# Programming Assignment 5: Handwriting Recognition Systems Tree vs. Table Search

PA5: Handwriting Recognition Systems - Tree vs Table Search is an assignment that tasks students to incrementally implement a new classification (either tree or table) handwriting recognition system. This assignment is more or less a large extension of PA4, while prompting students to use newer concepts discussed in class. There are several little nuances that students must pay close attention to when programming in C++, but one topic that separates PA5 from the others is the use of generic programming. PA5 further reinforces students' skills in these defining topics that have been recently covered in the lecture, mainly Topic 12 - Object Oriented Programming, Topic 13 - Generic Programming, Topic 14 - Functional Programming, and Topic 16 - Trees. Students are given the option to decide which type of algorithm they wish to implement between trees or tables. We chose the tree route and our discussion will be more specific to that. This assignment required students to write a total of six classes: Vector, Image, Tree, HRSLinearSearch, HRSBinarySearch, and HRSTreeSearch. Additionally, students were required to implement a sorting helper function called sort which is used in our classes and is derived from generic programming. This is fairly similar to PA4, yet our new classes can work for several types, as opposed to strictly ints. The end goal is to develop two handwriting recognition functions (linear and binary versions) and the Tree handwriting search system, all of which are designed to classify images of handwritten digits. In addition to the class implementations, it is also crucial to pass the extensive test suite consisting of both directed and random tests. Fortunately, students were given an appropriate amount of tests already to ensure that their implementations were working as expected. Finally, as always, when wanting to analyze your implementations and compare/contrast them with each other, programmers must evaluate their classes to determine the execution time. This is an important step in the optimization phase in ensuring that our performance is as powerful as possible.

## 1 Implementation

The overall implementation for PA5 borrows many ideas from PA4. The one main exception is the use of generic programming. This requires us to use a template to specify our own data type that may be applicable within each function. For PA4 our data worked only with ints, hence the difference in class name (ie: VectorInt vs Vector). PA5 draws on these previously defined implementations but allows users to have the option to work with different data types. Our group chose to implement the HRSTreeSearch class. Our initial intuition to this choice was partially due to the description of the specification in the handout. We perceived Tree search to be a better performing system than our already impressive binary search. With this in mind, we were also intrigued by how each node stores a lot of information such as: a value, a pointer to left subtree, and a pointer to the right subtree. To us this seemed very efficient and convinced us enough to implement this system. In regards to our implementation, first and foremost, we defined our private member field m training set as a functor. This allowed us to use the Image class comparison function which is necessary

for all of our public member functions. We then moved onto the default constructor which simply initialized our m training set with our k value and functor type previously defined. This intuitively compares the intensity of two images due to our Image class inheritance. Our next function was to train the system. Here we must use a loop to iterate through all of the training images by using a Vector instance "vec" depending on its size. Then we simply use the dot access operator to add each training image to our m training set based on vec's index. Our last function to implement in this class is the classify function and also is arguably the most important. Classify required us to first implement a helper function to determine the Euclidean distance between two images. Now we can successfully find the smallest Euclidean distance. By making use of the find closest function in our Image class we can set the respective parameters to the Image instance "img" as well as the result of our helper function. Once we return this our code will execute in a way that will provide us with the image with the smallest Euclidean distance from the given image "img". Tree is generic meaning that it allows us to store elements of any data type, as long as necessary functions (comparison and distance) are provided. This is extremely useful in many real-life contexts. One potential drawback is that if our tree is in a unique distribution like a nearly sorted order or even fully sorted to begin with, there can be an imbalance with the tree partitioning that may increase the time complexity. This is rare though as users would likely use this system with their handwritten data in an already sorted formation.

# 2 Complexity Analysis

The following complexity analysis is done by counting critical loop iterations. Here $N$ represents the number of images in the training set, $M$ represents the number of images being classified, and $K$ is the input k for the tree;

1. Training

   (a) The training function loops through the $N$ elements of a given vector, executing Tree::add on each. Add performs a binary search on the tree until an empty position is found that maintains the binary search tree, and adds the given value to that location. We can assume that the binary tree is roughly balanced, so on average it takes $log_2(N)$ iterations to reach an empty position. Add is executed $N$ times, so $T_K(N) = N * log_2(N)$. In big-O, this is $O(NlogN)$.

