Parallelization of Local Search on Optimal Trees

Shaun Fendi Gan (922486604),
Arkira Tanglertsumpun (915384323)

***Abstract*—The local-search heuristic was introduced as an attempt to remedy the limitations on the scalability of Bertsimas and Dunn's Optimal Classification Tree mixed-integer optimization formulation. This paper examines the effectiveness of this heuristic as well as introducing four parallelized versions that could improve model efficiency. The Half-Split method was found to be the most scalable and time efficient, reducing run time by 30%. Standard Multi-threading on random restarts did not have a significant improvement (<5%), whereas class assignments and Deep Subtree Search were found to only benefit on datasets with many features (m $\geq$ 15).**

I. Introduction

Upon the introduction of classification and regression trees (CART) in 1984 by Leo Breiman, CART has quickly become one of the most popular methods in modern day machine learning as a result of its interpretability.

One main drawback of CART however, is that it does not achieve the same state-of-the-art level of accuracy as do ensemble methods such as random forest and gradient boosting which tradeoff interpretability for predictive power [1].

This is because CART utilizes a top-down, multi-step tree-growing heuristic approach. It recursively bi-partitions observations in isolation of future split performance, yielding one-step optimal rather than globally optimal trees. Thus, CART tends to overgrow the tree before pruning off branches in an attempt to control how large the tree becomes. Following such an approach can lead to pruning that stops before finding the best tree with the lowest error. This can be described as weaker splits hiding stronger splits. As a result, CART can obtain final trees which do not capture the underlying relationship in the data and perform suboptimally when classifying future points. A clear progression is to optimize the tree-growing process to globally minimise for the error.

Optimal decision trees were designed to remedy the aforementioned shortcomings of CART in order to achieve better predictive performance while retaining the interpretability that is characteristic of CART trees. Optimal decision trees leverage optimization to build an entire tree in one single step - forming splits with full knowledge of all other splits to produce globally optimal trees. Empirically, the optimal tree approach has been shown to significantly outperform CART at small depths but is not able to prove optimality in practical time for trees of larger depths. This is due to the sheer number of variables and constraints that grow with the depth of the tree from the mixed-integer optimization formulation [2]. According to Bertsimas and Dunn, under a set of realistic assumptions, the complexity cost of OCT is $O(npKlog^2n)$ and the cost of CART is $O(npKlogn)$.

Advisor: Michael Lingzhi Li

To resolve this issue of scaling, an *optimization via local search* approach was introduced to reduce the search space and speed up computational time. Currently, the local search approach presents many opportunities for parallelization that may enable scaling to larger problems and increasing the number of random restarts within practicable time to improve the quality of solutions.

II. Theory

This section describes the Optimal Classification Tree formulation and its scalability limitations, motivating the development of the local-search heuristic.

A. Optimal Classification Tree

The Optimal Classification Tree is best described using a mixed integer optimization formulation which enforces a tree structure that jointly minimizes the training misclassification loss and tree complexity:

$$\min \frac{1}{\hat{L}} \sum_{t \in \mathcal{T}_L} L_t + \alpha \cdot C \tag{1}$$

where $\hat{L}$ is the baseline misclassification, L_t is the misclassification loss of each tree, t is a node, $\mathcal{T}_L$ is the set of points that are leaves in the tree, α is the complexity parameter that is tuned for, and C is the complexity of the tree, that is the number of splits the tree makes.

All aspects of the classification tree are modeled as decision variables within the formulation, constrained with certain conditions to ensure the output follows a hierarchical tree structure. An example of a tree can be seen in the Appendix (Figure 5).

A key constraint in the formulation is optimising the split in the branches. This is modeled as

$$\boldsymbol{a}_m^\top \boldsymbol{x}_i \geq b_m - (1 - z_{it}) \tag{2}$$

$$\boldsymbol{a}_m^\top (\boldsymbol{x}_i + \boldsymbol{\epsilon} - \epsilon_{\text{min}}) + \epsilon_{\text{min}} \leq b_m + (1 + \epsilon_{\text{max}})(1 - z_{it}) \tag{3}$$

where a represents the split feature in x, b is the split value, z_{it} is a key decision variable which keeps track of observation to leaf assignments, and ϵ is a constant introduced specifically to ensure numerical stability (integer optimisation problems often suffer from instability in greater than or less than constraints). Additional constraints in the formulation enforce assignments of observations to leaves, and ensure that branches, leaves and nodes enforce the correct tree structure. It should also be noted that this formulation requires normalization of features to be between zero and one. The full mixed integer optimization formulation is shown in Appendix Fig. 6.

