Studying Comparison Based Sorting via Decision Trees – Generation, Visualization, and Algorithmic Properties

Submitted by

A K M Shabab

Student Number: 417032

Submitted to

WG Programming Languages

Department of Computer Science

Rheinland-Pfälzische Technische Universität Kaiserslautern-Landau

67663 Kaiserslautern, Germany

First Examiner: Prof. Dr. Ralf Hinze

Second Examiner: Gregor Cassian Alexandru, M.Sc.

January 15, 2026

Abstract

In this thesis, I study comparison based sorting algorithms via a compact visualization of decision trees. I explain how different sorting algorithms correspond to decision trees and how these trees can be visualized using a circular layout. I generate decision trees for insertion sort, selection sort, bubble sort, quick sort, merge sort, and heap sort, and analyse differences between them. I also examine best case, worst case, and average-case paths in these trees, showing how sorting algorithms behave for different inputs.

Contents

List of Figures

Chapter 1

Introduction

Motivation Sorting is one of the most common problems in computer science. Sorting algorithms are used in many real life applications. For this reason study of sorting algorithms is important to understand their sorting mechanism for using the most appropriate sorting algorithms suitable for real life application. Many popular sorting algorithms work by comparing two elements. These are called comparison based sorting algorithms(eg. insertion sort, selection sort, bubble sort, merge sort, quick sort, and heap sort). Even though these algorithms follow different approach for sorting, they all follow the same basic idea. These algorithms do comparison asking the question " is this element smaller than other one?". And on the basis of the answer of this question the next step: either sort is needed or not. Decision trees give clean way to study this process of comparison based sorting.

For sorting, the information is needed about how the data set has been rearranged compared to the sorted version. When how the the new unsorted order of the data set is known then the sorting can be done by applying the data list permutation in reverse to the input to get the sorted list. The permutation can be seen as a series of swaps, and sorting the permutation means to sorting the swaps to make the permutation sorted means undoing the swaps that made the permutations until the the data list is sorted. This is why comparison based sorting algorithms corresponds to binary decision trees with n! leaves. A Binary decision tree operates by asking (Yes/No) question, and a sorting algorithm can leverage this from a binary decision tree by utilizing the (Yes/No) processing method for swapping purpose to sort the permutations. An in a binary decision tree for "n" inputs there can be $n!$ permutations which a sorting algorithm needs to solve. And this can be shown in decision tree as by solving permutation of a provided input a root to leaf binary decision tree can be created representing that sorting algorithm. So a motivation of this thesis to look at this decision trees to understand sorting algorithms.

A decision tree can visualize all possible comparison steps that algorithm can take. In a decision tree each internal node is a comparison, each branch is a possible result of that comparison, and each leaf represents a final sorted order. This makes decision trees useful for understanding best case and worst case behaviour. The main difficulty is that decision trees can grow very fast. For only a small number of elements, the number of possible permutations becomes very large. This makes the tree hard to understand. Which makes it difficult to compare different algorithms by looking at their trees. This creates the motivation to generate decision trees and to visualize them in a more clear visualization.

Research Objectives This thesis focuses on studying comparison based sorting through decision trees and producing from those trees an understandable visualization. The first objective is generating decision trees for several comparison based sorting algorithms (eg. insertion sort, selection sort, bubble sort, merge sort, quick sort, and heap sort). The second objective is to visualize these trees in a circular layout. The objective of a circular layout is to have a clear visualization for larger trees. Another objective is to compare the structures of the trees of different algorithms and to describe the main differences in their structures. Finally, the thesis aims to identify best case and worst case paths for the sorting algorithms in the decision trees and study how the number of comparisons behaves across different algorithms.

Significance of the Study Study of comparison based sorting algorithms is important because it gives a more clear way to understand the sorting mechanism. Through decision trees the full behaviour of an algorithm over all possible inputs can be seen. This helps visualize and explain how the different comparison based sorting algorithms behave in their sorting process .

This study is also significant because it makes it easier to compare sorting algorithms. When each algorithm is represented as a decision tree and visualized using circular visualization, it becomes easier to see aspect of that algorithms through the decision tree. This can help better understanding and more clear analysis of comparison based sorting algorithms.

Structure of the Thesis This thesis is organized to discuss theoretical foundations, implementation details and visualization methodology, of comparison-based sorting algorithms and their circular visualization. The contents of each chapter discusses :

- **Chapter 2** explains the theoretical background, including comparison-based sorting, the decision trees , lower bound and other basic theoretical aspect of approaches utilised in the implementation process.
- **Chapter 3** explains the core data structures used to represent decision trees in the implementation.

- **Chapter 4** explains the visualization method of trees in hyperbolic plane to fit large trees in a circular layout.
- **Chapter 5** explains the algorithms and comparison based decision tree visualizations for multiple sorting algorithms, including construction and tracing-based reconstruction, redundant nodes and best-/worst-case path calculation.
- **Chapter 6** concludes the thesis by summarizing the main findings and discussing the key insights .

Chapter 2

Theoretical Background

2.1 Basics of Comparison Based Sorting and Algorithms

The Comparison Based Sorting Approach Comparison based sort are algorithms for sorting elements. It depends on pairwise comparisons between elements. The process used in comparison based sorting is done by question of form $a_i \leq a_j$. Classic sorting algorithms, for example: Insertion-sort, Selection-sort, Bubble-sort, Quick-sort, Heap-sort, Merge-sort etc are the most common sorting algorithms following comparison based sorting approach. These sorting algorithms use comparison based sorting approach, where there are some other sorting algorithms those which do not use comparisons between elements. For example Counting sort, Radix sort, and Bucket sort these are some algorithms that do not use comparison based approach for sorting purpose. These algorithms sort data by using extra information about the provided data that needed to be sorted. These sorting algorithms only work when such extra information is available. As these sorting algorithms do not use comparisons they cannot be represented using decision trees.[1]

Decision Tree A decision tree is a binary tree that describes how a sorting algorithm behaves on inputs of a fixed size. Each internal node represents a comparison between two elements, and the two outgoing branches represent the two possible outcomes of that comparison. The decision tree provides a base model for representing comparison based sorting algorithms. For representing comparison based sorting, full binary tree structure with all computational paths is considered. For a decision tree for comparison based sorting algorithm all possible computational paths are considered because each comparison has only two possible outcomes, either it is true or false. Which means comparison based sorting of an n element determines which permutation of the n input positions the current input falls to. Then the decision tree organizes the possible

outcomes in a binary tree which can be called decision tree. For a decision tree, following this comparison based sorting approach to correctly sort n elements, it contains at least $n!$ leaf nodes.[1]. This model acts as a framework for representing any comparison based approach through tree like visualization.

Decision Trees of Fixed Height For n elements, the set of all possible permutations of a list of n elements is the symmetric group S_n, with $|S_n| = n!$. A comparison based sorting algorithm can be represented by a binary decision tree that has leaves representing elements of S_n.

If a decision tree has height h, then the tree can have at most 2^h leaves. Correct sorting requires to know the difference between all $n!$ permutations, as

$$2^h \geq n! \quad \Rightarrow \quad h \geq \lceil \log_2(n!) \rceil.$$

This applies to any comparison-based sorting algorithm and validates the $log_2(n!)$ lower bound on worst-case comparison complexity.[1]

The Lower Bound Theorem Comparison based sorting algorithm follows worst case complexity $log_2(n!)$, because $log_2(n!)$ is a lower bound on the depth of a binary tree with n! leaves [1]. This provides a barrier for comparison based approaches. From decision trees it is visible that they are structured and the factorial function grows in the decision tree in the decision making process. A comparison based sorting algorithm achieves asymptotic optimality[1] when its worst case time complexity is $log_2(n!)$. Algorithms that satisfies asymptotic optimality produces the best solutions for comparison based sorting.

