HPPS 2024
Assignment three
K-NN

Marcus Frehr Friis-Hansen lns611
Noah Wenneberg Junge qxk266
Group 7

December 15, 2024

Introduction

The code developed is functionally correct and designed to test all core components, including `io.c`, `util.c`, `bruteforce.c`, and `kdtree.c`. A single test program, `test.c`, automates this process by verifying the correctness of reading, writing, distance calculations, k-nearest neighbors (k-NN) search, and k-d tree operations.
To build and test the program, type `make all` in the terminal while in the correct directory. Then execute the following commands sequentially, replacing `n`, `d` and `k` with suitable values:

```
./knn-genpoints n d > points
./knn-bruteforce points queries k indexes
./knn-kdtree points queries k indexes
```

Alternatively, the provided `test.c` file streamlines the testing process. It verifies the functionality of:

- Reading and writing points and indexes (`io.c`/`io.h`).
- Distance calculations and closest point selection (`util.c`/`util.h`).
- Brute-force k-NN search (`bruteforce.c`).
- k-d tree creation and k-NN search (`kdtree.c`).

To execute the test program, simply compile it using:

```
gcc -o test test.c io.c util.c bruteforce.c kdtree.c -lm
```

Then run the program as follows:

```
./test
```

The output will confirm the correctness of each component with detailed debug information and the results of both brute-force and k-d tree k-NN searches. The test file was developed using ChatGbt.

io.c/io.h

`read_points` and `read_indexes` are functionally correct. The functions reads data from a binary file, and returns a pointer to said file. They also write the size of the files to they `n_out` and `d_out`/`k_out`, as per the description in the **io** header file.

`write_points` and `write_indexes` are functionally correct. The functions writes given data to a points or index file respectively. They both return 1 for error or 0 for success

We don't think any of these functions leaks memory or other resources. All actions these functions perform are safeguarded by if statements, that return null or error code 0 if something unexpected is returned from either `fread` or `fwrite` within each function. Any case where the `malloc` function doesn't work as expected, the data is also freed again, as to not cause any memory leakage. We think our functions are quite solid, so we cant think of any issues with them. ChatGbt was used to verify syntax and for debugging purposes.

util.c/util.h

The `distance` function is functionally correct as it follows the mathematical formula given in the header file. Data leakage isn't really relevant and the function remains as good as its gonna get.

`insert_if_closer` is also functionally correct, it follows the description given in the header file by keeping an array of the k closest points to the query point, inserting the current point if there are any unused spaces in the array. And replacing the point with the farthest distance, if the current points distance is shorter. **Again, there isn't much we could improve.** ChatGbt was used to help debug and fix the verify the correctness of the **insert_if_closer** function.

kdtree.c/kdtree.h

The `kdtree\_create` function serves as the access point for constructing a k-d tree. It begins by allocating memory to hold the tree structure and initializes an *indexes array* to represent the *points array* during the construction process. This approach minimizes data movement during sorting, which is particularly efficient when the points have multiple dimensions. The recursive helper function `kdtree\_create\_node` is then used to build the tree structure. By sorting the indexes array along alternating axes at each depth, the implementation effectively splits the space into hyperplanes.

We find this design choice efficient and elegant, as it avoids copying the entire points array during sorting, thereby reducing both time complexity and memory usage. Memory allocation is carefully managed, and the `indexes` array is freed once the tree is constructed. The code is functionally correct and has been verified to construct the tree as expected.

The `kdtree\_knn` function allocates memory for the `closest` array, which will hold the indexes of the k nearest neighbors. Initially, all entries in the array are set to `-1`, signifying that no points have been found yet. It uses the recursive helper function `kdtree\_knn\_node` to perform the actual search for the k nearest neighbors by traversing the tree efficiently.

The function uses the axis-aligned splitting property of the k-d tree to prune branches that cannot contain closer points. The distance to the current farthest neighbor in the `closest` array determines whether or not to explore both branches of a split node. The implementation correctly balances pruning and recursion, leading to improved search efficiency.

While the k-d tree implementation is functionally correct and efficient, there are a few potential shortcomings that could be addressed:

The current implementation may perform sub-optimally for extremely high-dimensional datasets due to the *curse of dimensionality*. As the number of dimensions increases, pruning becomes less effective, and the tree traversal approaches brute-force behavior. Testing with large, high-dimensional datasets could highlight these limitations and suggest further optimizations.

The implementation assumes valid input for points, dimensions, and queries. Adding additional error handling for edge cases, such as empty input files or invalid dimensions, would make the program more robust and prevent unexpected crashes or undefined behavior.
ChatGbt was used to get at better understanding of the helper functions, as well as debugging the code and verifying its correctness.