Constructing the tree in a single-step optimization model ensures that all modeling decisions at every step of the way are made considering the full impact such decisions have on the entire tree (in contrast to a local impact assessment as in CART). This single step process also removes the need for pruning and impurity found in CART that often causes weaker splits to hide stronger splits due to weak threshold. An example is seen in Fig. 1

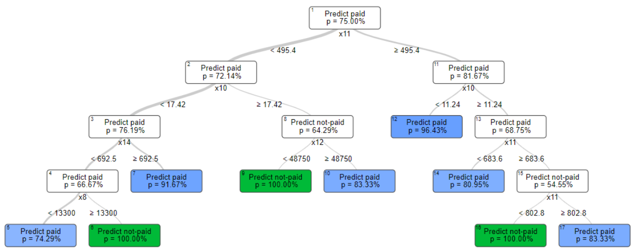
1 Predict paid p = 75.00%
x11
< 495.4
≥ 495.4
2 Predict paid p = 72.14%
x10
11 Predict paid p = 81.67%
x10
< 17.42
≥ 17.42
< 11.24
≥ 11.24
3 Predict paid p = 76.19%
x14
8 Predict not-paid p = 64.29%
x12
12 Predict paid p = 96.43%
13 Predict paid p = 68.75%
x11
< 692.5
≥ 692.5
< 48750
≥ 48750
< 683.6
≥ 683.6
4 Predict paid p = 66.67%
x8
7 Predict paid p = 91.67%
9 Predict not-paid p = 100.00%
10 Predict paid p = 83.33%
14 Predict paid p = 80.95%
15 Predict not-paid p = 54.55%
x11
< 13300
≥ 13300
< 802.8
≥ 802.8
5 Predict paid p = 74.29%
6 Predict not-paid p = 100.00%
16 Predict not-paid p = 100.00%
17 Predict paid p = 83.33%

Fig. 1. Example of an Optimal Classification Tree, ran using Interpretable AI software. This tree was run on loan data from Lending Club, to predict default rate.

While this Mixed Integer Optimisation(MIO) formulation can be solved using commercial optimization solvers for smaller datasets, the program becomes too costly to solve for large scale data. The difficulty arises from the large number of binary variables that are required to keep track of point to leaf allocations which rapidly grow by $n * 2^D$ where n is the number of observations and D is the maximum depth of the tree. The number of constraints also grow exponentially with increasing tree depth. Solvers therefore cannot scale to large datasets within an acceptable time period and solutions obtained after a long period of time can still be far from optimality.

B. *Local Search*

To remedy the shortcomings of the OCT MIO formulation, the local-search heuristic algorithm has been developed to reduce the problem size to hopefully enable scaling to large scale data. Local search iteratively refines the nodes within multiple randomly restarted and independent trees with the goal that the best of these trees will be the global optimum to the Optimal Classification Tree or very close to it. It consists of two components: 1. Random Restarts and 2. Split optimization:

Random Restarts warm-start the algorithm with multiple initial and independent trees configured from random forest to initiate a medley of random trees. Obtaining initial trees with Random Forests works well since the trees generated are good quality individually, reducing the number of local search iterations required. Since Random Forests trains trees on a subset of features and bootstraps samples, this ensures that the restarts have unique trees at each warm start. These trees are then iteratively refined in the Split Optimization stage into locally optimal trees with the hopes that one of these will be close to the global optimum.

Split optimization obtains locally optimal splits for all nodes in a tree. It starts by walking to a randomly located split within a tree, and performs a local optimisation on the subtree with the corresponding node as its root. There are three local optimization choices to re-optimize each node: replace the current split with another split value, split on a different variable, or replace the current subtree with upper/right or lower/left child subtrees. This process is repeated for all nodes in the tree in random order until a locally optimal tree is achieved with error within a given tolerance. A node can be revisited several times if it can be improved upon changing neighbouring nodes.

To obtain strong out-of-sample performance, the hyperparameters of the model including the complexity parameter α and the maximum depth, which controls the number of sequential splits that can be used to classify any point, will be tuned with a grid search.

It should be noted that local search is a heuristic to approximate a computationally expensive optimisation formulation, providing near optimal solutions in tractable time. It is not guaranteed to find globally optimal solutions but tends to find many locally optimal solutions with a similar performance at much faster speeds of which the best locally optimal tree is then selected as the alternative classifier to OCT [1].

This project sought to find ways to further improve on this heuristic by utilising serial code optimization and parallel computing principles with the underlying objective of improving both efficiency and accuracy. The original algorithm is detailed fully in Algorithm 1,2,3.

