time you see that subproblem again.

Raw Ingredient 2: Priority Queue

This orders the subproblems.

\[ Q = [101101111000101\ldots], [10010100100100100\ldots], [\ldots], [\ldots] \ldots \]

The algorithm will handle problems according to their priority in the queue.

Raw Ingredient 3: Dependency Graph

The dependency graph \( G \) stores all the subproblems, their relationships (e.g., parent-child relationships), and lower and upper bounds on the objective for
that subproblem. Below is a dependency graph for a small problem that we’ll work out later.

At the top is the master problem $[1,1,1,1,1,1]$ containing all 6 data points (it’s a small data set). There are only two features, which are both indicator variables. One is the indicator that $f1<0.6$ and the other is the indicator for whether $f2<0.3$. There are three possibilities for the top of the tree: turn the whole tree into a leaf (on the left), split at $f1<0.6$ (center), or split at $f2<0.3$ (right). Let’s say we traverse the graph along the option for $f1<0.6$, where we hit a white circle, which is the actual root node in the tree where we make that split. Going to the left, where $f1<0.6$, we have two options for splitting: turning into a leaf (continuing to the left), or splitting on $f2<0.3$ (continuing to the right). If we split on $f2<0.3$ we get to a white node for that split, and then that split leads to two leaves.

The data in a child subproblem is a strict subset of the data from any of its parent or ancestor problems. (You can see that by comparing the bit vectors.) You can also see a case where a child has two parents, which is the subproblem involving only the first three data points: $[1,1,1,0,0,0]$. The upper bounds and lower bounds for each subproblem are marked by $p.ub$ and $p.lb$, but don’t pay attention to their values quite yet. A subproblem (or the master problem) is solved when its upper and lower bounds are equal.

We can extract trees from the dependency graph once we’ve finished building it.
Below, we place three red circles on the graph showing the leaves of a tree. The tree is on the right. Its splits are the two white circles leading to the three red circles.

When the master problem is completed, its upper and lower bounds will be the same, and at that point, we can read off the optimal tree from the graph. Let us pretend that the upper and lower bound are both 3$\lambda$ for the master problem in the graph when solved. Now, we look at the upper/lower bounds for the children (all upper and lower bounds will all be the same after GOSDT’s computation is complete). We know that the optimal solution gives bound 3$\lambda$, so we need to find children that give us this value. For the option of splitting on $f_1 < 0.6$, we add up the 2$\lambda$ from the left subproblem and the $\lambda$ from the right subproblem and get 3$\lambda$, so we know that there is an optimal tree with $f_1 < 0.6$ as the top split. Interestingly, if we instead consider the root split on $f_2 < 0.3$, the left subproblem gives $\lambda$ and the right will give 2$\lambda$, so there is also an optimal tree with this root split. In other words, the optimal sparse tree is not unique! In fact, we could have a root split with $f_2 < 0.3$, that has a split on $f_1 < 0.6$ below it, in addition to the tree in the figure above; they are both optimal.

**Initialization**

Here is the initialization for GOSDT:

1: $Q = \emptyset$  
2: $G = \emptyset$  
3: $s_0 = [1, 1, 1, \ldots, 1]$  

$\triangleright$ empty priority queue  
$\triangleright$ empty dependency graph  
$\triangleright$ master problem, a bit vector of n ones
4: $p_0 = \textsc{FindOrCreateNode}(G,s_0) \triangleright \text{place the master problem into dependency graph}$

Before continuing onto the GOSDT algorithm, we must explain the subroutine $\textsc{FindOrCreateNode}$.

**subroutine $\textsc{FindOrCreateNode}$**

The subroutine $\textsc{FindOrCreateNode}(G,s)$ checks to determine whether subproblem $s$ is already in $G$, and if not, it places $s$ into $G$ and initializes its upper and lower bounds. I’ll place it below, and explain it afterwards.