2. Classify 1 Image

   (a) To classify a single image, we begin by doing a non-exhaustive search using the comparisons of the intensity to decide when to take the left/right node. We then perform an exhaustive search on a subtree with size $K$, where $K$ is given. This means that the non-exhaustive search will check $log_2(N) - log_2(K)$ values, adding as many loop iterations. The exhaustive search checks each entry in the subtree, so it adds $K$ iterations. This results in a complexity of $T_K(N) = log_2(N) - log_2(K) + K$ which is $O(log(N))$.

3. Training + Classify M Images

   (a) To classify M images we just multiply the time complexity for a single image times M, and we add the time to train N images. The resulting complexity is $N * log_2(N) + M(log_2(N) - log_2(K))$. Big-O with respect to $N$ is $O(Nlog(N))$, and with respect to $M$ is $O(M)$.

Table 1: Table for time complexity analysis.

| | | `HRSTreeSearch` |
|---|---|---|
| Training | $T_K(N)$ | $Nlog_2N$ |
| Training | big-O wrt N | $O(NlogN)$ |
| Classify 1 image | $T_K(N)$ | $log_2N - log_2K + K$ |
| Classify 1 image | big-O wrt N | $O(logN)$ |
| Training+classify M images | $T_K(N, M)$ | $Nlog_2N + M(log_2N - log_2K)$ |
| Training+classify M images | big-O wrt N | $O(NlogN)$ |
| Training+classify M images | big-O wrt M | $O(M)$ |

# 3 Optimization

Once we finalized the correctness of our code, it was time to evaluate it. Fortunately, both linear and binary search were a near copy from PA4 minus the generic aspect of it. Thus, our evaluations were very consistent with that of PA4. We both found that the tree version of our unoptimized code was a bit longer to fully execute than we expected. In fact, there was one instance where using the full training and classification dataset resulted in an execution time of 917 sec, roughly 15 min! We then turned to our flame graphs to observe what potential areas we could focus our attention to. We noticed that our find closest function was causing the greatest delays in potential performance, and that training that data set was also surprisingly slow. After extended testing, we discovered that we were using the Euclidean distance function everywhere. This meant that any functions that traversed the tree, like add, contains, and find closest, were performing a lengthy calculation for every node they visited. These slowdowns were mitigated by only calculating the Euclidean distance during the exhaustive search to find the closest and using intensity comparisons elsewhere. This dramatically improved our execution times to a place we were happy with.

# 4 Quantitative Evaluation

1. Overall our binary data is very impressive. As you can see in both the graph and table, for the same amount of training and classification images as linear search, binary search has a much faster execution time. However, the one down side about binary search is its accuracy for smaller values of k (size of the final linear search when using the binary search). Unlike linear search which has the same high accuracy of 96 percent for numerous numbers of images, binary search has a varying accuracy depending

Table 2: Table for experimental result on execution time and accuracy.

| **Data structure** | **Training time** | **Classification time** | **Total time** | **Accuracy** |
|---|---|---|---|---|
| HRSLinearSearch | 0.077255 | 566.052 | 566.129255 | 0.9691 |
| HRSBinarySearch | 1.06217 | 13.5073 | 90.876583 | 0.9134 |
| TreeSearch | 0.08477 | 31.1293 | 31.21407 | 0.9129 |

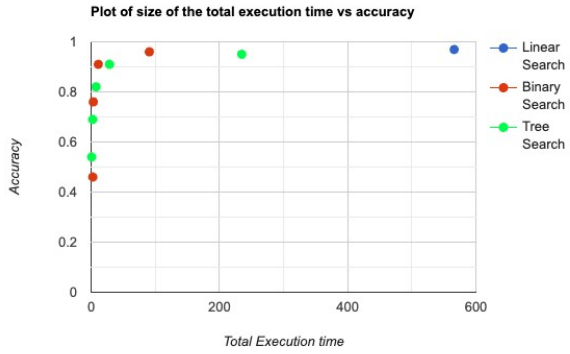


Figure 1: Scatter plot for execution time of training and classification phases. Trend to note: each dot towards the right for respective system represents each increase in k value

on the user-defined k value. This does however mean that a user can play with the relation between execution time and accuracy to meet their application needs. While this relation is discussed in more detail in PA4, note that binary maintains a level of accuracy nearing linear when k reaches 1000.