Algorithm 1: LocalSearch

```
  Input: Starting decision tree $\mathbb{T}$; training data $\mathbf{X}, \mathbf{y}$
  Output: Locally optimal decision tree
1 repeat
2     error $_{\text{prev}} \leftarrow$ loss($\mathbb{T}$,$\mathbf{X}$,$\mathbf{y}$)
3     for t $\in$ shuffle(nodes($\mathbb{T}$)) do
4         $\mathcal{I} \leftarrow \{i : \mathbf{x}_i$ is assigned by $\mathbb{T}$ to a leaf
           contained in $\mathbb{T}_t\}$
5         $\mathbb{T}_t \leftarrow$ OptimizeNodeParallel($\mathbb{T}_t, \mathbf{X}_{\mathcal{I}}, \mathbf{y}_{\mathcal{I}}$)
6         replace $t$th node in $\mathbb{T}$ with $\mathbb{T}_t$
7         error$_{\text{cur}} \leftarrow$ loss($\mathbb{T}$,$\mathbf{X}$,$\mathbf{y}$)
8 until error$_{\text{prev}}$ = error$_{\text{cur}}$;
```

III. Data

To benchmark the computational performance between the OCT MIO formulation, serial local search, and performance-engineered local search approaches, two datasets have been chosen due to their differences in dimensionality and problem complexity. The datasets are described below:

- Lending Club: consists of 1.38 million rows of loan data with 16 features and loan status (paid or not paid) between 2007 through 2015. Key variables such as loan terms, payment plans, and annual income are used for predicting the outcome variable, the loan status.
- Iris: a well-known pattern recognition dataset with 150 records of information on each iris plant species as well as 4 plant characteristics such as petal length and width,

Algorithm 2: OptimizeNodeParallel

```
Input: Subtree $\mathbb{T}$ to optimize; training data $\mathbf{X}$, $\mathbf{y}$
Output: Subtree $\mathbb{T}$ with optimized parallel split at root
1  if $\mathbb{T}$ is a branch then
2      $\mathbb{T}_{\text{lower}}$, $\mathbb{T}_{\text{upper}} \leftarrow$ children($\mathbb{T}$)
3  else
4      $\mathbb{T}_{\text{lower}}$,$\mathbb{T}_{\text{upper}} \leftarrow$ new leaf nodes
5  error$_{\text{best}} \leftarrow$ loss($\mathbb{T}, \mathbf{X}, \mathbf{y}$)
6  $\mathbb{T}_{\text{para}}$, error$_{\text{para}} \leftarrow$ BestParallelSplit($\mathbb{T}_{\text{lower}}$,$\mathbb{T}_{\text{upper}}$,$\mathbf{X}$,$\mathbf{y}$)
7  if error$_{\text{para}} <$ error$_{\text{best}}$ then
8      $\mathbb{T}$, error$_{\text{best}} \leftarrow \mathbb{T}_{\text{para}}$, error$_{\text{para}}$
9  error$_{\text{lower}} \leftarrow$ loss($\mathbb{T}_{\text{lower}}, \mathbf{X}, \mathbf{y}$)
10 if error$_{\text{lower}} <$ error$_{\text{best}}$ then
11     $\mathbb{T}$, error$_{\text{best}} \leftarrow \mathbb{T}_{\text{lower}}$, error$_{\text{lower}}$
12 error$_{\text{upper}} \leftarrow$ loss($\mathbb{T}_{\text{upper}}, \mathbf{X}, \mathbf{y}$)
13 if error$_{\text{upper}} <$ error$_{\text{best}}$ then
14     $\mathbb{T}$, error$_{\text{best}} \leftarrow \mathbb{T}_{\text{upper}}$, error$_{\text{upper}}$
15 return $\mathbb{T}$
```

Algorithm 3: BestParallelSplit

```
Input: Lower and upper subtrees $\mathbb{T}_{\text{lower}}$ and $\mathbb{T}_{\text{upper}}$ to
       use as children of new split; training data $\mathbf{X}$, $\mathbf{y}$
Output: Subtree with best parallle split at root; error
        of best tree
1  $n, p \leftarrow$ size of $\mathbf{X}$
2  error$_{\text{best}} \leftarrow \infty$
3  for $j = 1, \cdots, p$ do
4      values $\leftarrow \{X_{ij} : i = 1, \cdots, n\}$
5      sort values in ascending order
6      for $i = 1, \cdots, n-1$ do
7          $b \leftarrow \frac{1}{2}$(values$_i$values$_{i+1}$)
8          $\mathbb{T} \leftarrow$ branch node $\mathbf{e}_j\mathbf{x} < b$ with children
             $\mathbb{T}_{\text{lower}}$,$\mathbb{T}_{\text{upper}}$
9          if minleafsize($\mathbb{T} \geq N_{\text{min}}$) then
10             error$\leftarrow$ loss($\mathbb{T}$,$\mathbf{X}$,$\mathbf{y}$)
11             if error $<$ error$_{\text{best}}$ then
12                 error$_{\text{best}} \leftarrow$ error
13                 c m m $\mathbb{T}_{\text{best}} \leftarrow \mathbb{T}$
14 return $\mathbb{T}$, error$_{\text{best}}$
```

and sepal length and width. The plant characteristics are used for predicting the plant species.

IV. Approach

This section details the steps taken towards performance engineering the local-search heuristic, using both serial code optimization as well as parallelization techniques.