1: function $\textsc{FindOrCreateNode}(G,s)$

trace:  
2:  \textbf{if} $G.\text{find}(s) == \text{NULL}$ \triangleright Here, we used bit-vector comparisons to see whether $s$ is identical to a subproblem already in $G$, and if it’s not, we proceed.
3:  \hspace{1em} $p.\text{id} \leftarrow s$ \triangleright identify problem $p$ by bit-vector $s$
4:  \hspace{1em} $p.\text{lb} = 0+2\lambda$ \triangleright maybe I could get to 0 loss if I make some splits
5:  \hspace{1em} $p.\text{ub} = \frac{1}{n} \text{(\# minority labels in $s$)} + \lambda$ \triangleright objective if it was a leaf
6:  \hspace{1em} \textbf{if} $p.\text{ub}−p.\text{lb} \leq 0$ \triangleright in this case, trivial tree is optimal tree
7:  \hspace{2em} $p.\text{lb} \leftarrow p.\text{ub}$ \triangleright make it a leaf
8:  \hspace{1em} $G.\text{insert}(p)$ \triangleright insert $p$ into $G$
9:  \hspace{1em} return $\text{Find}(G.\text{find}(s))$ \triangleright return a pointer to location of $s$ in $G$

The “$G.\text{find}(s)$” step means that we compare $s$ to every subproblem in $G$ using bit-vector comparisons. The bit-vector representation of the subproblems is the key to making this step very computationally fast. If $s$ is already in $G$, we just return its location in $G$. If $s$ is not in $G$, we need to calculate its bounds and insert it, and return its location. There are three interesting parts we have not explained yet:

**Lower Bound $p.\text{lb}$:** The lower bound sets the loss to 0, on the possibility that if we keep splitting, the loss of the subproblem might eventually become 0. We’ll keep at least $2\lambda$ for our regularization term since we have at least 2 leaves for this subproblem.

**Upper Bound $p.\text{ub}$:** The upper bound we give when we create the node is just the value of the objective if we had made the subproblem into a leaf. If we did that, the number of points we would classify incorrectly is the minority label.
Let’s say we have 23 points in the leaf, with 12 positive and 11 negative points. In that case, the majority vote is positive, and the number of minority points is 11. We’ll misclassify 11 points, and the loss is $11/n$, where $n$ is (still) the number of points in the dataset. The regularization term is $\lambda$ times 1.

**The “if” Condition:** The condition $p.ub - p.lb \leq 0$ arises from the following theorem. We use notation $X_{s,.}, Y_s$ to indicate that we use only the observations with value 1 in bitvector $s$ (i.e., the points in the subproblem).

**Theorem 1** If $p.ub - p.lb \leq 0$, then

$$R(\text{trivial tree}, X_{s,.}, Y_s) \leq R(\text{any child tree}, X_{s,.}, Y_s).$$

In other words, if the upper bound and lower bound differ by less than 0, the trivial tree (that is just a leaf) is just as good as any child tree. In that case, there’s no point splitting, because it’s not going to reduce the objective. Here is the proof:

**Proof.** Any child tree must have at least 2 leaves from splitting a node. In the best case it will have 0 misclassification error.

$$R(\text{any child tree}, X_{s,.}, Y_s) \geq \ell(\text{any child tree}, X_{s,.}, Y_s) + 2\lambda \geq 0 + 2\lambda. \quad (2)$$

A trivial tree (with 1 leaf) obeys:

$$R(\text{trivial tree}, X_{s,.}, Y_s) = \frac{1}{n} (\#\text{minority labels}) + \lambda = p.ub,$$

where this is the definition of $p.ub$, which comes from the earlier steps in **FindOrCreateNode** before the “if” condition. Thus,

$$p.ub - p.lb = \frac{1}{n} (\#\text{minority labels}) + \lambda - 2\lambda$$

$$= \frac{1}{n} (\#\text{minority labels}) - \lambda$$

$$\leq 0 \quad \text{(this uses } p.ub - p.lb \leq 0 \text{ by assumption).}$$

Thus,

$$R(\text{trivial tree}, X_{s,.}, Y_s) = \frac{1}{n} (\#\text{minority labels}) + \lambda \leq \lambda + \lambda = 2\lambda$$

$$\leq R(\text{any child tree}, X_{s,.}, Y_s) \quad \text{(from (2)).}$$

8
Thus, the trivial tree is as good or better than any of its children. □

The theorem reveals that if the condition $p.ub - p.lb \leq 0$ holds, then an optimal solution to the subproblem is a tree that is just a leaf (a trivial tree). When that happens, the lower bound and upper bound are just the objective for the trivial 1-leaf tree.

Next, I’ll present a shortened version of GOSDT’s main while loop.

