HPPS - Assignment 2

pwn274 and npd457

December 2021
Segmentation fault (core dumped)

1 Introduction

In this assignments we were tasked to build two versions of the K-nearest neighbour algorithm. One crude brute-forcing version that finds all the distances from a given query point to all the points in the points array, and one less crude version that utilizes tree structures, in order to limit the number of points that the algorithm has to consider. The implementations we came up with are succinctly outlined below.

2 Implementation

2.1 Brute force method

2.1.1 The *util* library

First and foremost we had to make the functions in the utility library (util.h, util.c). We implemented the distance function quite easily by looping over the dimensions of the points. The `insert_if_closer` was a bit more intricate. The goal of the code is to check if the candidate has a smaller distance to the query point than a point already in the `closest` array. If this is the case, then the index of the point with the largest distance to the query point in the array is to be replaced with the index of the new candidate point. We manage this by defining the variables `highest_in_array` and `highest_index_array` and continually update the values as we loop through `closest`. The code snippet below illustrates this:

```
double dist = distance(d, query, &points[d*closest[i]]);
if(dist >= highest_in_array && candidate_dist < dist){
  highest_in_array = dist;
  highest_index_in_array = i;
}
if (highest_index_in_array != -1){
    closest[highest_index_in_array] = candidate;
    return 1;
}
```

2.1.2 The actual k-NN-algorithm

The actual brute force k-NN algorithm was easily implemented after the utility library had been made. We simply allocate space for the index array through `malloc(k * sizeof(int))`, initialize all the the entrances in the array as -1 and loop the points through the `insert_if_closer` in the following manner:

```
for(int i = 0; i < n; i++){
insert_if_closer(k,d,&points[d*i],closest,query,i);
}
```

2.2 k-d trees

The work on implementation of the k-d tree was twofold: Firstly the trees had to be created and then the neighbours had to be found. The former required the creation of a recursive function that created a leaf and assigned it two subleafs, the latter would (also recursive) run through the created nodes checking distances.

2.2.1 tree-creation

As the node-struct was already created for us, the first line we wrote was memory allocation. After this was taken care of, the actual implementation could begin:

```
struct node *node = malloc(sizeof(struct node));

sort_indexes(d,axis,points,indexes,n);

int half = (int) floor(n/2);
int median = indexes[half];
int other_half = n - half - 1 ;

node->left = kdtree_create_node(d, points, depth+1, half, indexes);
node->right = kdtree_create_node(d, points, depth+1, other_half, &indexes[
    half+1]);

return node;
```

We sort the list for very node, but as each leaf simply holds an index for the "points"-array, no problems arose. As the time needed to sort an array is not negligible, we presume simply sorting for every dimension once and then looking up the ordering would improve the time needed per leaf immensely.

To start with, we spend quite a lot of time on trying to sort indices ourselves. At one point, we spoke to a member of another group and they pointed us towards[1] a snippet found on the course GitHub. After this was implemented, the toughest part consisted of figuring out how to relay the partitioned array to the child nodes. This was solved by counting indices on our fingers so the parameters all lined up. //jeg ved ikke hvad jeg skal skrive, mand

2.2.2 Neighbour-finding

The k-NN-finding on the nodes were implemented quite directly from the pseudo-code found int the assignment text. The only thing worth highlighting are hundreds of bugs

[1] Memory-address joke very much intended

caused by forgetting to multiply by the dimensionality when indexing and a reuse of the `insert_if_closer`-function written for the brute-force k-NN. But we expect most other groups to also have figured this out.

3 Testing and marking benches

3.1 Brute-force algorithm

During most of the debugging we used the `knn-svg` tool on small datasets to find out whether or not the algorithm was doing the correct thing. An example of these images can be observed in figure 1 (ignore the tree lines). This however only works for small datasets in two dimensions. In order to ascertain if the algorithm was doing alright on larger datasets we wrote a script that takes `points`, `queries` and the `indexes` from the brute-force algorithm and tests if there exists points in `points` that are closer to a given query than the one presented in `indexes`. Below is a function code-snippet that checks this for a single point and a single query. The main script simply loops over this function:

```
int check_if_lower(int k, int d, const double *points, int *q_indexes, const
    double *query,int candidate)  {
    double *point_candidate = &points[d * candidate];
    double candidate_dist = distance(d, query, point_candidate);
    if(candidate_dist == 0) {
        return 1;
    }
    for(int i=0; i<k; i++) {
        if(candidate == q_indexes[i]) {
            return 1;
        } }
    for(int i=0; i<k; i++) {
        double index_dist = distance(d, query, &points[d * q_indexes[i]]);
        if(candidate_dist < index_dist ) {
            return 0;
        } }
    return 1;}
```

We ran the brute-force algorithm for varying values of k, n and d and checked them the outputs with this program. All of the results were positive, and we do therefore very much believe that the brute force algorithm works as it should.

3.2 k-d tree algorithm

As with the brute force algorithm most of the initial debugging for the k-d tree algorithm was done using the `knn-svg` tool. For larger datasets we compare the output from the algorithm with that of the brute-force algorithm, as we are pretty sure that that works as it should. We do this comparison using the Unix-command `cmp`.

The k-d tree algorithm seems to *mostly* work. We say this as we have noticed some faults in it. One of these faults can be found inspecting 1. In this image we observe two *completely* image-spanning vertical lines. As far as we understand there is only supposed to be one. Having run through the code multiple times, we cannot seem to find the culprit. We have a hunch that the way we partition the indices when creating sub-leafs is to blame, but we have unfortunately not been able to solve the problem.