A. Serial Optimization

A critical step in performance engineering any algorithm is the optimization of serial code. In this regard, the following serial code optimization techniques have been implemented to ensure good baseline computational performance:

- variable pre-allocations, function barriers and multiple dispatch have been used throughout the local search implementation to enable end-to-end type specialization and inference to generate optimal code for the compiler
- views, variable mutation, and stack allocating small arrays have been utilized to avoid unnecessary, costly heap allocations
- `@inbounds` macro has been used in tight inner loops, removing the bounds checking process to reduce computational time
- `@inline` has been utilized to remove certain function calls in the compilation process, instead executing the code in the line

Completing this serial optimisation significantly reduced the runtime for each Local Search iteration from an average of 1.3 seconds per run, to only 330 ms for 100 observations, and from 55 seconds to 13 seconds for 500 observations. Serial code optimization thus contributed large improvements in runtime, making the local search extremeley time efficient compared to the MIO approach.

B. Parallel Optimization

The following section examines several algorithm design choices and parallelization techniques considered to further speedup local-search algorithm computational time.

The main parallelization techique considered was loop-based multi-threading with static scheduling since many loops in the algorithm are independent of each other but take approximately the same amount of time and thus can be simultaneously computed. The main loops that were parallelized include random restarts, split re-optimization and point to leaf assignments. As long as the computational costs arising from stack construction each time a computational thread is spun up can be counter-balanced, this method will reduce the algorithm runtime.

Task-based parallelism and distributed processing were not considered in this project since trees have a hierarchical structure and tasks within the algorithm typically have to be executed sequentially and each step is heavily dependent on previous steps. Moreover, it is expected that distributed processing would incur significant overhead costs of message passing when updating different splits of each tree and when each observation needs to be allocated to a leaf node and thus would be less efficient for this particular algorithm.

C. Parallelizing Random Restarts

As each random tree is generated and refined independently of all other trees, the trivial improvement is to parallelise the random restarts by running local search on multiple starting trees simultaneously on separate cores. Parallelizing across starting solutions is expected to improve runtime performance linearly with the number of cores.

D. Parallelizing Half Splits

The current optimization algorithm re-optimizes a single random node at a time. The revised approach will instead attempt to re-optimize more than a single node simultaneously and in parallel to accelerate finding the optimal solution. This is challenging as decisions on each subtree split can affect its children or its neighbours. The revised approach thereby required that each node search iteration select nodes in a way such that local re-optimization can proceed independently of all other nodes. When optimizing this process, it was therefore crucial to separate the optimisation onto different threads.

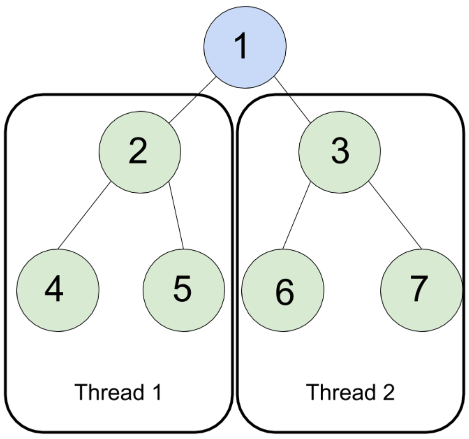

Fig. 2. Representation of half split parallel optimisation. Here thread 1 and 2 optimise separate sections of the tree with green nodes located in the black outlined box. Node 1 in blue is not optimised for in this process.

The most convenient way to do so is to optimize each half of the trees in parallel and on separate cores. This process could be easily extended in factors of 2 that would separate the optimisation into deeper subtrees. A quarter split could be used to optimise the 4 subtrees simultaneously.

It is however clear that parallelizing this split into quarters or eigths on the same tree leads to additional complexity and would only be useful for trees with high depth as so few splits can be simultaneously optimized on smaller trees. Additionally, this method creates an artificial separation and does not optimize the split joining these two substructures - in this case the first split is never selected for re-optimization. This can lead to trees that are suboptimal compared to those generated from non-parallelized local search. Parallelization therefore has significant implications on both the accuracy and speed of the refined local search approach.

An alternative approach of using a Sparse Matrix Factorization method was considered to perform calculations that allowed for communication between results. However, the method was deemed unnecessary for this proof of concept.

E. Deep Subtree Search

Currently, local search uses a breadth-first search approach to quickly search across all nodes. It considers three re-optimization possibilities at each split: to replace the current split value, to split with a different variable, or to replace the current subtree with subtrees of the children node.