**The Main While Loop (Short Version)**

```
1: while $p_0.lb \neq p_0.ub$ do ▷ master problem not solved
2:     $s \leftarrow Q.pop$ ▷ index of problem to work on, pop it from the queue
3:     $p \leftarrow G.find(s)$ ▷ find it in $G$
4:     if $p.ub = p.lb$ then continue (problem solved) ▷
5:     \[ \sqcup \] ▷ go back to top of while loop and choose the next subproblem ▷
6:     Loop over $j$. For each $j$, split on feature $j$ to form 2 subproblems, get their lb and ub, and put both of them in $G$ if they aren’t there already.
7:     Update bounds for $p$ and its parents ▷
8:     \[ \sqcup \] ▷ Check again if we’re done:
9:     if $p.lb = p.ub$ then problem solved, continue ▷ go back to top of while loop
10: Loop over $j$ again. For each $j$ split on $j$ to form 2 subproblems again, evaluate bounds for all the children and enqueue only the viable ones.
11: return $G.find(s)$ ▷ Return a pointer to location of $s$ in $G$
```

The main while loop first gets a subproblem $p$ from the front of the priority queue $Q$. If it’s already solved, we don’t need to do anything else so we just go back and get another subproblem. If it’s not already solved, we split it into 2 child problems in all possible ways, and add all of the newly created subproblems to $G$. We also calculate their lower and upper bounds.

We use these bounds from the child problems to update the bounds for $p$ and pass those bounds up to its parents. To update the lower bound for $p$, we calculate the following, where $p^l_i$ and $p^r_i$ are the left and right subproblems after we split problem $p$ on feature $j$:

$$\text{argmin}_j \ p^l_i . lb + p^r_i . lb$$

The $j$ with the lowest value of this lower bound is the one we are most curious
about, because this subproblem could potentially lead to a low loss. Similarly, we want to choose the best upper bound, which is the split that would give us the lowest known value of the loss:

$$\text{argmin}_j p^j_{l}.\text{ub} + p^j_{r}.\text{ub}.$$ 

(It is possible that just keeping $p$ as a leaf gives the lowest lower bound, in which case we leave it as a leaf.)

If we do make an update to either the lower or upper bounds, we need to pass that information upwards through the graph, to $p$’s ancestors, to get us closer to solving the master problem. We put the parents of $p$ in the front of the priority queue so their bounds get updated first – hopefully, that way, we will discover less child problems to explore.

Then, we loop again over $j$ and get all of the child subproblems, but this time we evaluate whether to put them into the queue. There are two reasons we might not enqueue them. First, if the subproblem is already solved (i.e., $p.\text{lb}=p.\text{ub}$) we do not need to enqueue it. To enqueue it, the lower bound should be strictly less than the upper bound. Second, if the sum of lower bounds from the subproblem’s two children is higher than the subproblem’s upper bound, our subproblem can’t be in any optimal solution to the problem. In that case, we proved that we are better off with what we had before trying to make that split.

Let’s give the pseudocode for the main while loop with a bit more detail.
The Main While Loop (Regular Version)