Optimal Decision Trees and Comparison Efficiency A comparison in a binary decision tree has optimal efficiency when minimum depth of the binary tree has $n!$ leaves [2]. The $log_2(n!)$ lower bound are an unavoidable case of information-theoretic constraints and the study of $S(n)$ provides understanding of optimal comparison strategies.

Depth first search (DFS) Depth first search (DFS) is a searching algorithm that can be used as a tree traversal strategy that can search for the shortest path is a tree following a single root to leaf branch. Then backtracks when progress is no more possible. For rooted ordered trees like decision trees, this is similar to a pre order traversal. It visits the root first, then recursively traverse sub trees from left to right [6, 4].

Theoretical Background on Sorting Algorithms Sorting algorithms sorts provided data in a ascending/descending order. Different sorting algorithms performs sorting differently de-

pending on the input size. Many sorting algorithms (eg. bubble, selection, insertion, merge, quick, heap) operate by repeatedly comparing elements. Bellow are some theoretical description of such sorting algorithms

Bubble sort Bubble sort compares neighbouring elements and swaps them on the base of the order of the provided data. This process continues until the provided data set is sorted [1, 3].

Selection sort Selection sort finds the minimum element in the provided data set. Then it swaps it into the next fixed position. The algorithm repeats the process on all the data provided to it until the full sorting process is done [1, 3].

Insertion sort Insertion sort sorts a data set by inserting each new element into its correct position. [1, 16]

Merge sort Merge sort is a divide and conquer sorting algorithm. It splits the data set into halves and repeatedly sorts them. Finally it merges the two sorted halves to form a fully sorted order of the provided data set [1, 5].

Quick sort Quick sort also follows divide and conquer approach. In the sorting process quick sort selects a pivot and then divides the data set around that pivot. Finally it recursively sorts the resulting data set [13, 5].

Heap sort Heap sort uses a heap structure and repeatedly selects/removes the maximum/minimum element while restoring the heap property [1, 2].

2.2 Basics of Visualization theories

H-tree A H-tree is a self similar fractal tree made from perpendicular line segments arranged like the letter "H." While constructing a H-tree at each step creates "H" shapes that added at the endpoints of the current segments. And this pattern of creating H shaped segments keeps repeating in the whole construction process. This recursive process creates a tree like structure.[12]

Poincaré disk The Poincaré disk model is a geometric model of the hyperbolic representation in which all points lie inside a fixed circular area. In this model, hyperbolic geodesics are represented by Euclidean diameters of the disk contained in a circular disk that meet the boundary circle orthogonally. The model preserves angles locally, even though it distorts lengths and distances.[14]

Möbius transformation A Möbius transformation is a mapping process that moves points around on the complex plane together with the point at infinity. It is reversible where no two

different points mapped to the same place and it preserves angles of the edges of the points mapped using this transformation approach. [15]

Chapter 3

Decision Tree Representation: Tree Data Structure

3.1 Core Data Structures for Decision Trees

The foundation of the implementation of this study is built on tree data structures that represents the tree. This data structure represents the nodes in the tree which acts in the decision making process of sorting algorithms.

Abstract Base Class The `DecTree` is the abstract base class that serves as the fundamental backbone for the decision tree representation. This is the mandatory interface that all tree nodes must implements:

```
class DecTree:
    def evaluate(self, arr: Optional[List[Any]] = None) -> List[Any]:
        raise NotImplementedError
```

The `evaluate` represents how a decision tree processes an input array. It also provides a sorted permutation.

Decision Nodes Decision nodes are placed in the internal part of the tree, where the main comparisons happens. The `DecisionNode` class encapsulates the comparison logic that is used in the the sorting process:

```
@dataclass
```

```python
class DecisionNode(DecTree):
    i: int                  # Index for decision comparison
    j: int                  # Another index in the decision
    left: DecTree           # Left subtree if decision is True
    right: DecTree          # Right subtree if decision is False
```

Each `DecisionNode` has two kinds of information used in comparison process: comparison indices (`i` and `j`), which represent the positions in the input array that will be compared.(`left` and `right`) are branch pointers which are the path the sub trees follows based on the comparison outcome

The evaluation logic implements the comparison and branching:

```python
def evaluate(self, arr: Optional[List[Any]] = None) -> List[Any]:
    if arr[self.i] < arr[self.j]:
        return self.left.evaluate(arr)
    else:
        return self.right.evaluate(arr)
```

This recursive process continues until a leaf node is reached. In this process each comparison may reorders the sequence of subsequent comparisons.

Leaf Nodes Leaf nodes represent the final outcomes of the sorting process which is the finally sorter permutations:

```python
@dataclass
class LeafNode(DecTree):
    permutation: List[int]  # Result stored in this leaf
```

The `permutation` variable contains the sorted order. When evaluation reaches a leaf node, the process ends and returns the final permutation:

```python
def evaluate(self, arr: Optional[List[Any]] = None) -> List[Any]:
    return [arr[i] for i in self.permutation]
```

This applies the stored permutation to the input array, providing the sorted output.

Recursive Evaluation Strategy The recursive evaluation pattern is same to the the theoretical decision tree model. Each node only concerns itself with its immediate decision.Then the algorithm continues by handling the two child nodes. This matches the way comparison based sorting algorithms work in a recursive approach.

3.2 Relationship to Theoretical Model

The implementation follows the theoretical rule of the decision tree model discussed in [2][1]. In the decision tree **Internal nodes** works for comparison operations. **Branches** works for visualizing comparison outcomes (True/False). **Leaf nodes** shows the completed permutations. Finally, paths from root to leaves visualizes the execution sorting algorithms.

The tree data structure serves as the foundation for decision tree to represent all analysis, visualization, and algorithm comparison done in this study.This helps to study how different sorting algorithms behave as decision tree for all possible input permutations.

Chapter 4

Drawing binary trees on the hyperbolic plane

This chapter I documented the hyperbolic-geometry approach that is implemented in the code base. Hyperbolic representation was utilised to handle the increasing number of nodes with the value of "n" makes the decision tree more dense with larger number of nodes and connecting edges. Hyperbolic visualization solves the the problem of dense decision tree visualization. In this chapter the approach that is taken in the implementation process is discussed.

4.1 Geometric Idea of Poincaré disk

In the visualization implementation nodes and edges are placed inside the Poincaré disk model. In the implementation, the circular area is the open Euclidean unit disk. The hyperbolic lines are represented by Euclidean circular arcs orthogonal to the boundary circle [7]. The main idea of using the Poincaré disk in the visualization implementation was to use its construction process to place many nodes in a circular area without the edges not intersecting each other, where Möbius transformations is a mapping process used for the disk preserving the hyperbolic geometry[8][9]. From [10] the formula achieved:

$$\delta(0, r) = \frac{1}{2} \log\left(\frac{1+r}{1-r}\right) = \operatorname{artanh}(r), \tag{4.1}$$

and to get the mid point, halving hyperbolic distance:

$$r_{\text{mid}} = \tanh\left(\tfrac{1}{2} \operatorname{artanh}(r)\right). \tag{4.2}$$

These equation is the main mathematical equation implemented in the visualization.

4.2 From theory to implementation

Mapping points to the origin using Möbius transformations In (4.3)[8] the Möbius transformations formula is provided:

$$z \mapsto e^{i\theta} \frac{z_0 + z}{1 + \overline{z_0} z}, \tag{4.3}$$

where the translation part $z \mapsto z_0 \oplus z$ is separated from the rotation. In the code similar special case that maps a chosen point w to the origin is used:

$$\phi_w(z) = \frac{z - w}{1 - \overline{w} z}, \qquad \phi_w^{-1}(z) = \frac{z + w}{1 + \overline{w} z}. \tag{4.4}$$

This formula is the same family as (4.3), obtained by taking $\theta = 0$ and $z_0 = -w$.