Instead of a breadth-first search however, a depth-first search approach could be used to optimize the current node. All potential subtree replacements which includes every possible

Algorithm 4: Deep-OptimizeNodeParallel

```
Input: Subtree $\mathbb{T}$ to optimize; training data $\mathbf{X}$, $\mathbf{y}$
Output: Subtree $\mathbb{T}$ with optimized parallel split at root
1  if $\mathbb{T}$ is a branch then
2      $\mathbb{T}_t \leftarrow$ store all subtrees of all children($\mathbb{T}$)
3  else
4      $\mathbb{T}_{\text{lower}}, \mathbb{T}_{\text{upper}} \leftarrow$ new leaf nodes
5  $\text{error}_{\text{best}} \leftarrow \text{loss}(\mathbb{T}, \mathbf{X}, \mathbf{y})$  $\mathbb{T}_{\text{para}}$, $\text{error}_{\text{para}} \leftarrow$
   BestParallelSplit($\mathbb{T}_{\text{lower}}, \mathbb{T}_{\text{upper}}, \mathbf{X}, \mathbf{y}$)
6  if $\text{error}_{\text{para}} < \text{error}_{\text{best}}$ then
7      $\mathbb{T}$, $\text{error}_{\text{best}} \leftarrow \mathbb{T}_{\text{para}}, \text{error}_{\text{para}}$
8  $\text{error}_{\text{lower}} \leftarrow \text{loss}(\mathbb{T}_{\text{lower}}, \mathbf{X}, \mathbf{y})$
9  for all subtrees do
10     Begin Multi Threading
11     $\text{error}_{\text{sub}} \leftarrow \text{loss}(\mathbb{T}_t)$
12 if $\min(\text{error}_{\text{sub}}) < \text{error}_{\text{best}}$ then
13     $\mathbb{T} \leftarrow \mathbb{T}_t[\arg\min(\text{error}_{\text{sub}})]$
14 return $\mathbb{T}$
```

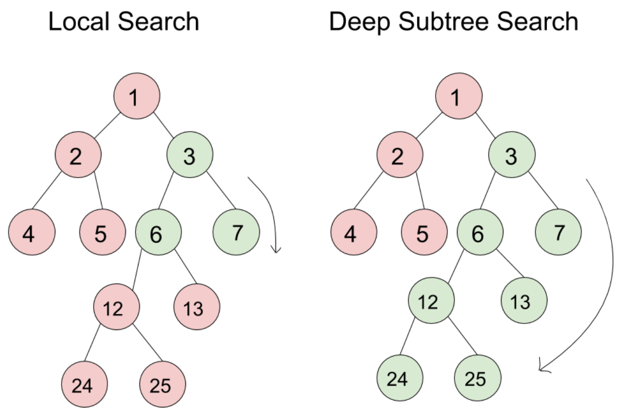

Fig. 3. Above shows the difference between local search and deep subtree search. Green nodes indicate nodes that are searched and red nodes are not searched. Local search truncates at children of the subtree node, deep subtree search truncates at the terminal leaf node, where all combinations are tested.

subtree and child are verified for the lowest error, where the subtree replacement with the lowest error is kept. In Fig 3, the difference between both searches are shown and Algorithm 4 described the modification made in order to carry out this method.

In this replacement, we are replacing a current node with its children where all its subsequent children are relabeled into the correct indices at their corresponding depth. For example, if node 3 is replaced with node 6, node 12 would change to 6, 13 to 7, 24 to 12 and 25 to 13. A formula was found to calculate this process for relabelling nodes as

$$\Gamma = \text{node} + (\text{parent root} - \text{subtree root}) \cdot 2^d \tag{4}$$

where Γ is the nodes new destination index, node is the node's index on the subtree, the parent root refers to the root of the tree it is replacing, subtree root is the root of the subtree, and

d is the depth of the subtree. The depth of the subtree can be calculated using

$$d = \left\lfloor \log_2 \frac{\text{node}}{\text{subtree root}} \right\rfloor \tag{5}$$

providing an efficient formula to replace subtrees onto their new expected location. As an example calculation, if a subtree at node 7 with children as 14,15 would replace the parent node 3, the destination of the child 14 would be

$$\Gamma = 14 + (3 - 7) \cdot 2^1 = 6 \tag{6}$$

where 6 is its new index.

Using Equation 4 and 5, the potential subtree replacement can be found in an efficient manner, and its potential can be evaluated to determine if it should replace the current node.

The process of checking all potential subtrees was then multi-threaded to reduce strain on computational time.

F. Observation assignment

In the serial implementation of local search, runtime is substantially bottlenecked by the process of assigning each observation as a result of the tree structure of the classifier.

In order to assign a point to the the correct corresponding leaf, the values of each observation is evaluated across all the splits in the tree. This continues until all observations reach a leaf node and is a process that is frequently called upon when evaluating the loss of a new potential splits. For larger datasets with more observations, this can quickly make local search impractical to run.

Since each observation is assigned to a leaf independently of other observations, this process can be parallelized in order to improve run time. This was the fourth and final parallel method explored in the project for improving local search.

G. Multithreading for parameter selection

Multithreading was utilized to gridtune the model parameters such as the maximum depth of the tree (D) and the complexity parameter (α).