1: \textbf{while} \( p_0.\text{lb} \neq p_0.\text{ub} \) \textbf{do} \hspace{1cm} \triangleright \text{master problem not solved} \\
2: \hspace{1cm} s \leftarrow Q.\text{pop} \hspace{1cm} \triangleright \text{index of problem to work on} \\
3: \hspace{1cm} p \leftarrow G.\text{find}(s) \hspace{1cm} \triangleright \text{find it in G} \\
4: \hspace{1cm} \textbf{if} \ p.\text{ub}=p.\text{lb} \ \textbf{then} \ \textbf{continue} \hspace{1cm} \triangleright \text{problem solved, go back to while loop and choose next subproblem} \\
5: \hspace{1cm} (\text{lb}',\text{ub}') \leftarrow (\infty, \infty) \hspace{1cm} \triangleright \text{loose starting bounds} \\
6: \hspace{1cm} \textbf{for} \ j=1, \ldots \ # \ \text{features} \ \textbf{do} \hspace{1cm} \triangleright \text{create left and right subproblems for each j} \\
7: \hspace{2cm} s_l, s_r \leftarrow \text{split s on feature j} \hspace{1cm} \triangleright \text{left and right subproblems} \\
8: \hspace{2cm} p^j_l \leftarrow \text{FindOrCreateNode}(G,s_l) \hspace{1cm} \triangleright \text{Put them in G} \\
9: \hspace{2cm} p^j_r \leftarrow \text{FindOrCreateNode}(G,s_r) \hspace{1cm} \triangleright \text{calculate bound for j by adding up its left and right child pairs.} \\
10: \hspace{2cm} \triangleright \text{Among all j, find lowest lower bound and the lowest upper bound.} \triangleleft \\
11: \hspace{2cm} \text{lb}' \leftarrow \min(\text{lb}', p^j_l.\text{lb} + p^j_r.\text{lb}) \\
12: \hspace{2cm} \text{ub}' \leftarrow \min(\text{ub}', p^j_l.\text{ub} + p^j_r.\text{ub}) \\
13: \hspace{1cm} \triangleright \text{Update bounds for p and its parents} \triangleleft \\
14: \hspace{2cm} \textbf{if} \ p.\text{lb} \neq \text{lb}' \ \text{or} \ p.\text{ub} \neq \text{ub}' \ \textbf{then} \ p.\text{ub} = \min(p.\text{ub}, \text{ub}'), \ p.\text{lb} = \min(p.\text{lb}, \max(p.\text{lb}, \text{lb}')) \hspace{1cm} \triangleright \text{if bounds change, update them} \\
15: \hspace{2cm} \triangleright \text{tell my parents I got updated, parents go to the front of the queue} \triangleleft \\
16: \hspace{2cm} \textbf{for} \ p_\pi \in G.\text{parent} \ \textbf{do} \ Q.\text{push}(p_\pi.\text{id}, \text{priority}=1) \\
17: \hspace{1cm} \triangleright \text{check again if we’re done:} \triangleleft \\
18: \hspace{1cm} \textbf{if} \ p.\text{lb}=p.\text{ub} \ \textbf{then} \ \text{then problem solved, continue} \hspace{1cm} \triangleright \text{back to while loop} \\
19: \hspace{1cm} \textbf{for} \ j=1 \ \text{to} \ # \ \text{features} \ \textbf{do} \hspace{1cm} \triangleright \text{enqueue viable children} \\
20: \hspace{2cm} s_l, s_r \leftarrow \text{split s on feature j} \hspace{1cm} \triangleright \text{left and right subproblems} \\
21: \hspace{2cm} p^j_l \leftarrow \text{FindOrCreateNode}(G,s_l) \hspace{1cm} \triangleright \text{get them from G} \\
22: \hspace{2cm} p^j_r \leftarrow \text{FindOrCreateNode}(G,s_r) \hspace{1cm} \triangleright \text{now calculate bounds} \triangleleft \\
23: \hspace{2cm} \text{lb}_{\text{sum}} \leftarrow p^j_l.\text{lb} + p^j_r.\text{lb} \\
24: \hspace{2cm} \text{ub}_{\text{sum}} \leftarrow p^j_u.\text{ub} + p^j_r.\text{ub} \\
25: \hspace{2cm} \textbf{if} \ \text{lb}_{\text{sum}} < \text{ub}_{\text{sum}} \ \text{and} \ \text{lb}_{\text{sum}} \leq p.\text{ub} \ \textbf{then} \hspace{1cm} \triangleright \text{child problem is viable} \\
26: \hspace{2cm} \text{Q.push}(s_l,\text{priority})=0 \hspace{1cm} \triangleright \text{back of the queue for children} \\
27: \hspace{2cm} \text{Q.push}(s_r,\text{priority})=0 \\
28: \hspace{1cm} \textbf{return} G \hspace{1cm} \triangleright \text{return G, from which we can easily extract the optimal tree} \\

When the while loop terminates, it’s because we have solved the master problem. Now we can extract the optimal tree from G by seeing which split leads to the
upper (and lower) bounds that we recorded for the optimal solution (I gave an example earlier where there were two optimal solutions).

Walkthrough

On a set of slides by Chudi Zhong, there is a walkthrough of GOSDT for the very simple n=6 dataset.

Speeding Up Computation

There are several important techniques that GOSDT uses to reduce computation time. If you don’t use these techniques, GOSDT could run slowly. Remember, the problem of finding an optimal sparse decision tree is known to be NP-hard.

