HPPS 2025 – Assignment 3

Brute-force and k-d Tree k-NN

¡Your Name¿

December 12, 2025

1 Introduction

In this assignment I implemented the k-nearest-neighbour (k-NN) algorithm in C in two different ways: a simple brute-force method and an accelerated method using a k-d tree. The goal was both to obtain correct results and to study the performance trade-offs between these implementations on different workloads.

The brute-force implementation has $O(n)$ time per query, where n is the number of reference points. The k-d tree implementation has an $O(n \log n)$ preprocessing cost to build the tree, but then can answer each query in $O(k \log n)$ expected time for small k, by pruning large parts of the search space. The provided code skeleton included I/O, a generic quicksort implementation, and driver programs, while the core logic of I/O, brute-force k-NN, and k-d tree construction/querying had to be filled in.

2 Implementation

2.1 Data format and I/O

The handout defines a simple binary format for both point sets and index files. My `io.c` implementation reads and writes these formats as:

- *Points file*: first two 32-bit integers n and d (number of points and dimensionality), followed by $n \cdot d$ doubles in row-major order.
- *Indexes file*: first two 32-bit integers n and k (number of queries and neighbours per query), followed by $n \cdot k$ 32-bit integers containing the neighbour indexes.

On reading, I allocate the necessary arrays with `malloc` and return `NULL` on any short read or invalid size. On writing, I return 0 on success and 1 on any `fwrite` failure. This matches the interface in `io.h` and allows the driver programs to give clear error messages.

2.2 Distance computation and neighbour maintenance

The utility module `util.c` provides two core helpers:

1. **Euclidean distance**:

$$\text{distance}(x, y) = \sqrt{\sum_{i=0}^{d-1} (x_i - y_i)^2}.$$

2. **Maintaining the k closest neighbours** in an array `closest` of indices, where `-1` denotes an empty slot.

The function `insert_if_closer()` maintains `closest` as a sorted sequence in *increasing* order of distance from the query point:

- If `closest` contains fewer than k valid entries (some `-1`s), the candidate point is always inserted.
- If `closest` is full, I find the farthest current neighbour; the candidate replaces it only if it is closer.
- After any change, I perform a simple insertion sort on the k entries to keep them ordered by distance.

This simple $O(k^2)$ maintenance is acceptable in practice because k is typically small (e.g. $k = 1, 5, 10$), while n and the number of queries can be very large.

2.3 Brute-force k-NN

The brute-force implementation is very direct:

1. Allocate an array `closest` of length k and initialise all entries to `-1`.
2. Loop over all n reference points.
3. For each point i, call `insert_if_closer()` with the candidate index i.

At the end of the loop, `closest` contains the indexes of the k nearest reference points to the given query. The function returns this array, and the caller is responsible for freeing it. This implementation has $O(n)$ work per query and is very simple to reason about.

2.4 k-d tree construction

The k-d tree is built recursively from an array of point indexes:

- At recursion depth `depth`, I choose the splitting axis as `axis = depth % d`.
- I use the provided generic quicksort (`hpps_quicksort`) to sort the subarray of indexes by the coordinate of the corresponding points along `axis`.
- I select the median index as the point stored at the current node. This yields a reasonably balanced tree.
- The left child is built recursively from the indexes before the median; the right child from the indexes after the median.

Each tree node stores:

- `point_index`: the index into the original points array,
- `axis`: the splitting dimension,
- pointers to `left` and `right` child nodes.

A separate `kdtree` struct stores the dimensionality, a pointer to the original points array, and the root pointer. The tree does not copy point coordinates, which keeps memory overhead moderate.

2.5 k-d tree search

The recursive search function follows the algorithm from the assignment:

- At each visited node, the node's point is treated as a candidate and inserted into `closest` with `insert_if_closer()`.
- After any update, I recompute the current radius, defined as the distance from the query to the *farthest* point currently in `closest`. If fewer than k neighbours have been found so far, the radius is treated as infinity.
- Let a be the node's splitting axis. I compute

  $$\text{diff} = \text{node\_coord}_a - \text{query}_a.$$

  This tells me on which side of the splitting hyperplane the query lies.
- The recursive traversal uses the following conditions:

  $$\begin{aligned}\text{visit left child} &\iff \text{diff} \geq 0 \;\vee\; \text{radius} > |\text{diff}|,\\ \text{visit right child} &\iff \text{diff} \leq 0 \;\vee\; \text{radius} > |\text{diff}|.\end{aligned}$$

Crucially, the second condition sees any updates to the radius that happened while exploring the first child, so the pruning radius tightens dynamically as closer neighbours are found.

This implementation guarantees we always visit the subtree containing the query (the "near" side). The other subtree (the "far" side) is visited only if the ball of radius `radius` around the query intersects the splitting hyperplane, meaning there could be a closer point on the far side.