In addition to serial code optimization and developing a parallelised implementation of this algorithm, code profiling and efficient memory management was applied.

V. Results and Discussion

The following section details the results of performance engineering the local search algorithm with the methods described in the previous section.

A. Code profiling

To examine code block performance profiles within the serial code implementation, Juno.jl was leveraged and results are shown in Appendix Fig. 7

From the code profile, it can be seen that there is not a clear function dominating processing time, indicating that each function appears to perform well. The largest section in the center refers to `getindex`, which was difficult to optimize further without changing the structure of the code. These calls are bottlenecked by the recursive nature of the tree structure that assigns nodes to leaves based on the splitting constraints. On prior versions of the code many substantial function calls were removed from the profile once multiple dispatch, inlining and inbounding were implemented.

Throughout this project, dictionary structures were used to store values observation assignments, as well as corresponding split or (**a**,**b**) values to their value. Although dictionaries are typically a slow structure to work with, this was necessary in maintaining a clear relationship between points are their assigned leaf/ node. Moreover, since splits are continually being refined and re-optimized in the algorithm, the use of dictionaries is necessary to quickly identify the right node and its dependents or to update the split or remove the node.

It should be noted that Local Search is an algorithm which necessitates the storing of multiple large structures, containing many variables that scale with the number of observations and features in the dataset. Therefore, although allocations were reduced to a minimal, it was still allocated at megabytes orders of magnitude. Due to this, `BenchmarkTools` was avoided due to frequent kernel crashes.

B. Parallel Local Search Methods

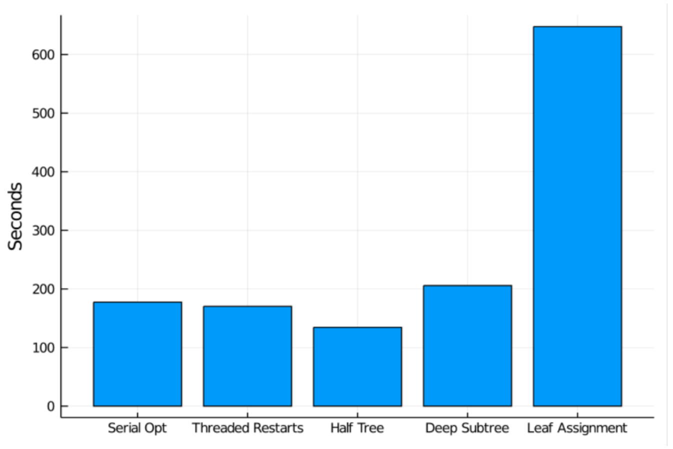

Fig. 4. Time taken to complete a local search procedure. This was done for MIO was excluded α =0.0001, Max Depth = 4, 200 observations in training data, and 50 random restarts. The OCT MIO formulation was excluded from the plot as the run did not terminate in over an hour.

Firstly, the results seen in Fig.4 indicate that in general, local search achieved significantly faster runtimes than the OCT MIO formulation in both the Iris and lending models. It was found that MIO was only tractable to about 100 observations, after which solver took an exponentially long time to run and thus was terminated after 60 minutes. On the other hand, local search was able to scale up to 1000 observations from the same dataset without significantly impeding on runtime.

The substantial benefit from improved runtime further allowed for larger training datasets to be utilized in local search, significantly improving the heuristic's accuracy. For instance, given its long runtime, the OCT MIO, at its best, achieved a 69% out-of-sample accuracy on the smaller training set in which local search obtained 72%. However, with 1000 observations, local search was able to attain an out-of-sample R^2 of 83%.

TABLE I
HALF SPLITS, $\alpha = 0.0001$, $D = 4$

Number of Observations	Time	Out of Sample R^2
100	16.75	0.721
200	134.53	0.814
500	164.62	0.828
1000	416.96	0.800

From Table I, it is also evident that re-optimizing half-trees on different threads perform the best out of all methods considered and scale up in observations in practical time. Deeper analyses into further splitting the tree should therefore be conducted as a follow-up study as additional benefits in runtime appear promising.

It is surprising to note that parallelizing each tree on separate threads and threading the assignments of observations to leaves did not improve computational time substantially from the optimized serial code. This is likely due to non-negligible costs associated with spinning-up additional threads that may not outweigh the reduction in performing parallel computations. However, it should be noted that both the Iris and Loan data are relatively simpler datasets. As such, these observations may not generalize to more complex datasets which may have more complicated tree structures and may benefit from threading frequent loops with relatively heavier computations.

Preliminary analysis into the deep subtree searches, on sample sizes of 100, 200, 500 for depths of 2 to 5, showed no obvious signs of improvement on the out of sample R^2 performance. Despite this however, this method yielded comparable run times to other methods, indicating that parallelized searching was aiding in its efficiency. This method should be further tested on datasets with inherently deeper tree structures as both datasets used in this project had optimal tree depths of around 2 to 4.