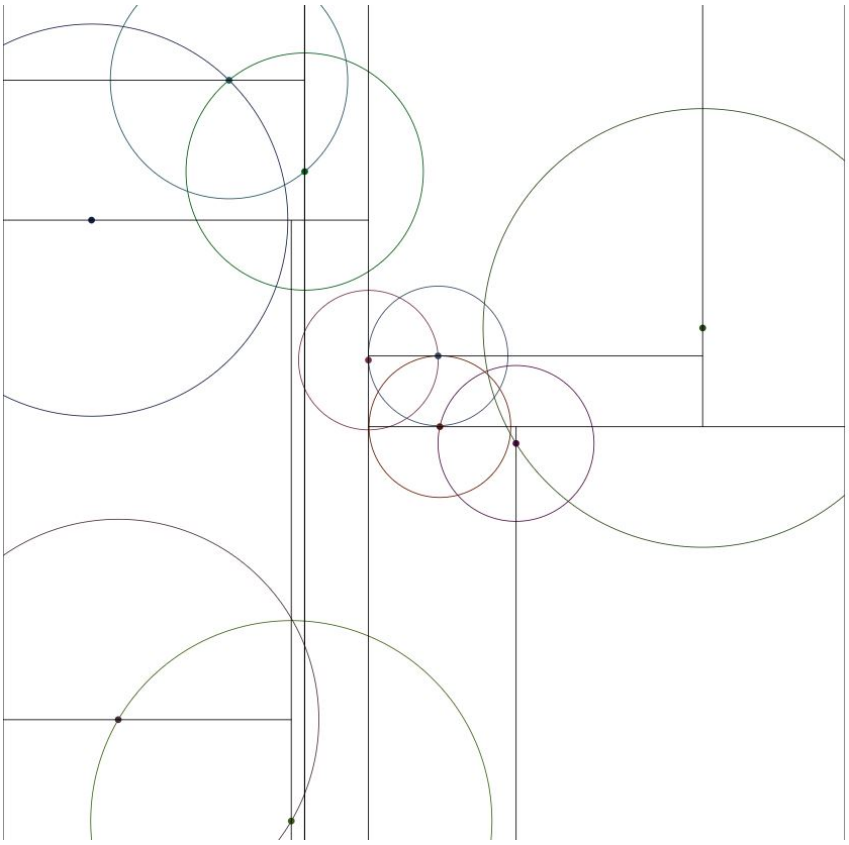

Figure 1: A visual representation of the single nearest neighbour and tree structure for 20 points.

Another problem we have found with our k-d tree algorithm is that it never agrees with our brute force algorithm when we compare the binary index files, but they do agree when we compare small data sets visually via the svg-plots, and when we run the program that checks if there exists points in `points` that are closer to a given query than the one presented in `indexes` (described above) it also gives us no errors. We believe that the discrepancy between the two functions are either due to the way that the programs read and write data, or inconsistent ordering of the indexes, but we have not been able to find it out exactly. CMP always seem to find a difference 9 bytes into the files when working with any number of points.

3.3 Timing

When finding 5 neighbours for 10000 points in 10 dimensions, the following results were found:

	brute force	k-d tree
real	0m33.884s	1m32.615s
user	0m33.656s	1m31.094s
sys	0m0.000s	0m0.063s

We are pretty surprised by this result, as the brute force algorithm seemed faster for all dimensions and number of points we tested. The time included both the creation of the trees and the neighbor finding. Given an pre-created tree, we assume that the tree-based method would result in far better times.

3.4 valgrind

As most of the memory freeing came pre-written in the assignment source code, the memory management mostly consisted of writing the freeing functions and correctly allocating when needed.

To make sure we had no memory leaks (see figure 2) or threading issues, we ran a verbose `valgrind` on both knn implementations as well as the check function using large datasets. The analyses all came up with a version of the following:

```
HEAP SUMMARY:
in use at exit: 0 bytes in 0 blocks
total heap usage: 297,245 allocs, 297,245 frees, 4,006,736 bytes allocated

All heap blocks were freed -- no leaks are possible
```

Which we liked.

4 Conclusion

We are pretty confident in our memory management, and all visual inspections of the code seem correct, but there is still some bugs that need ironing out.

Our implementation had some obvious improvement to be made:
Firstly, when creating the trees, every node sorts the indices according to their own axis. Given more time we would no doubt have tried implemented some sort of pre-sorting.
Secondly, timing the k-d tree algorithm separately for tree creation and neighbor finding could give a more accurate representation of the quality of the algorithm.
Thirdly, fixing the mysterious double-line-error would is of course top-priority, but we unfortunately did not have the time.
Lastly, as we know there is a difference between the outputs, writing a code snippet that could find the exact differences between the two methods would be very good for further testing.

5 Appendix

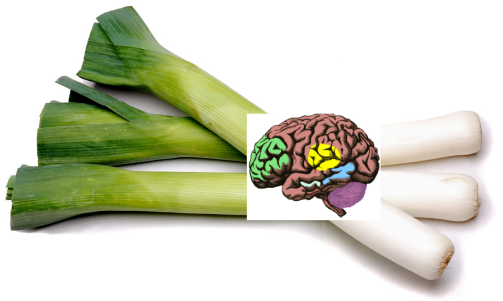

Figure 2: a memory leek, i think

I want to personally thank you, as this assignment has furthered my appreciation for Python and all other comfortable things in life.