# HPPS - Assignment 3

pwn274 and npd457

December 2021
*The furthest place from East Greenwich is West Greenwich*

## 1 Introduction

In this assignment we were tasked with writing four programs that performed queries on a real world dataset. Three of the algorithms essentially do the same thing: Given an index - the programs print the corresponding name of the geographical location, if it exists. The programs `id_query_indexed.c` and `id_query_binsort.c` do a lot better than `id_qery_naive.c` in large part due to better locality.

## 2 Querying by id

### 2.1 Naive approach

The naive approach to querying was a linear search through all the records to find the desired ID. This approach has very poor spatial locality. The poor spatial locality comes from the fact to check each ID the program has to 'open a record', which essentially means that the stride between ID's is pretty huge. The code below outlines this process.

```
const struct record* lookup_naive(struct naive_data *data, int64_t needle) {
  for (int i = 0; i<data->n; i++){
    struct record *rec = &(data->rs[i]); //A record is opened -> big stride
    if(rec->osm_id == needle){
      return rec;
    }
  }
```

### 2.2 Querying by indices

By querying by indices we get a lot better spatial locality, as the strides between each index becomes a lot smaller - it becomes '2' as the `struct_index_record` contains only the index and a pointer to the record.

### 2.3 Binary search of indices

Another benefit of extracting the ID's were, that the records could now easily be sorted according to these. Once the records are sorted, a more sophisticated search through them seems obvious - the binary search, which was implemented in the following way:

```
while (1) {
    struct index_record *rec = &(data->irs[index]);
    int64_t diff = rec->osm_id - needle;

    if(diff == 0){
      return rec->record;
    } else if (diff < 0){
        min = index;
    } else {
        max = index;
    }

    index = (max + min)/2;

    if(abs(max-min)<=1){
        return NULL;
    }
}
```

As for as we can tell the implementation of the binary search does not affect the locality, at least not in a postive way. It might even have worse spatial locality than a binary search as it 'hops around' within the indices. The implementation of it does however limit the number of lookups required for the search substantially.

# 3 Querying by coordinates

The coordinate querying was surprisingly simple to implement. A distance function was written and a simple loop over all data points gave us the ability to find the shortest distance between any two points on the globe (or plane, more precisely).

The assignment only asked for a naive approach. If we wanted to improve this function, an obvious place to start would have been to implement the same sub-structure as was done in the ID querying case. Extracting the longitude and latitude to a smaller structure would improve the spatial locality like the `indexed`-function.

# 4 Memory management

Having run valgrind, we are not completely happy about our result. There are seemingly no errors, but there are some bytes in the "still reachable" leak section. We assume this is due to some unfreed memory, but we cannot seem to find the source.

Addendum: It seems we are horrible at reading C-code and had not understood that the program needed to be stopped by ctrl+d. Doing this improves the error message but does not fix the problem completely. Given more time (or a better understanding of the problem a couple of hours ago) the lost-memory-problem could definitely be corrected!

# 5 Confidence

Generally, the id-based programs finds the same outputs when given the same queries which tells us we have an even number of mistakes in our code: Either they are both correct or

both wrong.

We also checked some of the id's to what we could visually see in the text file, where we saw that we found the correct id's.

`coord_query_naive.c` was tested by assuring that when given several 'sets' of coordinates all closest to the same geographical location, they would return that location.

Having automated tests would have improved this report. Especially implementing the `random_ids`-function that was handed to us. Unfortunately we didn't have the time for this.

# 6 Benchmarking

## 6.1 ID Querying

To be thorough with the benchmarking we looked at the performance, both for queries that were found and queries that were not. Below are some examples hereof, along with our general observations:

**Naive search:**

```
Reading records: 232449ms
Building index: 1ms

0: not found
Query time: 80274152us

99999999: not found
Query time: 39898445us

3219806: Pierre-Perthuis 3.789289 47.432325
Query time: 21391us

9987163: Via del Sur -96.875736 32.968626
Query time: 22320422us
```

**Indexed**

```
Reading records: 160130ms
Building index: 52536ms

0: not found
Query time: 538003us

9999999: not found
Query time: 166701us

3219806: Pierre-Perthuis 3.789289 47.432325
Query time: 100us

9987163: Via del Sur -96.875736 32.968626
Query time: 53814us
```

**Binary search**

```
Reading records: 107073ms
Building index: 49132ms

0: not found
Query time: 4us

9999999: not found
Query time: 3us

3219806: Pierre-Perthuis 3.789289 47.432325
Query time: 21us

9987163: Via del Sur -96.875736 32.968626
Query time: 8us
```

**General Comments**:

By comparing the timings above it is easy to see that generally the performance improves as we go down the list. The performance improvement from the naive method to the indexed method is especially interesting as this improvement is most likely due to better locality in the code.

## 6.2 Coordinate querying

**Naive coordinate querying**

```
(51.000000,0.000000): El Endanane (47.172501,3.619100)
Query time: 9206483us
51 0.1
(51.000000,0.100000): Mareeg (47.302336,3.752664)
Query time: 511528us
52 1
(52.000000,1.000000): Ceel Cabdi (47.951500,4.480200)
Query time: 488785us
51.5 0.001
(51.500000,0.001000): African Banks (53.386854,-4.881029)
Query time: 524597us
01 52
(1.000000,52.000000): Hadleigh Road (0.998305,51.991572)
Query time: 502767us
0.001 51.4
(0.001000,51.400000): Pellings Close (0.000499,51.399688)
Query time: 730658us
```

The coordinate query always finds something to return and it generally takes an equal amount of time to fetch any entry. As mentioned in the appropriate section, this algorithm could be better and the times therefore shorter, but unfortunately we did not have the time for a better implementation.

# 7 Conclusion

In conclusion we have made the four functions. The functions with best locality were definitely `id_query_indexed.c` and `id_query_binsort.c`. All of the functions seem to

work just fine without huge memory leaks or corruption. If we had had more time, we would have liked to implement better locality for the the coordinate query function.