TABLE II
TIMING FEATURES COMPARSION, $\alpha = 0.0001$

Model	t [Lending-club (m=15)]	t [Iris (m=4)]
OCT MIO	563.5	54.34
Serial Opt.	31.24	5.62
Threaded Restarts	25	5.06
Threaded Half-Tree	16.75	3.05
Deep Subtree Search	38.19	9.41
Leaf Assignment	134.15	169.15

The stark differences between the training time of Iris and Loan, both of which were trained on 100 observations, illustrate the impact of dimensionality and problem complexity on local search runtime. Although the Loan dataset had quadruple the number of features contained in the Iris dataset (16 vs. 4 features), the time required to train the Loan tree more than quadrupled that of the Iris tree. Interestingly, the parallelized assign class method (or z method) improved in time on datasets with larger features, performing approximately 20% better. This could indicate that the parallelized observation assignment would see the most benefit for datasets with many features(m $\geq$ 15)

VI. CONCLUSION

The results from this study demonstrate the effectiveness of utilizing serial code optimization and parallel computing principles to performance engineer the local-search heuristic, laying a foundation to remedy the limitations of the Optimal Classification Tree mixed-integer optimization model. Deeper analysis however, is necessary to prioritize between predictive power and speed, and to uncover how to most efficiently incorporate parallelization in algorithm design across different datasets with varying dimensionality levels and relationship structures. The following section outlines the key next steps for future work.

Data Dimensionality

As previously mentioned, in a high dimensional setting, there are significantly more candidate parallel split values to consider. This has direct implications on the runtime of Algorithm 3: BestParalleSplit, which conducts a search of the best parallel split values across all features. Algorithm 3 may therefore be worthy of study in future works.

Tree Depth and Complexity

In most classification problems, tree depth is extremely crucial to correctly capture nonlinear and complex relationships between the output and the covariates. As mentioned earlier, a large tree depth has significant implications on the number of binary variables and constraints in the Optimal Classification Tree MIO formulation. Similarly, depth also substantially affects the number of potential nodes the local search heuristic has to iterate through to establish a locally optimum tree. An investigation into a depth-first search approach against breadth-first search in problems of varying tree depths and dimensionality could bear interesting results

Random Restarts

The predictive performance of local search is highly dependent on the number of random restarts used in the algorithm to find the best tree, and the accuracy of the method is expected to improve with additional restarts but at a much slower rate. Deeper analysis is therefore required to determine the appropriate number of restarts to balance the tradeoff between speed and out-of-sample accuracy across different applications.

Other modes of parallelism

Further work is needed to examine the applicability and effectiveness of other parallel computing methods. For instance, a dynamically-scheduled multi-threading approach may be more effective in parallelizing the observation to leaf assignments and individual tree restarts as there may be large variations between different iterations. Dynamic scheduling can thus be used to better utilize threads as they become free.

REFERENCES

[1] Leo Breiman, Jerome Friedman, Charles J. Stone R.A Olshen *Classification and regression trees*, 1984.
[2] Dimitris Bertsimas, Jack Dunn. *Machine Learning under a Modern Optimization Lens*. Dynamic Ideas LLC, Massachusetts, 2019.

APPENDIX

As a final note, both the authors would like to thank the advisor to this paper, Michael Li for the support and guidance throughout this project.

Please find the figures referenced in the report below.

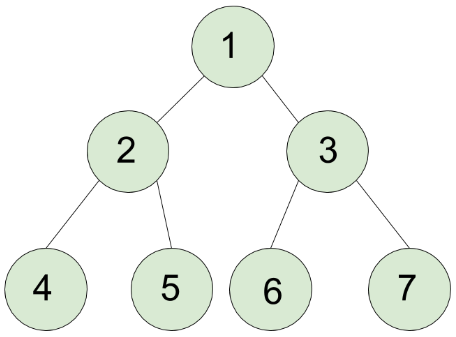

Fig. 5. This tree shows the number labelling scheme used in this project, where nodes are labeled left to right, starting from top to bottom.