Using regularization: You might have noticed that the FindOrCreateNode(G, s) subroutine uses a 2\(\lambda\) term, where \(\lambda\) is the regularization parameter. Specifically, GOSDT doesn’t even create a node if the lower bound is within 2\(\lambda\) of the upper bound. In other words, if \(\lambda\) is large, we’ll create much fewer nodes. This is because the large regularization provably excludes trees with small leaves from being the optimal solution, so GOSDT won’t create small leaves at all. One must thus never reduce \(\lambda\) to a small number, because it will lead to overfitted trees with small leaves and slow computation time. (Of course, if you make \(\lambda\) too large, you’ll just get a leaf as your optimal tree.)

Using a reference model to speed up computation: Another very useful way to reduct computation is to use a reference model to approximate the lower bounds in the algorithm. Here, instead of computing the lower bounds as we did in the Main While Loop, we use a black box algorithm like AdaBoost or random forest to produce the bounds. For instance, for a subproblem, let’s say the lower bound from our current computation is 3/n and AdaBoost’s accuracy on that subproblem is 20/n; that is, the black box missed 20 points rather than 3. In that case, we say that it’s unlikely that our single tree will be better than the black box, so it’s fairly safe to assume that the best single tree will miss at least 20 of these points. So now, our lower bound has gone from 3 to 20. Then, as long as the upper bound can get down to 20, we can stop trying to make that part of the tree more accurate and thus converge faster. In this case, we lose the guarantee of optimality, but we have performance at least as good at that of the
black box.

**Hard constraints for easing computation:** Let us discuss how to handle the hard constraint that the depth \( \leq D \); this is slightly complicated because a sub-problem that is encountered at one depth is no longer the same as a subproblem encountered close to \( D \). However, using a well-chosen value for \( D \) can be very helpful: if \( D \) is as large as, or larger than, the depth of the optimal solution, we do not need to search deeper depths. If we choose \( D \) too small, unfortunately we will miss the optimal solution and need to make \( D \) larger and try again. In practice, we rarely require depth more than 5, which we have found to be a generally good choice.

**Scaling and binary preprocessing:** I described two options earlier for binary preprocessing of continuous variables. If we use the first option (all possible splits), the problem will be computationally expensive, but GOSDT will provide the optimal solution. The second option (using only the splits from random forest or boosted decision trees) will be less expensive, and generally achieves excellent performance. GOSDT scales more slowly with the number of features than the number of observations, so using only the splits from random forest or boosted decision trees really helps.

**Some perspective**

As you can tell, modern decision tree optimization is almost nothing like greedy tree induction of early algorithms like CART or C4.5. This set of notes aims to give you the perspective of how the field has changed. I wanted to leave you with a few last notes on modern decision trees.

**Other loss functions:** GOSDT does generalize to any loss function that increases in the number of false positives and increases in the number of false negatives. So it is possible to weight the false negatives higher than the false positives if desired.

**Performance:** GOSDT tends to produce solutions that are better than C4.5 and CART, and is more reliable. While sometimes CART’s solutions are as good as GOSDT’s, we won’t know in advance when this will happen. GOSDT
is particularly useful when the user wants to optimize a custom loss function, since CART’s splitting criteria is not tuned for that loss function. Of course, GOSDT doesn’t have splitting criteria, instead it will use dynamic programming as described above.

**Theorems to reduce the search space:** Note that GOSDT has a number of additional theorems that reduce the size of the search space (that I did not discuss because of length).

**GOSDT Extension:** A 2022 extension to GOSDT is called TreeFARMS (Trees Fast RashoMon Sets) [Xin et al., 2022]. TreeFARMS uses GOSDT’s dependency graph and implementation to store *all* the trees that are optimal or close to optimal, and provides *all* of them to the user, rather than just providing a single tree. The reason for doing this is that if the user doesn’t like the tree that an algorithm like CART, C4.5, or GOSDT produces, there’s usually not much one can do about it. So, TreeFARMS allows the users to look through all the trees that are about equally good. There is a user interface called TimberTrek [Wang et al., 2022] that allows users to explore the set of all close-to-optimal sparse decision trees.

**References**


Zijie J. Wang, Chudi Zhong, Rui Xin, Takuya Takagi, Zhi Chen, Duen Horng Chau, Cynthia Rudin, and Margo Seltzer. TimberTrek: Exploring and cu-
rating trustworthy decision trees with interactive visualization. In 2022 IEEE Visualization Conference (VIS), 2022.