High Performance Programming and Systems

Assignment Two

k-NN

Set: *9th of December 2021*
Due: *17th of December 2021 @ 23:59 CEST*

Synopsis:
Implement the k-nearest-neighbour (k-NN) algorithm with both a brute-force and a k-d tree approach.

1 Introduction

This is the second of five assignments in the *High Performance Parallel Systems* course. For this assignment, you will be implementing the k-nearest-neighbour (k-NN) algorithm. In a KNN problem, we are provided a collection of *reference points* in some multidimensional space. Given a distinct *query point*, the nearest-neighbour (NN) problem is then to determine the reference point that is closest to the query point. The k-NN problem is to determine, for some k, the k nearest reference points to the query point. Solving k-NN problems can be used to perform classification ("which previous observations is this new observation most similar to?") as well as regression ("what is the average of similar observations"). Despite its simplicity, it is a widely used method, often used on large data sets, and so it is important that implementations run fast. This means they are often implemented in low-level code and with clever algorithms. For this assignment, you will do both.

1.1 Brute-force k-NN

The most straightforward way to implement k-NN is to loop over all reference points while maintaining a list of the k points seen so far that are closest to the query point. In pseudocode, we could express this as follows:

```
Procedure knn(k, points, query)
    closest ← [];
    foreach point in points do
        if closest has fewer than k elements then
            insert point in closest
        else if point is closer to query than any point in closest then
            remove most distant element in closest;
            add point to closest
    return closest
```

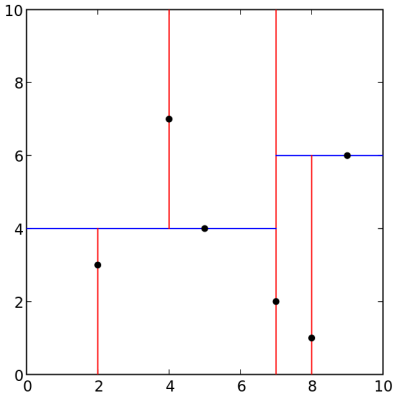
There are ways to implement this more efficiently via low-level optimisation. For example:

1. We can save the distances for the elements in *closest* so we do not have to recompute them often.
2. Instead of storing actual points in *closest*, we can store indexes to the reference points.
3. By maintaining *closest* as sorted by increasing distance, we make it cheaper to insert new points.

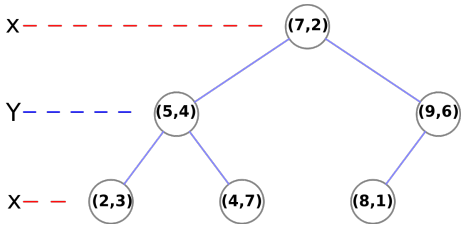
But still, this will require inspecting all n reference points, so the asymptotic complexity is $O(n)$ for every query.

1.2 k-NN with a k-d tree

In practice we are usually not looking up the k nearest neighbours for just a single query point, but for *many* queries in the same space of reference points. This means that we can afford to spend some time precomputing an auxiliary data structure that will by itself be expensive, but will be amortised by speeding up each subsequent query. Specifically, we can use a k-d tree, which is a *spatial data structure* that subdivides the space. Note that, confusingly, the k in a k-d tree refers to the number of *dimensions*, not the k in k-NN. Whenever k is used by itself in the following, it refers to the k in k-NN. We will use a two-dimensional space for ease of visualisation in the following, but your solution must work for any dimensionality.



(a) k-d tree decomposition for the point set. From `https://en.wikipedia.org/wiki/File:Kdtree_2d.svg`.



(b) The k-d tree structure corresponding to the decomposition. From `https://en.wikipedia.org/wiki/File:Tree_0001.svg`

Figure 1: Visualisation of k-d tree decomposition of the point set $(2,3), (5,4), (9,6), (4,7), (8,1), (7,2)$.

In a k-d tree, every non-leaf node splits the space along an axis-aligned cutting hyperplane. For the two-dimensional case, this is either a horizontal or a vertical line. See fig. 1 for an example.

The advantage of using a k-d tree for finding nearest neighbours is that we can efficiently exclude large swathes of the space. Construction of a perfectly balanced k-d tree for n points takes $O(n \log n)$ steps, while determining the k nearest neighbours a query point takes $O(k \log n)$. If we have many query points, then using a k-d tree is faster than the brute-force approach.

Construction

A k-d tree is constructed with a recursive algorithm. It is defined as follows in pseudocode, where d is the number of dimensions in the space:

```
Procedure kdtree(points, depth)
    axis ← depth mod d;
    select median by axis from points;
    node ← new node;
    node.point ← median;
    node.axis ← axis;
    node.left ← kdtree (points before median, depth+1);
    node.right ← kdtree (points after median, depth+1);
    return node
```

Querying

Finding the k nearest neighbours to a point query is done by a recursive tree walk, where we maintain an array closest of the k nearest neighbours encountered *so far*. When we visit a node, we insert the corresponding point into **closest** if it is closer than any any of the previously picked points, just as in the brute-force implementation. However, the trick is that we do not necessarily visit *both* children of the node. While we do have to visit the child corresponding to the space that contains the query point, we only have to visit the other child if the (hyper-)sphere centered at query and containing all points in closets intersects the cutting (hyper-)plane described by the node. Because the hyperplane is axis-aligned, this can be done trivially, as shown in the following pseudocode:

Procedure treeknn(k, closest, query, node, depth)

```
if node is NULL then
    return
else if closest has fewer than k elements then
    insert node.point in closest
else if node.point is closer to query than any point in closest then
    remove most distant element in closest;
    add node.point to closest

diff ← node.point[node.axis] - query [node.axis];
radius ← the distance to the most distant element in closest

if diff ≥ 0 ∨ radius > |diff| then
    treeknn (k, closest, query, node.left)
if diff ≤ 0 ∨ radius > |diff| then
    treeknn (k, closest, query, node.right)
```

You do not have to understand why this works in higher-dimensional cases—simply confirm for yourself why this makes sense for a two-dimensional space (fig. 1), and take it on faith that it also works in geometries that are harder to visualise on paper.