Equation (4.4) is a special case of (4.3) obtained by setting $\theta = 0$ and choosing $z_0 = -w$, which removes the rotation and leaves only the disk translation. The implementation is a direct translation:

```
def mobius_map_to_zero(w: complex):
    def phi(z: complex) -> complex:
        return (z - w) / (1 - np.conj(w) * z)
    def phi_inv(z: complex) -> complex:
        return (z + w) / (1 + np.conj(w) * z)
    return phi, phi_inv
```

This Möbius mapping is used to move any point to the center of the Poincaré disk so that distances and curves are easier to compute and draw. The Poincaré distance stays the same after applying Möbius transformations [10, 11].

Calculating midpoints by halving the radial distance After mapping u by ϕ_u, v is $v' = \phi_u(v)$ and lies on some radius. Following that radius, hyperbolic distance is a function of Euclidean radius, similar to (4.1). Therefore the hyperbolic midpoint on the $0 \to v'$ is obtained by halving the hyperbolic distance that can be obtained from (4.2). This is also the same thing that we can get from the implementation code:

```
def hyperbolic_radius_to_mid(r: float) -> float:
```

```
    return np.tanh(0.5 * np.arctanh(r))
```

The full midpoint calculation process is obtained from implementing the two formulas (4.4) and (4.2).

Geodesic drawing To draw an edge between two nodes u and v inside the Poincaré disk, a disk automorphism that sends u to the origin was implemented in the visualization code. After this change, v becomes v'. After this transformation, the connecting line from 0 to v' is simply a radial line. For this reason, it is easy to generate points on it.

Let the total hyperbolic distance from 0 to v' be $D = \delta(0, |v'|) = \operatorname{artanh}(|v'|)$ from (4.1). To create this curve, in the visualization code, it moves with the edge by taking a fraction $t \in [0, 1]$ of this hyperbolic distance. Converting that hyperbolic step back into an ordinary Euclidean radius in the disk gives

$$r(t) = \tanh(tD) = \tanh\big(t \operatorname{artanh}(|v'|)\big), \tag{4.5}$$

which is implemented in the visualization code:

```
atanh_r = np.arctanh(r_vp)
for t in np.linspace(0.0, 1.0, num):
    r_t = np.tanh(t * atanh_r)
    zprime = r_t * cmath.exp(1j * arg)
    pts.append(phi_inv(zprime))
```

The code maps all sampled points using ϕ_u^{-1}. After mapping, the straight radial line becomes the line curve between u and v in the disk. In this implementation, in the Poincaré disk, lines have the diameters that meet the boundary circle at right angles [7, 10], and because Möbius disk preserve the Poincaré distance, so they also preserve lines [10, 11].

Chapter 5

Decision Tree Generation Algorithms

This chapter discusses the approaches and implementation for constructing decision tree representations of various sorting algorithms, including visualization and analysis of their behaviour.

Space of Possible Permutations At any stage during sorting, the algorithm operates on a set of possible permutations consistent with previous comparisons. Let, P be the set of remaining permutations. Each comparison (i, j) divides P into two parts:

- $P_{left} = \{\pi \in P \mid \pi(i) < \pi(j)\}$
- $P_{right} = \{\pi \in P \mid \pi(i) > \pi(j)\}$

Decision Tree Representation of Comparison Based Sorting A decision tree for comparison sorting represents how a sorted order can be achieved by asking a binary question of form "is $x_i < x_j$?". Each internal node corresponds to one comparison, each outgoing edge corresponds to the outcome of that comparison, and each leaf corresponds to sorted order of the input elements. For n elements there are $n!$ possible input permutations. So decision tree must have at least $n!$ leaves.[1].

Decision Tree (Full Comparison Tree) The decision tree implementation in this study is a baseline tree that represents the tree representation for permutation space S_n of size $n!$. The purpose of this tree is to first understand how the full permutations of provided indices as array for n behaves in a circular H-tree before we go deeper into other sorting algorithms.

In the implementation, the decision tree is generated by using a fixed schedule that provides all possible permutations for (i, j) with $i < j$. This schedule is common for all kind of sorting algorithms including full comparison decision tree. The schedule used is:

```python
def generate_all_comparisons(n: int):
    return [(i, j) for i in range(n) for j in range(i + 1, n)]
```

In the full visualization of decision tree in circular H-tree, all $n!$ leaves are placed on the boundary of the disk. Figures 5.1 show the full comparison decision trees for $n = 4$ in the Poincaré disk representation.

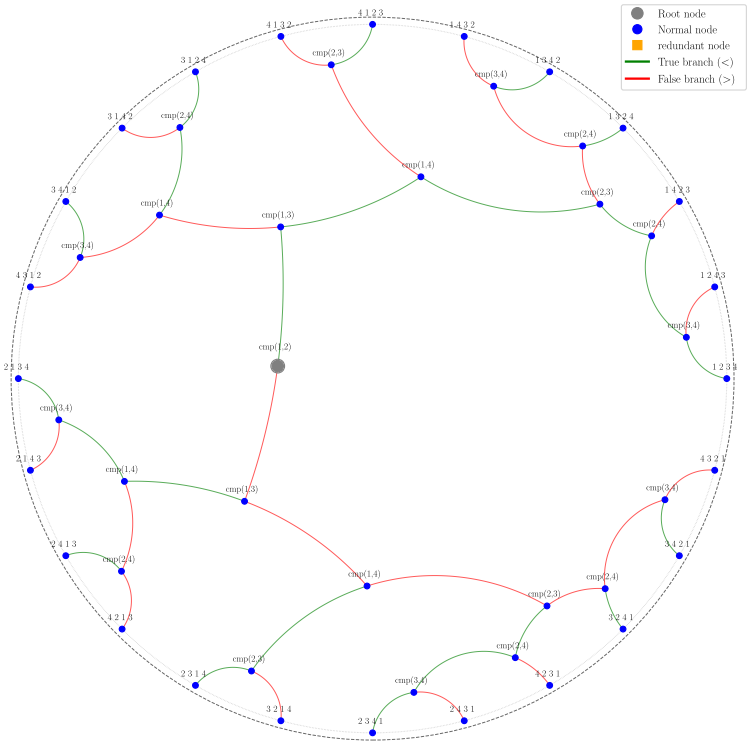

Figure 5.1: Poincaré disk visualization of the decision tree for $n = 4$.

In the tree visualization for decision tree, circular tree for n=4 is visualized. This works as an example for the decision tree representation for comparison tree. And, for the higher value of n, the higher number of nodes, the higher number of edges. And the more edges and nodes in a tree, the denser the circular tree representation gets.

5.1 Construction of decision trees for Uniform comparison sorting and Insertion Sort

The implementation of decision tree for comparison based sorting representation in circular H-tree and the insertion sort decision tree are constructed by directly generating the branching structure for permutations of identities $\{0, 1, \ldots, n-1\}$.

For constructing comparison tree for decision tree, construction starts from the complete permutation set and repeatedly applying comparisons that split the remaining possible permutations. The implementation begins by:

```
perms = list(permutations(range(n)))
```

and uses a fixed schedule of all index pairs (i, j) with $i < j$:

```
fixed_sequence = [(i, j) for i in range(n) for j in range(i + 1, n)]
```

At recursion step `seq_index`, the current comparison (i, j) partitions the current possible set into two subsets. This depending on whether i appears before j in the permutation.

```
i, j = fixed_sequence[seq_index]
left  = [p for p in perms if p.index(i) < p.index(j)]
right = [p for p in perms if p.index(i) > p.index(j)]
```

In this process, when both subsets are non empty, the recursion produces a binary `DecisionNode`. And when a subset is empty, recursion continues with the non empty side. The recursion stops once a single possible set is reached:

```
if len(perms) == 1:
    return LeafNode(permutation=list(perms[0]))
```

This after processing the nodes and leaves through the circular representation approach we can get a circular comparison tree.

