A1: Dynamic Memory and Cache Optimisations

Computer Systems 2022
Department of Computer Science
University of Copenhagen

Troels Henriksen

Due: Sunday, 9th of October, 16:00
Version 1 (September 26, 2022)

This is the second assignment in the course Computer Systems 2021 at DIKU. We encourage pair programming, so please form groups of 2 or 3 students. Groups cannot be larger than 3, and we strongly recommend that you do not work alone.

For this assignment you will receive a mark out of 4 points; You must attain at least half of the possible points to be admitted to the exam. For details, see the *Course description* in the course page. This assignment belongs to the OS category together with A2. Resubmission is not possible.

---

1 Introduction

> *It is not the task of the University to offer what society asks for, but to give what society needs.*
>
> — Edsger Dijkstra, EWD1305 (2000)

This assignment is meant to test three competences:

1. Writing modular C programs comprising multiple files.
2. Handling dynamic memory and pointers, particularly avoiding memory leaks and invalid memory accesses.
3. Analysing cache behaviour of code and performing cache optimisations.

Keep this in mind when working on the code, and particularly when choosing what to focus on for your report. Make sure to read the entire text before starting. In particular, read section 5 to understand the code handout.

2 Purpose

You will be writing a series of programs to perform *queries* on a real-world dataset. The dataset in question is OpenStreetMap's dataset of "place names" for the whole planet. The full dataset is available at `https://osmnames.org/download/`, and contains 21 million records. Each record contains many fields, representing information about a place somewhere on Earth. For our purposes, the important fields are:

`osm_id`**:** a distinct number ("ID") identifying a place (e.g. "3546485").

`lon`, `lat`**:** the longtitude and latitude of the place (e.g. "12.5604108", "55.7026665").

`name`**:** the canonical human-readable name of the place (e.g. "Universitetsparken").

After decompression with `gunzip`, the dataset is a file `planet-latest_geonames.tsv` that contains an initial line labeling the record fields, and then one line per record, with tab-seperated fields (`'\t'`). The handout contains functions for reading the dataset into memory in the `record.h` and `record.d` files.

The full dataset is large and somewhat unwieldy. For testing, it can be useful to extract a smaller subset. For example, we can extract the first 20000 records by running the following Unix command[1]:

```
$ head -n 20000 planet-latest_geonames.tsv > 20000records.tsv
```

You will be writing various programs for performing two kinds of queries against the dataset:

1. Given an integer, find the place with the corresponding `osm_id`, if any.
2. Given longtitude and latitude, find the closest place.

Each program will be *interactive*: after loading the dataset (given by a filename), the program will run a loop that continuously reads and processes queries from the user. This is intended to simulate a server application, as you have not yet been taught network programming. Such a design is common in practice, and has the interesting property that it can be advantageus to perform costly *preprocessing* to produce an *index* of the data in order to respond more efficiently to queries. This is because the data will only be loaded once at startup, from which we might service millions of queries.

The two kinds of queries are covered by two subtasks. Apart from your code, you are also expected to write a short report (no more than 5 pages). Each subtask contains details on what your report *must* contain, and how much of the final score it counts for. Beyond that, use your own judgment about what you think your TA will enjoy reading.

For some tasks, the code handout will contain files you can modify. For others, you will need to create new files from scratch. Feel free to base them on the handed out ones. Remember to also update the Makefile!

[1] Technically this extracts only 19999 records, as the first line is metadata.

3 Querying by ID

In the following subtasks you will be writing programs that given an ID, prints the name of the place with that ID, if any.

3.1 `id_query_naive.c`: Naive brute-force querying (10%)

The first part of this subtask is finishing the program `id_query_naive.c`, which makes use of the library code in `id_query.h`/`id_query.c`. Read this code *carefully* (particularly the header file) to understand how to use it. The `id_query_naive.c` program should operate by performing a linear search through all records for the desired ID. Usage example:

```
$ ./id_query_naive planet-latest_geonames.tsv
Reading records: 32070ms
Building index: 0ms
45
45: Douglas Road -1.820557 52.554324
Query time: 72697us
1337
1337: not found
Query time: 118261us
```

The next two subtasks involve creating faster versions of this program. Do *not* optimise `id_query_naive.c` itself—keep it around as a "known correct" implementation so you can see whether your faster versions are still correct, and so you can easily measure the impact of your changes.

3.2 `id_query_indexed.c`: Querying an index (10%)

Write a program `id_query_indexed.c` that, instead of searching an array of `struct record` values, searches an array of the following:

```
struct index_record {
  int64_t osm_id;
  const struct record *record;
};
```

E.g. we might define the following structs and functions:

```
struct indexed_data {
  struct index_record *irs;
  int n;
};

struct indexed_data* mk_indexed(struct record* rs, int n);
void free_indexed(struct indexed_data* data);
const struct record* lookup_indexed(struct indexed_data *data,
                                    int64_t needle);
```

3.3 `id_query_binsort.c`: Querying a sorted index (10%)

Sort the array of records by their `osm_id` (use `qsort()`), and use binary search when looking up records.

3.4 Bonus? (up to ∞%)

Feel free to write more programs if you wish. Maybe you can think of optimisations I didn't. One fun idea is to use an Eytzinger layout for the sorted records, as discussed at `https://algorithmica.org/en/eytzinger`.

3.5 The report (20%)

For this task, your report should cover at least the following:

- Evaluate the temporal and spatial locality of the three programs you have implemented.
- Do your programs corrupt memory? Do they leak memory? How do you know?
- How confident are you that your programs are correct?
- How confident are you that your optimised programs are fast? How did you pick the benchmarking data?