$$
\begin{aligned}
\min \quad & \frac{1}{\hat{L}} \sum_{t \in \mathcal{T}_L} L_t + \alpha \cdot C \\
\text{s.t.} \quad & L_t \geq N_t - N_{kt} - n(1 - c_{kt}), && \forall t \in \mathcal{T}_L,\ k \in [K], \\
& L_t \leq N_t - N_{kt} + n c_{kt}, && \forall t \in \mathcal{T}_L,\ k \in [K], \\
& L_t \geq 0, && \forall t \in \mathcal{T}_L, \\
& N_{kt} = \sum_{i:\ y_i = k} z_{it}, && \forall t \in \mathcal{T}_L,\ k \in [K], \\
& N_t = \sum_{i=1}^{n} z_{it}, && \forall t \in \mathcal{T}_L, \\
& \sum_{k=1}^{K} c_{kt} = l_t, && \forall t \in \mathcal{T}_L, \\
& C = \sum_{t \in \mathcal{T}_B} d_t, \\
& \mathbf{a}_m^{\mathsf{T}} \mathbf{x}_i \geq b_m - (1 - z_{it}), && \forall i \in [n],\ t \in \mathcal{T}_L,\ m \in \mathcal{R}(t) \\
& \mathbf{a}_m^{\mathsf{T}} (\mathbf{x}_i + \boldsymbol{\epsilon}) \leq b_m + (1 + \epsilon_{\max})(1 - z_{it}), && \forall i \in [n],\ t \in \mathcal{T}_L,\ m \in \mathcal{L}(t), \\
& \sum_{t \in \mathcal{T}_L} z_{it} = 1, && \forall i \in [n], \\
& z_{it} \leq l_t, && \forall t \in \mathcal{T}_L, \\
& \sum_{i=1}^{n} z_{it} \geq N_{\min} l_t, && \forall t \in \mathcal{T}_L,
\end{aligned}
$$

$$
\begin{aligned}
& \sum_{j=1}^{p} a_{jt} = d_t, && \forall t \in \mathcal{T}_B, \\
& 0 \leq b_t \leq d_t, && \forall t \in \mathcal{T}_B, \\
& d_t \leq d_{p(t)}, && \forall t \in \mathcal{T}_B \setminus \{1\}, \\
& z_{it}, l_t, c_{kt} \in \{0, 1\}, && \forall i \in [n],\ k \in [K],\ t \in \mathcal{T}_L, \\
& a_{jt}, d_t \in \{0, 1\}, && \forall j \in [p],\ t \in \mathcal{T}_B.
\end{aligned}
$$

Fig. 6. Full mixed integer optimiztion formulation of Optimal Classification Trees

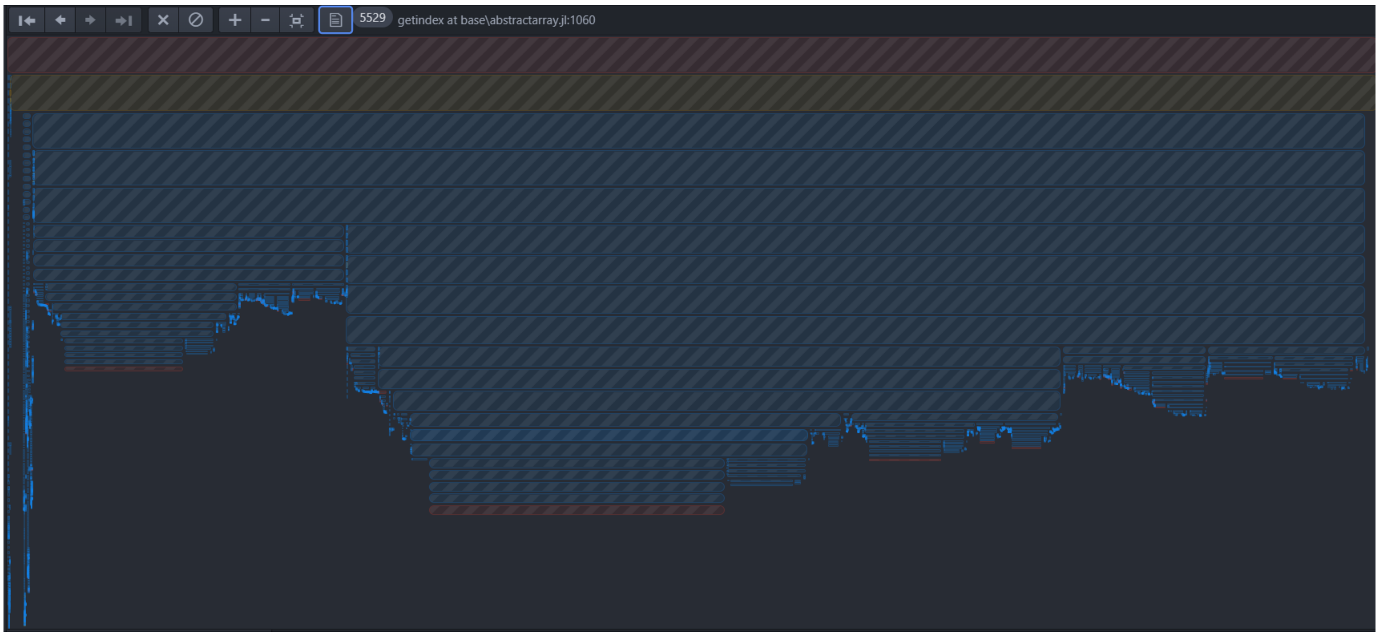

Fig. 7. Profile of code optimised serial code LocalSearch. It can be seen that there is no one clear function domintating processing time, indicating that the function is optimised. The largest section in the center refers to `getindex`, which is difficult to further optimize.