Comparison tree for insertion sort is generated differently than comparison tree for decision tree. For creating comparison tree for insertion sort, it does not split S_n directly. The insertion sort tree is built by recursively constructing the tree for the sorted prefix. Then it expands each leaf by inserting the next element into that prefix order. In the expansion step, a leaf representing a sorted prefix permutation, the sorting process tests next positions until the new element is placed. The implementation is similar to the previously discussed process:

```
current_tree = LeafNode(permutation=[idx_to_insert] + perm)
for k in range(1, len(perm) + 1):
```

```
    i, j = idx_to_insert, perm[k - 1]
    left = current_tree
    right = LeafNode(permutation=perm[:k] + [idx_to_insert] + perm[k:])
    current_tree = DecisionNode(i=i, j=j, left=left, right=right)
```

From here we can get the comparison rule that is used in generating comparison tree for insertion sort. Each additional comparison corresponds to testing , that if the inserted element should appear before or next to prefix element, and each leaf corresponds to a final position sorted by using insertion sort. Applying this process recursively over the entire prefix tree provides the complete insertion sort decision tree for size n. And by using this process and presenting them through the Poincaré disk we can get the circular representation for insertion sort.

5.2 Construction of comparison trees using tracing for other sorting algorithms

Unlike previously discussed, for creating comparison tree for full comparison decision tree and insertion sort, the other sorting algorithms that are tested in this study (bubble sort, merge sort, quick sort, selection sort, heap sort) follow a similar framework creating comparison tree. The main idea is to run each sorting algorithm on all permutations of size n. In this approach every comparison is recorded in terms of original element indices. Then reconstruct the decision tree from these comparison paths.

Core Idea For a input size n, the sorting algorithm runs on all permutations of the input elements. During algorithm execution, every comparison is traced and recorded in terms of the *original input indices*. Each execution induces a root to leaf path in the algorithm's comparison decision tree. The complete tree is obtained by merging all paths and identifying shared prefixes.

Tracking Original Element Identities Since sorting algorithms operate by frequently swap elements, array positions cannot be used to reliably identify which original inputs are being compared. To solve this problem, a *bookkeeping permutation* is used to manage the working array.

Initially, the bookkeeping permutation is the identity. Whenever the algorithm swaps two array positions, the same swap is applied to the bookkeeping permutation. When a comparison is performed, the bookkeeping permutation maps the current positions back to the corresponding original input indices.

The main comparison tracing logic used in bookkeeping process:

```
def compare(self, i: int, j: int) -> bool:
    orig_i = self.bookkeeping_perm[i]
    orig_j = self.bookkeeping_perm[j]
    result = self.arr[i] <= self.arr[j]
    self.comparison_log.append((orig_i, orig_j, result))
    return result
```

Each comparison is recorded as a triple (i, j, b), where i and j are original input indices and b is the comparison outcome. This ensures that the traced comparisons reflect the logical structure of the algorithm.

Comparison Paths For each input permutation, tracing produces the final output permutation along with the sorted sequence of recorded comparisons. This sequence defines a comparison path through the decision tree.

Decision Tree Reconstruction The decision tree is reconstructed by recursively merging all comparison paths. If all remaining paths correspond to the same output permutation, a leaf node is created. Otherwise, all paths shares the same next comparison, which labels the current internal node. The paths are then split according to the comparison outcome and processed recursively.

The core structural operation:

```
i, j, _ = paths[0][1][0]
left_paths  = [p for p in paths if p[1][0][2]]
right_paths = [p for p in paths if not p[1][0][2]]
```

Each internal node is labelled by a pair of input indices (i, j). The left and right subtrees correspond to the outcomes " <" and ">" respectively. Unary nodes may occur when all executions take the same branch.

Sorting as identifying the input permutation In the implementation process, the input elements move around in their swapping process, so their original positions are lost unless tracked. For this reason, the implementation following this framework for constructing comparison trees for sorting algorithms. Whenever any algorithm swaps two positions, the same swap is applied to the bookkeeping permutation that is a part of this framework. This ensures that it is always known that which original elements are currently sitting in each array position.

```
def swap(self, i: int, j: int):
    self.arr[i], self.arr[j] = self.arr[j], self.arr[i]
    self.bookkeeping_perm[i], self.bookkeeping_perm[j] = (
        self.bookkeeping_perm[j],
        self.bookkeeping_perm[i],
    )
```

If the swap process gives the sorted order of the inputs, then applying the same swaps in reverse order turns the sorted array back into the original input.

```
def apply_swaps(arr, swaps):
    arr = arr.copy()
    for i, j in swaps:
        arr[i], arr[j] = arr[j], arr[i]
    return arr

def invert_swaps(swaps):
    return list(reversed(swaps))
```

The bookkeeping permutation framework records permutation of the input. Once the swapping process is finished, the bookkeeping permutation stores the final sorted order of the original elements. That permutation is then saved in a leaf of the decision tree.

So sorting is just finding which permutation of the sorted list the input belongs to. The swap history builds that permutation, bookkeeping saves element identities and the decision tree stores the result. Once the permutation is known, the list is already sorted.

Outcome The resulting structure following this framework, for different sorting algorithms produces is an explicit comparison decision tree that is sorted following the sorting approach following the required sorting algorithm. In this comparison process Internal nodes represent comparisons between original input elements, and leaves correspond to output permutations.

Merge sort as an exception to swap-based bookkeeping The bookkeeping framework described above follows a swap based approach that goes best with sorting algorithms such as bubble sort, selection sort, quick sort, and heap sort. This is because these algorithms sort data by swapping two data positions. After this the bookkeeping permutation updates applying the same swap to the identity labels.

Merge sort is different from those algorithms because it usually does not swap elements. Instead, it sorts recursively and then merges two sorted halves by copying elements into a temporary list and writing them back into the array. Because of this, merge sort did not use the the swap process in the implementation while using the bookkeeping framework. Instead, during the merge step, the algorithm copies both the values and their bookkeeping labels into a second temporary list. When the merged result is written back, the bookkeeping labels are written back in the same order.

```
# during merge: copy both the value and its original-identity label
if tracer.compare(i, j):                # arr[i] < arr[j]
    temp_vals.append(tracer.arr[i])
    temp_ids.append(tracer.bookkeeping_perm[i])
    i += 1
else:
    temp_vals.append(tracer.arr[j])
    temp_ids.append(tracer.bookkeeping_perm[j])
    j += 1

# write back merged segment (value and identity stay aligned)
tracer.arr[left + k] = temp_vals[k]
tracer.bookkeeping_perm[left + k] = temp_ids[k]
```

This is the difference with the swap based algorithms approach to merge sorts approach. In bubble, selection, quick, and heap sort, element identities are preserved by swapping bookkeeping entries when two data are swapped. In merge sort implementation, elements are sorted by copying instead of swapping.

5.3 Unary nodes/Redundant comparison nodes and its visualization

While constructing a decision tree for sorting algorithm from bookkeeping approach discussed previously, the resulting structure is not always a full binary tree. In those trees *unary* internal nodes can appear which are nodes that have only one non empty child.