More resources

It can be difficult to understand the behaviour of spatial data structures solely from a textual description. The following YouTube video explains visually how the k-d-tree is constructed, and how lookups are performed: `https://www.youtube.com/watch?v=ivdmGcZo6U8`. Note that the k-d-trees you will construct differ from the video in the minor way that internal nodes, *not* just leaves, also contain points. Generally, k-d trees are widespread data structures, so you should be able to easily find more expository material online if the pseudocode above is not sufficient to build your intuition.

2 Code handout

This section describes the files in the code handout (the `src/` directory) and the modifications you are expected to make, in the order you are expected to work with them. First of all: **don't panic**. There is a lot of code here,

but it is meant to be understood and implemented incrementally. For every incomplete implementation file, you should finish and test it before moving on. In the program usage examples that follow, the data file names are not important, and can be anything you want.

`Makefile`: How to build the source files. Contains a generic rule that compiles any `.c` file to a corresponding `.o` file, but you will need to add new rules if you want to add new executable programs.

`io.h`/`io.c` **(incomplete):** Reading and writing data files that contain points and indexes of point. The exercise tasks involved writing exactly this code, so you can use your own solution from there, or use the one we gave you.

`knn-genpoints.c`: Compiles to `knn-genpoints`. If run as

$$\texttt{\$ ./knn-genpoints } n \ d \texttt{ > points}$$

the program will generate n random d-dimensional points with coordinates in the range $(0, 1)$ and write them to the file `points`. We will use this tool to generate both reference points and query points.

`knn-svg.c`: Compiles to `knn-svg`. You do not need to understand the implementation of this program, but it can be useful for debugging. If run as

```
$ ./knn-svg points > points.svg
```

it will generate an SVG file showing the locations of reference points, which can then be viewed in an image editor or browser. If run as

```
$ ./knn-svg points queries indexes > points.svg
```

it will also show the query points and a circle encompassing their k nearest neighbours as given by `indexes` (see below for how to generate this file).

Once you have begun implementing `kdtree.c`, you can set the variable `draw_kdtree=1` in the source code to also draw the k-d tree structure. This can be very useful for debugging.

`util.h/util.c` **(incomplete):** The header file declares two functions that you will need to implement.

`bruteforce.h/bruteforce.c` **(incomplete):** brute-force k-NN. Your task is to implement the behaviour documented in the header file. You will need to make use of the definitions in `util.h`.

`knn-bruteforce.c`**:** compiles to `knn-bruteforce`. If run as

`$ ./knn-bruteforce points queries` k `indexes`

the program will read reference points from `points`, query points from `queries`, and write the indexes of the k nearest reference point for each query point to `indexes`.

`sort.hsort.c`**:** a generic implementation of the quicksort algorithm. You will need this for the k-d tree construction, as computing the median is best done by sorting and then picking out the middle element.

`kdtree.h/kdtree.c` **(incomplete):** construction of a k-d tree and performing k-NN lookups in the tree. The two `struct` definitions are given, and do not have to be modified (but you are allowed to modify them if you believe you can do better than us). Further, there is a function `kdtree_svg()` that visualises the tree structure when called from `knn-svg.c`, which may be useful for debugging (or just pretty pictures).

`knn-kdtree.c`**:** compiles to `knn-kdtree`. Behaves like `knn-bruteforce`, but uses the k-NN implementation from `kdtree.c`, so it will run faster for large problem sizes.

3 Testing and benchmarking

Consider and implement some form of correctness testing of your work. Here are some ideas:

- For small datasets, visual inspection of the output of `knn-svg` can be used.

- For larger datasets, visual inspection is infeasible. Instead you can write a separate program that reads in `points`, `queries`, and `indexes` files, and *verifies* the result by checking that for each query point, there is no reference point that is closer than any of the points from to be the nearest.
- Once you are confident that `knn-bruteforce` computes the correct results, you can test `knn-kdtree` by simply checking whether the two programs produce identical results (the Unix `cmp` tool can be used to check whether two files are identical).

Since the entire purpose of using a k-d tree is to be faster than the brute-force approach, you should also use the Unix `time` tool to measure the run-time of your programs on various input sizes. Demonstrate at least one workload where brute-force is fastest, and one where the k-d tree is fastest.

4 Your Report

You are expected to comment on the *interesting* details of your implementation. You are *not* expected to give a line-by-line walkthrough of your code. Most importantly, you are expected to reflect on the *quality* of your code:

- Do you think it is functionally correct? Why or why not?
- Do you think it leaks memory or other resources? Why or why not?
- Is there some improvement you'd have liked to make, but didn't have the time?

It is more important to be aware of the strengths or shortcomings of your solution, than it is to have a complete solution.

5 Deliverables for This Assignment

You should submit the following items:

- A single PDF file, A4 size, no more than 5 pages, describing each item from report section above
- A single zip/tar.gz file with all code relevant to the implementation

6 Handing In Your Assignment

You will be handing this assignment in using Absalon. Try not to hand in your files at the very last-minute, in case the rest of the students stage a DDoS attack on Absalon at the exact moment you are trying to submit. **Do not email us your assignments unless we expressly ask you to do so.**.

7 Assessment

You will get written qualitative feedback, and points from zero to 4. There are no resubmissions, so please hand in what you managed to develop, even if you have not solved the assignment completely.