3 Correctness and Testing

I used several complementary strategies to gain confidence in correctness:

3.1 Visual inspection (2D)

For small two-dimensional datasets I generated point sets and queries using `knn-genpoints` and then:

1. Ran `knn-bruteforce` to compute neighbour indexes and wrote them to disk.
2. Visualised the result using `knn-svg`, which draws (i) the reference points, (ii) the query points, and (iii) a circle around each query whose radius equals the distance to its kth neighbour.

For small n and k I manually verified that the circles encompassed exactly the intended neighbours and that no obviously closer points were left out.

3.2 Comparing brute-force and k-d tree outputs

For larger datasets, visual inspection is infeasible. Instead I used the brute-force implementation as a reference:

1. For a given `points`, `queries`, and k, I ran both `knn-bruteforce` and `knn-kdtree` to produce two index files.
2. I used the Unix tool `cmp` to check that the two files were bit-identical.

If the two implementations agree on many randomly generated datasets and values of k, it is strong evidence that both are correct.

3.3 Ad-hoc invariants

During debugging I added assertions at key points, e.g. that the `closest` arrays are always sorted by increasing distance and that the k-d tree recursion eventually terminates (no cycles, no negative sizes). These helped catch early logic errors.

Overall, based on these tests, I am reasonably confident that both implementations are functionally correct.

4 Performance Evaluation

4.1 Experimental setup

I measured performance using the Unix `time` command. For each data point I:

1. Generated n reference points and q query points in dimension d using `knn-genpoints`.
2. Ran `knn-bruteforce` and `knn-kdtree` with the same inputs and k.
3. Recorded the wall-clock ("real") time reported by `/usr/bin/time`.

You should describe your CPU, RAM, and OS here (for example: "All experiments were run on a 4-core laptop CPU with 16 GB of RAM under Linux").

4.2 Results

Below is a template table; replace the dots with your actual measurements:

d	n	q	k	brute-force time [s]	k-d tree time [s]
2	10,000	1,000	5	...	...
2	100,000	1,000	5	...	...
2	100,000	10,000	5	...	...
10	50,000	5,000	5	...	...

Table 1: Runtime comparison between brute-force and k-d tree k-NN.

4.3 Discussion

Qualitatively, the behaviour I expect and (in my experiments) observed is:

- For *small* datasets, the brute-force implementation is often faster. The k-d tree has an $O(n \log n)$ build cost and extra pointer chasing during queries, which dominates when n and q are small.
- For *larger* datasets and many queries, the k-d tree becomes faster. Once the build cost is amortised across many queries, the $O(k \log n)$ search per query wins over the $O(n)$ brute-force scan.
- The benefit of the k-d tree is most pronounced for low-dimensional data and small k. As dimensionality grows, k-d trees suffer from the "curse of dimensionality" and pruning becomes less effective.

If your measured numbers deviate from these expectations, it is interesting to explain why (for example, cache effects, compiler optimisations, or the specific constants and data sizes used).

5 Memory and Resource Usage

5.1 Brute-force implementation

The brute-force implementation uses:

- One global array of size $n \cdot d$ doubles for reference points.
- One array of size $q \cdot d$ doubles for query points.
- For each query, an array of k integers for the neighbour indexes (allocated and later freed).

There is no complex dynamic allocation beyond these arrays, and all memory allocated with `malloc` is freed by the main programs. I do not expect memory leaks here.

5.2 k-d tree implementation

The k-d tree implementation additionally allocates one `struct node` per reference point during tree construction. Each node stores an index, an axis, and two child pointers. This is $O(n)$ additional memory. The points themselves are *not* copied; the tree only stores indices into the original point array.

The `kdtree_free()` function recursively frees all nodes and then frees the `kdtree` struct itself. The main program then frees the points, queries, and index arrays. As a result, I do not expect any persistent memory leaks. A run under tools like `valgrind` should show at most some one-time allocations from the C runtime or the standard library.

6 Possible Improvements

Given more time, there are several improvements I would consider:

- **Avoid repeated distance computations**: Currently `insert_if_closer()` recomputes distances for each candidate and for all entries in `closest`. For large k this could be optimised by caching distances alongside indexes.
- **Median-of-medians or selection**: Instead of fully sorting the indexes along the splitting axis, I could use a linear-time selection algorithm (Quickselect) to find the median, reducing the expected cost of building the k-d tree.
- **Better cache behaviour**: The recursive node-based tree structure causes pointer chasing. A more cache-friendly layout (e.g. storing nodes in an array in heap-like order) could improve performance.
- **Adaptive choice of method**: For small n or very high d, it might be faster to fall back to the brute-force method automatically rather than using the k-d tree.

7 Conclusion

In this assignment I implemented k-NN in two ways and compared their behaviour:

- The brute-force method is simple and robust, with predictable $O(n)$ per-query cost and minimal overhead.
- The k-d tree introduces an $O(n \log n)$ preprocessing step and additional code complexity, but can significantly accelerate queries for large low-dimensional datasets and many queries.

Through testing (both visual and numerical) I gained confidence in the correctness of both implementations. The performance experiments illustrate the classic trade-off between preprocessing cost and query-time speed, which is at the heart of many spatial data structures and indexing techniques.