Why unary nodes occur Unary nodes are created in the comparison trees when a comparison is *redundant* with respect to the set of possible permutations remaining in the end. For example,

after early comparison, all still possible inputs already imply the outcome of the next comparison (i, j). Then every traced path takes the same branch :all True or all False. So the other branch is impossible and the reconstructed node has only one child.

This effect is especially common in bookkeeping approach following algorithms because they keep making comparisons even after the final sorted order is determined or they compare elements whose order is already sorted by earlier comparisons.

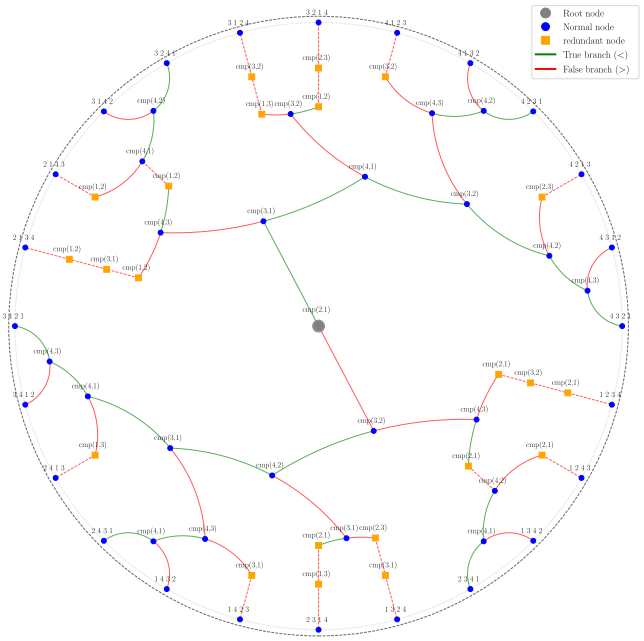

Figure 5.2: Comparison decision tree for $n = 4$ of Bubble sort showing unary (redundant) nodes.

How unary nodes appear in the implementation In the implementation, all paths were merged by their common prefix comparison. If one outcome never occurs among the remaining paths, the corresponding subtree is empty. In the compact representation, such forced comparisons can be skipped. However, in the implementation process the unary nodes were kept so that redundant nodes remains visible in the tree.

The following code shows the logic used in the code implementation: paths that have already finished their trace are duplicated to both branches, and paths with a forced outcome all flow into only one branch. This produces unary nodes in the comparison tree.

```
for perm, path in current_paths:
    if not path:
        # Finished trace: this comparison is redundant for this permutation.
        # Duplicate into BOTH branches so the forced node is still rendered.
        left_paths.append((perm, path))
```

```
            right_paths.append((perm, path))
        else:
            went_left = path[0][2]
            if went_left:
                left_paths.append((perm, path[1:]))
            else:
                right_paths.append((perm, path[1:]))

# If one side ends up empty, the node is unary in the reconstructed tree.
```

Figure 5.2 shows a full comparison tree for $n = 4$ for bubble sort as an example. unary nodes for all the sorting algorithms following the bookkeeping framework are all rendered similarly. Bubble sort is here taken as an proper example where the unary nodes are clearly visible and understandable. I n the implementation unary nodes appear as orange square markers with dashed outgoing edges. These mark redundant comparisons where only one outcome is possible.

5.4 Patterns Revealed by Hyperbolic Visualization

In this part, the discussion is on the structures visualized in the implementation process. By reviewing the the structures present in the comparison based sorting algorithms decision trees, it is visible that some patterns are available in those structures and each tree might present a repetitive patters in that decision tree.

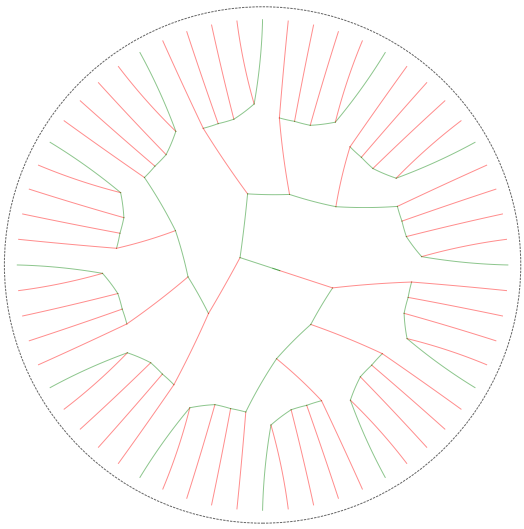

Figure 5.3: Half comparison tree for insertion sort with $n = 5$.

Patterns in Insertion Sort comparison tree Based on the visualization in Figure 5.3, the insertion sort decision tree follows a distinct, regular pattern. The tree is not a standard binary tree but a denatured binary tree. From Figure 5.4 which represents the decision tree and factorial tree, it is clear that the insertion sort decision tree (left) is a structured approximation of the factorial tree (right). While the factorial tree on the right shows branch equality at each level, the insertion sort tree on the left follows this structure. In the insertion sort decision tree each new level of the tree introduces a new degree of branching. This means with sorting progressing each layer the tree grows new branch per node than the layer before it.

Figure 5.4: Comparison of insertion sort decision trees in tree visualization. Left: The full denatured decision tree for insertion sort. Right: A factorial tree structure.

From Figure 5.4 and Figure 5.3 it is visible for insertion sort that, if the sorting is done in several levels and in the root node is the final output, then from the visualization, the first level is sorting a list of two elements. Then the second level corresponds to inserting a new element into a sorted 2-element list, then the third to inserting a new element into a sorted 3-element list and so on. It can be observed that, it represents exactly the behaviour of insertion sort algorithm, that is if the element is smaller insert it and stop. If it is larger then continue. This is the exact process what insertion sort follows in its sorting process.

Patterns in other sorting trees In this study, other sorting algorithms like bubble sort, merge sort, quick sort, selection sort, heap sort were studied. From reviewing the comparison based trees built for all that sorting algorithms, it was clear that the sorting algorithms show some kind of similar sub-tree pattern in them for value of $n = 4, 5, 6$. This kind of pattern is not visible in $n = 3$ as for this the permutation is very small, so extracting any pattern from them was not easy. Following is a detail discussion on this. While discussion half tree visualization of all the sorting algorithms were considered as all the algorithms are mirror symmetric in the both half of them.

For the **Bubble sort** comparison decision tree, the comparison is a continuous scan and swap/no-swap process to achieve sorted order. But after observing Figure 5.5 it is visible that the overall *tree shape* is not regular because of swap/no-swap comparison, which is a behaviour of bubble sort. Figure 5.5 shows that many paths share the same early comparison steps but they separate at different points depending swaps. This provides an irregular structure, which results bubble sort having no regular sub-tree patterns. From the visualization of Figure 5.5

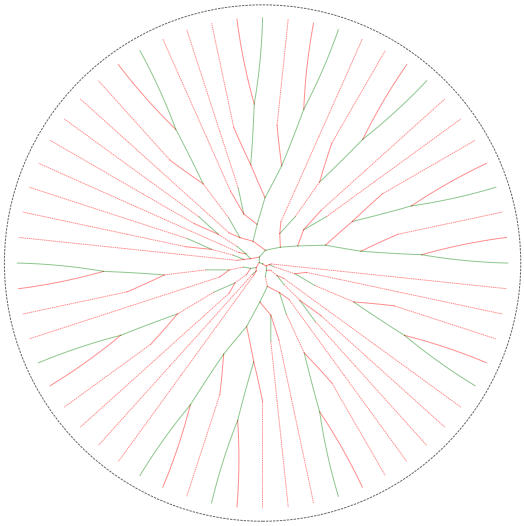

Figure 5.5: Half comparison tree for bubble sort with $n = 5$.

it is understandable by looking that internal node refers to one comparison. Each branch of the decision tree refers to the comparison outcome. The *dotted lines* highlight **redundant nodes/comparisons** in the visualization. These are comparisons that do not any new swapping because their outcome is already determined by earlier comparisons. These dotted lines representing redundant nodes/comparisons can be conceptually removed without changing correctness for that path. So, after this analysing it can be finally said for bubble sort that the structure shows bubble sort's repeated swapping/-no-swapping schedule.