2. Originally, our tree performance was a bit underwhelming. The total execution time was taking around fourteen minutes for each level of k. With that being said, the tree class was very consistent with the total execution time with not much variance and was also exceptionally accurate for all evaluations, unlike binary search. Comparing this to linear and binary, the tree possesses the worst performance out of the three for the full classification set but matches the high accuracy of linear and when binary has a larger k value. On the contrary, our optimized tree was a complete 180 result compared to the unoptimized version. As you can see in the graph, tree performance varies on k. As k increases, so does the total execution time as well as the accuracy. For lower levels of k, the execution time is rather short, and much of the accuracy is compromised. Functionally, binary search and tree search are very similar. Differences in execution time seem to favor binary search likely due to the overhead of traversing a tree's nodes. Differences in accuracy on the other hand likely were caused by the fact that the tree was not necessarily balanced.

3. One specific application where we might want to use a handwriting recognition system is when a computer needs to transcribe a document, which could be especially useful for people with disabilities. This automatic conversion of text can be crucial to making the multitude of different handwriting styles legible. A cloud server or a laptop would

be the best platform of choice to use. There are several different resources accessible via the cloud or a personal computer that would be able to detect when a handwriting recognition system is appropriate to use. Moreover, when dealing with documents and other important pieces of text, they usually tend to be shared among other people, thus these platforms can store these documents easily. Sometimes handwritten documents are written with haste and can be hard to read/interpret. With a handwriting recognition system, the goal is for this writing to be transcribed to a basic font where everyone would be able to read freely. Depending on the length of the text, a reasonable amount of time for the passage to be transcribed would range anywhere from fifteen seconds to thirty minutes. A book for example with numerous pages would be expected to take fairly long to transcribe and closer to the thirty minute mark, whereas a page or two of a document should only take a matter of seconds. Ideally, the accuracy should be greater than ninety-nine percent for all lengths of text. This is far above what our implementations could achieve, but this could be done with a larger training set and more powerful computers. I would say that if you have a large sample set and time to spare, then the linear system would be your option of choice. Otherwise, tree is more advantageous. As k (the size of the final linear search) grows, so will tree accuracy and the execution time will be faster than linear search. The approximate classification time per image for the full dataset is 31 sec divided by 10k equals about 0.0031 sec per image. For a long piece of text, this is reasonable, and this would be much faster when not running on a limited server. The accuracy would be expected to be 95+ which would maintain its rank among both linear and binary.

# 5 Conclusion

Overall, this assignment was great practice for students to develop their C++ programming skills and also enhance their understanding of the capabilities that generic programming can have on an already functioning class. As previously discussed, we found that tree was more or less on par with binary, both of which completely outclassed linear. Tree had slightly better performance when k less than or equal to 100 with hindered accuracy, but binary had a better performance otherwise. Thus, users of this program should opt to use both binary and tree systems. In terms of linear search, we would only advise using it for smaller datasets (much smaller than the default provided), because then the total time execution would fall in a reasonable range while still having high marks on accuracy. We also did evaluations for when k was in between the given range and we noticed how our trends held true. Additionally, given the data from the table and graph above, we can predict that both the binary and tree systems will still be relatively fast and relevant for ranges greater than the training set of N = 60K and classification set of M = 10K, and of course with varying levels of k.

## Work Distribution

Jack and Brandon both collaborated on the coding aspect and written report. Jack did a lot of debugging and worked extensively on the Tree class and generic programming while

also helping Brandon. Jack also worked on complexity analysis and researched the best optimizations to use given our original code. Brandon worked on the classes too along with the two hand recognition classes and tree, while doing some debugging too. Brandon collected all of the evaluative data and worked on the analysis for optimizations and the qualitative sections of the report. Both students collaborated on the intro and conclusion.