4 Querying by coordinate

In the following subtasks you will be writing programs that given a longtitude/latitude coordinate, prints the name of the closest place in the dataset.

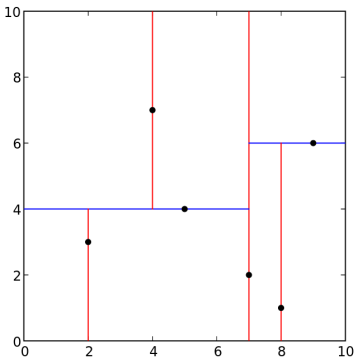
For the purpose of this assignment, we consider the distance between two coordinates (x_1, y_1) and (x_2, y_2) to be given by the Euclidean distance

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

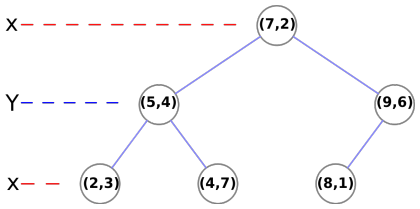
although this is actually nonsense for longtitude/latitude coordinates since they occur on a sphere. You'll have plenty of other details to worry about, so we'll pretend the Earth is a rectangle for now.

4.1 `coord_query_naive.c`: Naive brute-force querying (10%)

The first part of this subtask is finishing the program `coord_query_naive.c`, which makes use of the library code in `coord_query.h/coord_query.c`. Read this code *carefully* (particularly the header file) to understand how to use it. The `coord_query_naive.c` program should operate by performing a linear search through all records and pick the record whose location is closest to the query coordinates. Usage example:



(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 $(2,3), (5,4), (9,6), (4,7), (8,1), (7,2)$.

```
$ ./coord_query_naive planet-latest_geonames.tsv
Reading records: 30823ms
Building index: 0ms
10 10
(10.000000,10.000000): Mun-Munsal (9.962352,10.011060)
Query time: 178534us
45 45
(45.000000,45.000000): Levokumsky District (45.085875,45.022457)
Query time: 172391us
```

4.2 `coord_query_kdtree.c`: Using a spatial data structure (20%)

In this subtask you must write a program `coord_query_kdtee.c` that is used the same way as `coord_query_naive.c`, but internally uses a *k*-d-tree for finding the closest place.

A *k*-d tree is a *spatial data structure* that subdivides some *k*-dimensional space. For this assignment, $k = 2$—which is also a lot easier to visualise. 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 searching a *k*-d tree 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. In this subtask you must implement construction of a *k*-d tree (for $k = 2$), as well as lookups into the tree.

4.2.1 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
```

We let axis $= 0$ denote longtitude and axis $= 1$ denote latitude.

4.2.2 Querying

Finding the point closest to a given point query is done by a recursive tree walk, where we maintain a variable closest of the closest point encountered *so far*. When we visit a node, we replace closest if the node is closer to query, 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 sphere centered at query and whose radius is the distance from **closest** to **query** 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 lookup(closest, query, node)

```
if node is NULL then
    return
else if node.point is closer to query than closest then
    replace closest with node.point;

diff ← node.point[node.axis] - query [node.axis];
radius ← the distance between query and closest

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

Here we assume that we can index a point with the axis (0 or 1) to obtain the longtitude or latitude respectively. In code, this likely requires a branch.

4.2.3 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.

4.3 The report (20%)

For this task, the report should cover at least the following:

- Do your programs use memory correctly? Do they leak memory? How do you know?
- How confident are you that your programs are correct?
- How much faster is the solution from section 4.2 than the one from section 4.1? Can you find an input that the brute-force solution handles faster? Why is that?

5 Handout/Submission

Alongside this PDF file there is zip archive containing a framework for the assignment.

```
$ unzip src.zip
```

You will now find a `src` directory containing the following:

`.gitignore`
: A suitable `.gitignore` file, should you choose to use Git.

`record.h`, `record.c`
: Library code for reading the OpenStreetMap dataset. You *must* read and understand the header file, but reading the implementation file is optional.

`id_query_naive.c`
: A skeleton file for the first subtask.

`Makefile`
: The file that configures your `make` program. As a special treat,

```
$ make planet-latest-geonames.tsv
```

will download the dataset for you.

`random_ids.c`
: A program for generating random valid IDs corresponding to some data file. Compile with `make random_ids` and run as e.g.:

```
./random_ids planet-latest_geonames.tsv  | head -n 1000000 > 1M_ids
```

This is useful for generating test data for the first task.

`id_query.h`, `id_query.c`

A reusable implementation of the loop that reads ID queries from the user and looks them up in the index.

`coord_query.h`, `coord_query.c`

A reusable implementation of the loop that reads coordinate queries from the user and looks them up in the index.

You should hand in a ZIP archive containing a `src` directory containing all *relevant* files (no ZIP bomb, no compiled objects, no auxiliary editor files, etc.).

To `make` this a breeze, we have configured the `Makefile` such that you can simply execute the following shell command to get a `../src.zip`:

```
$ make ../src.zip
```

Alongside a `src.zip` submit a `report.pdf`, and a `group.txt`. `group.txt` must list the KU-ids of your group members, one per line, and do so using *only* characters from the following set written using *standard alphabetical encoding*:

$$\{\texttt{0x0A}\} \cup \{\texttt{0x30}, \texttt{0x31}, \ldots, \texttt{0x39}\} \cup \{\texttt{0x61}, \texttt{0x62}, \ldots, \texttt{0x7A}\}$$

Please make sure that the file is UTF-8 encoded with UNIX line endings.