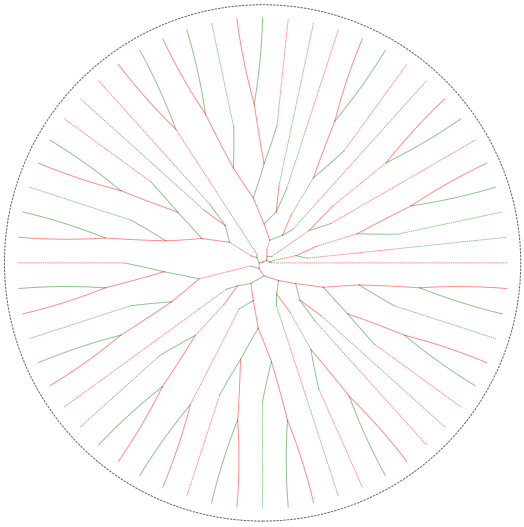

Figure 5.6: Half-tree visualization of Selection Sort comparison tree for $n = 5$.

The **selection sort** comparison decision tree does not show regular and repeated repeated pattern in Figure 5.6 comparison tree representation. Selection sort processes the sorting approach by a "scan for the minimum" approach. But the *current minimum choice* changes depend-

ing on comparison. In the decision-tree representation, each internal node refers to comparing the current data minimum with a new element. Each branch in decison tree represents if the data swapped. Therefore, different root-to-leaf paths visualize different candidate updates. This creates a tree that appears irregular. The leaf nodes refers to the final outcomes of comparisons. The *dotted edges* in Figure 5.6 refers to **redundant comparisons** in the visualization similar to what we see in bubble sort tree , where they do not contribute to sorting as their sorting process is already done in previous steps.

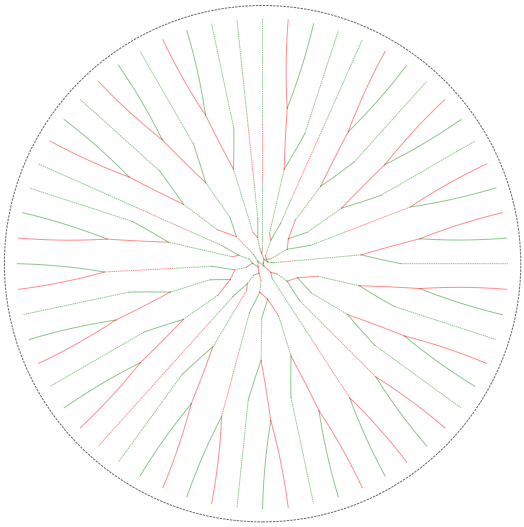

Figure 5.7: Half-tree visualization of Heap Sort comparison tree for $n = 5$.

For the comparison decision tree of **Heap sort**, Figure 5.7 does not show regular pattern. In the decision-tree, each internal node represents one local comparison between a node and one of its children. Each branch represents the outcome that either keeps the heapify process at the current level or pushes it down. As the path taken during heapify depends on previous swaps and the current heap, many branches are shown to show irregular pattern. This makes the tree have less regular pattern in visualizations. The dotted edges in the visualization indicate redundant comparisons whose outcomes are already performed by previous comparisons.

From the comparison decision tree of **Merge sort**,there is no regular repetitive patter that describes the the divide and merge approach of this algorithm. The visualized tree for merge sorts shows that merge sort merge sort even though having a irregular pattern it does not have a structured branching process that we can see in the decision tree visualization of Insertion Sort. This can be seen from the Figure 5.4 which is a half tree representation of merge sort for n=5.[1]

For the comparison decision tree of **Quick sort**, the repeated pattern is a pivot where chain of comparisons all involve the same pivot before the algorithm splits and does recursion.The tree is not balanced , which is observed from Figure 5.9 as some pivot outcomes create smaller

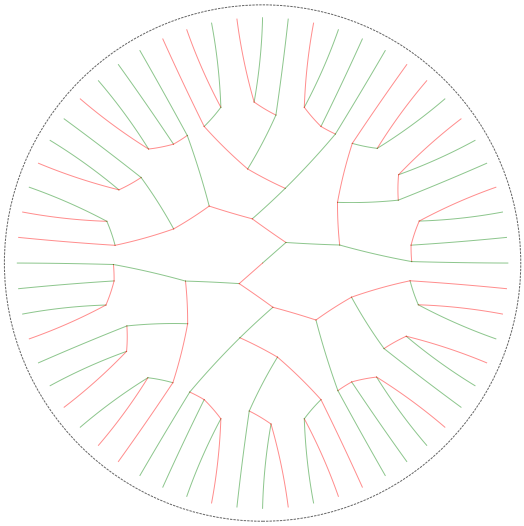

Figure 5.8: Half-tree visualization of Merge Sort comparison tree for $n = 5$

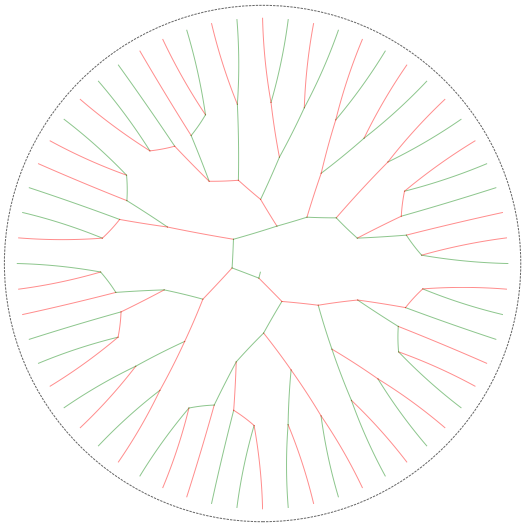

Figure 5.9: Half-tree visualization of Quick Sort comparison tree for $n = 5$ (paths-only view).

sub-trees, while others creates big sub-tree. This sub-tree structure is same to the nature that quick sort follows while sorting.

After analysing the decision tree visualizations of all the sorting algorithms, it can be observed that Bubble Sort, Selection Sort, and Heap Sort have dotted edges, which are redundant comparisons nodes. On the other hand, Merge Sort and Insertion Sort show no redundant nodes and displays regular decision tree structure. Among the algorithms that has redundant nodes, Bubble Sort has the highest number of redundant comparisons, Selection Sort shows fewer redundant comparisons, and Heap Sort shows the fewest redundant comparisons.

5.5 Best and Worst Case Paths in Comparison Decision Trees

After a comparison decision tree for sorting algorithms are generated, the best and worst case number of comparisons can be computed from the tree. The method used in the implementation is a **depth first search (DFS)** over the tree to calculate the root-to-leaf paths. This process is followed by selecting the shortest and longest paths by path length in the tree.

Use of DFS over the rendered tree The implementation builds an list `children` from the constructed tree. It then identifies leaf nodes, and then uses **DFS** to travel all complete paths. A node is treated as a leaf if it is in `leaf_ids` or has no children. This process also covers unary endings in the comparison tree.

```
def dfs(u, path):
    if u in leaf_ids or u not in children or len(children[u]) == 0:
        paths.append(path[:])
        return
    for v in children[u]:
        path.append(v)
        dfs(v, path)
        path.pop()
```

Each collected `path` is a list of node IDs from the root to one leaf.

Selecting best and worst paths : After all root-to-leaf paths are collected, the best case and worst case are found by taking the minimum and maximum path lengths. If multiple paths tie for best or worst, all of them are kept.

```
for path in paths:
    length = len(path) - 1  # comparisons = edges
    if length < best_len:
        best_len = length
        best_paths = [path]
    elif length == best_len:
        best_paths.append(path)
```

```
    if length > worst_len:
        worst_len = length
        worst_paths = [path]
    elif length == worst_len:
        worst_paths.append(path)
```

Mapping paths to edges for highlighting To highlight best and worst paths in the visualization, each path is converted into a set of directed edges (u, v). The union of all best (and worst) edges is then highlighted with thicker line width for best case (and dashed thick line highlighting for work case paths).

```
def path_to_edges(path):
    return {(path[i], path[i+1]) for i in range(len(path)-1)}
```

This DFS based analysis gives a direct and proper best-case and worst-case comparison count for the algorithm being visualized. The best case corresponds to the shortest path for the leaf reachable from the root for a sorting algorithms, and the worst case corresponds to the longest path from root to leaf.

Figure 5.10 shows the shortest and longest execution paths of Quick Sort for $n = 5$ inside the comparison decision tree. from the figure for quick sort best and worst case calculation s clearly seen for n=5 comparison tree for quick sort. This calculation follows the DFS algorithm to search the best and worst path from root to leaf . It is also visible from the figure that if there are multiple paths that corresponds to the best and worst path those all are properly visualized in the figure clearly.

Input sensitivity of different algorithms I call an algorithm with a large spread in the between their worst and average case complexity [1, Chapter II] *input sensitive*. This an algorithms whos running time changes significantly depending on the structure of the input. Here from the comparison trees we can verify the difference between an algorithms run time depending on how many comparison a sorting algorithm in their best worst case paths. In the comparison base decision tree , when there is a large gap between the comparison done by best case paths and the worst case path, then it means that sorting algorithm is input sensitive. If best and worst cases do similar amount or the comparison done is very close to each other then the algorithm is not input sensitive because the input order barely affects performance of the sorting algorithm.

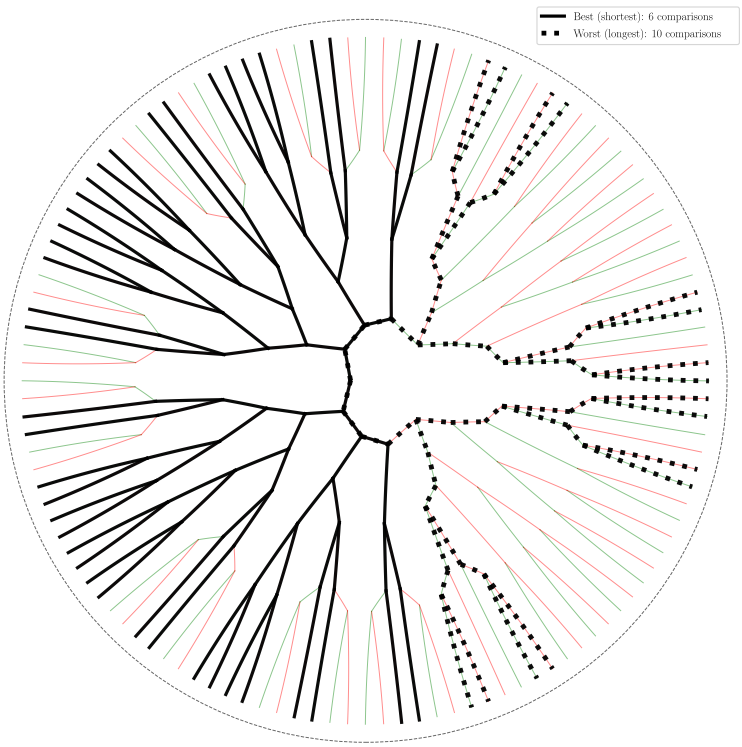

Figure 5.10: Best and worst case execution paths of Quick Sort for $n = 5$

By looking at the visualizations of the best worst path of the sorting algorithms , for Bubble and selection sort they have all the comparisons highlighted as best and worst path. This means that this two sorting algorithms do similar number or comparisons for both both best and worst case comparisons, regardless of how much the provided data list is sorted or not which make this two algorithms not input sensitive. For Insertion sort the number of best case comparisons are small with regard to worst case path, but it makes 4 comparisons for best case path, where 10 comparisons for worst case paths, which is a big difference for best and worst case comparison numbers. So Insertion sort can be detected as input sensitive, as the nature of the input (how sorted is the provided input indices) can effect the performance of Insertion sort. For heap sort the best worst case comparison numb er are relatively tight, there are close to similar number of comparisons for best and worst case. But the number of best case path is still smaller than the worst case path, so this means the heap sort also dependent on the input indices, so it can be said for heap sort that it is not that much input sensitive.For quick sort Figure 5.10 the number of comparison best case taken (six) is a smaller than worst case(ten). So it can be understood from seeing that quick sort can be flagged as very dependent on the provided input indices , so it can be detected as input sensitive. Finally the case of merge sort is mostly similar to heap sort. For quick sort too, the best worst case comparisons are relatively tight, there are close to similar number of comparisons for best and worst case. So , it can be said that quick sort also,

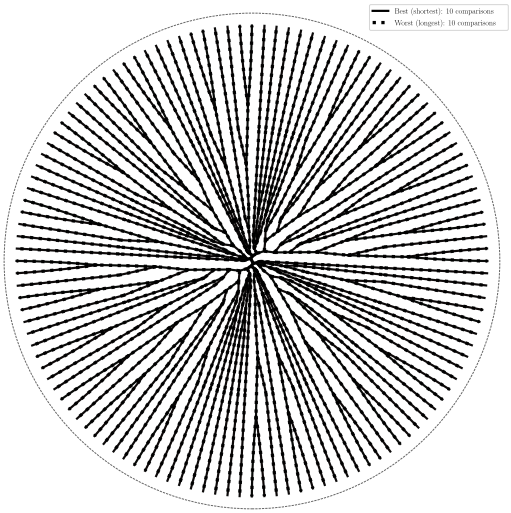

Figure 5.11: Bubble Sort best/worst case visualization

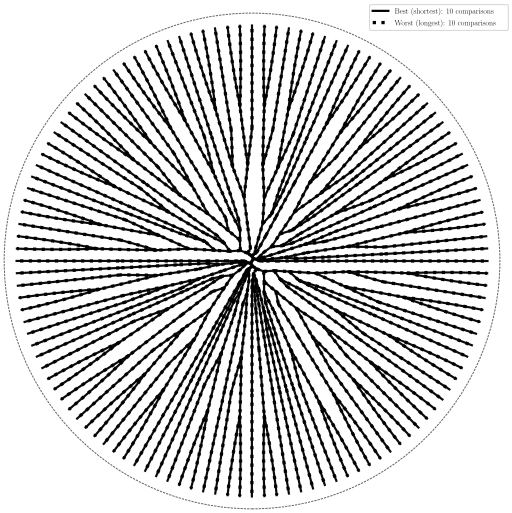

Figure 5.12: Selection Sort best/worst case visualization

like heap sort is not input dependent.

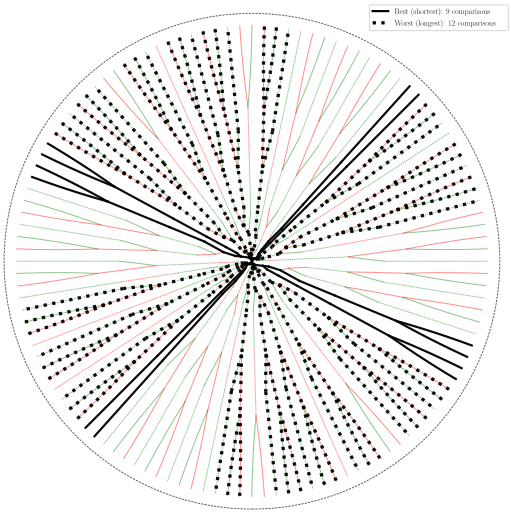

Figure 5.13: (a) Heap Sort best/worst case visualization

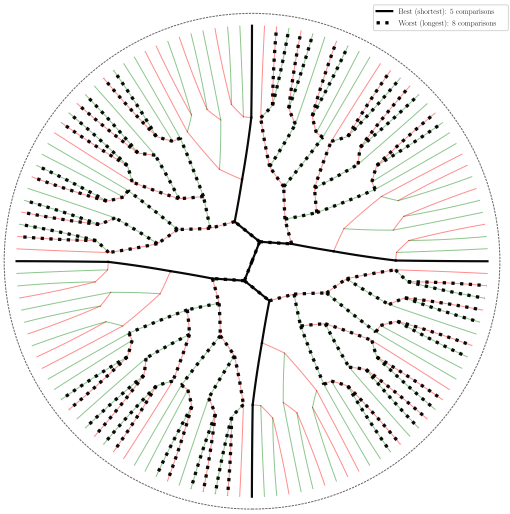

Figure 5.14: (b) Merge Sort best/worst case visualization

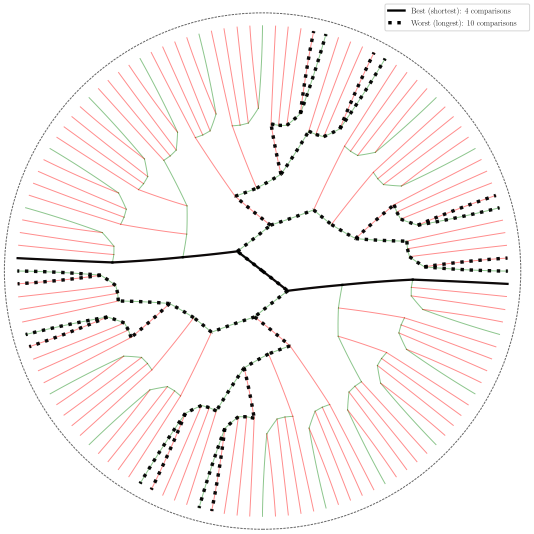

Figure 5.15: Insertion Sort best worst case visualization

Chapter 6

Conclusion

In this thesis I looked into comparison based sorting algorithms using decision trees and visualized them in hyperbolic plane utilizing the Poincaré disk model. On the algorithm side, I built decision trees for several sorting methods. For insertion sort, the trees were built directly by splitting the set of possible permutations. For bubble sort, selection sort, merge sort, quick sort, and heap sort, the approach was to run them on every possible input permutation, every comparison was recorded, and then the decision tree was reconstructed from those recorded comparison paths. This visualization implementation reveals many algorithms keep doing comparisons even when the outcome is already reached by earlier comparisons. In the decision tree these show up as nodes with only one possible branch (unary nodes). Instead of removing them, I kept kept them visible in the implementation so the visualization shows where redundant work happens. The visualization implementation also made structural differences between algorithms easier to see.

Finally, I used the generated trees to compute best case and worst case comparisons by searching all root to leaf paths. It also shows how sensitive each algorithm is to input order.

Overall, in this thesis I tried to understand comparison based sorting via circular visualization of decision tree. Circular representation was studied for decision trees as main problems with normal decision trees are they become too large to understand with normal drawings. Hyperbolic visualization solves that problem and makes it possible to compare algorithms by looking at their tree structure, redundancy, and best/worst case behaviour in a visual way.

Chapter 7

Acknowledgements

I would like to thank Prof. Dr. Ralf Hinze for taking me under supervision and for guiding the research of this thesis.

I would also like to thank Gregor Cassian Alexandru, M.Sc., for co-supervision and for strong support in implementation and writing, for important insight for the Section 5.2 *Construction of comparison trees using tracing for other sorting algorithms*, especially the bookkeeping method for tracing comparisons, tracking swaps, and building the comparison decision tree from recorded executions, for important insight for the Section 5.4 *Sub-tree Patterns Revealed by Hyperbolic Visualization*, especially the visualization of the insertion sort decision tree pattern as a structure placed on top of the factorial tree and helping draw the Figure 5.4 left side part. I would also like thank Gregor Cassian Alexandru, M.Sc. for guidance on organizing the thesis and improving the writing during the full process.

Bibliography

[1] T. H. Cormen, Ed., *Introduction to Algorithms*, 3rd ed. Cambridge, MA: MIT Press, 2009.

[2] D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*. Reading, MA: Addison-Wesley, 1998.

[3] A. Levitin, *Introduction to the Design and Analysis of Algorithms*, 3rd ed.

[4] M. T. Goodrich and R. Tamassia, *Algorithm Design: Foundations, Analysis, and Internet Examples*. New York: John Wiley & Sons, 2001 Boston, MA: Addison–Wesley (Pearson), 2012.

[5] R. Sedgewick and K. Wayne, *Algorithms*, 4th ed. Upper Saddle River, NJ: Pearson Education, 2011.

[6] S. S. Skiena, *The Algorithm Design Manual*, 2nd ed. London: Springer-Verlag London Limited, 2008.

[7] D. Veres, "Constructions on the Poincaré-disk," *Acta Mathematica Hungarica*, vol. 176, no. 2, pp. 437–446, 2025.

[8] C. Barbu and L.-I. Pişcoran, "The orthopole theorem in the Poincaré disc model of hyperbolic geometry," *Acta Univ. Sapientiae, Mathematica*, vol. 4, no. 1, pp. 20–25, 2012.

[9] O. Demirel and E. Soytürk, "The hyperbolic Carnot theorem in the Poincaré disc model of hyperbolic geometry," *Novi Sad Journal of Mathematics*, vol. 38, no. 2, pp. 33–39, 2008.

[10] C. Bisi and G. Gentili, "Möbius Transformations and the Poincaré Distance in the Quaternionic Setting," *Indiana University Mathematics Journal*, vol. 58, no. 6, pp. 2729–2764, 2009.

[11] L. V. Ahlfors, "Möbius Transformations and Clifford Numbers," in *Differential Geometry and Complex Analysis*, I. Chavel and H. M. Farkas, Eds. Berlin, Heidelberg: Springer, 1985. Available at: `https://doi.org/10.1007/978-3-642-69828-6_5`.

[12] H. A. Lauwerier, *Fractals: Endlessly Repeated Geometrical Figures*, Princeton Science Library. Princeton University Press, 1991. Available at: https://cir.nii.ac.jp/crid/1971993809811556877.

[13] R. Sedgewick, "Implementing Quicksort Programs," *Communications of the ACM*, vol. 21, no. 10, pp. 847–857, Oct. 1978.

[14] É. Charpentier, É. Ghys, and A. Lesne, Eds., *The Scientific Legacy of Poincaré*, trans. J. Bowman. Providence, RI: American Mathematical Society; London: London Mathematical Society, 2010.

[15] R. Kennes, "Computational Aspects of the Möbius Transformation of Graphs," IEEE Transactions on Systems, Man, and Cybernetics, vol. 22, no. 2, pp. 201–223, Mar.–Apr. 1992.

[16] M. A. Bender, M. Farach-Colton, and M. A. Mosteiro, "Insertion Sort is $O(n \log n)$," *Theory of Computing Systems*, vol. 39, pp. 391–397, 2006. doi: 10.1007/s00224-005-1237-z.