Assignment 1 — Dynamic Memory and Cache Optimisations

Schmidt, Victor Alexander, `rqc908`
Ibsen, Helga Rykov, `mcv462`
Barchager, Thomas Haulik, `jxg170`

Sunday 16:00, October 9th

1 Querying by Coordinates

1.1 Do your programs use memory correctly? Do they leak memory? How do you know?

Throughout our code, we did indeed allocated memory to use in our code. Mostly to store data in defined structures to be used in our search algorithms. At the same time, once we had allocated memory, we then also made sure to free it, when we were done using it.

For testing, we did exactly as in Querying by ID, where we ran two memory-tests. Firstly, we added additional sanitize flags to our `Makefile`. Then we compiled and ran our programs to see if any memory leaks where detected, which there were none.

Secondly, we ran the `pstree` command to see all running processes. As in Querying by ID, we ran the program three times. First, before running the programs, second, while running each program individually and, third, after each program had terminated. The first output did not contain any process ID of the program. In the second output, we could identify the process ID of the running program, while the third output no longer contained the program's ID. That means that the allocated memory has been freed correctly after the programs have terminated.

1.2 How confident are you that your programs are correct?

To make sure that our code work as intended, we have performed a series of manual testing, where we tested over 20 different coordinates to verify that the found cities/geographical places were indeed closest to the search query. To do this, we have shorted the given file of countries to about 100 inputs, so it was easier to determine the correctness of our program. Every query search ended up as a correct lookup.

We are pretty confident that `coord_query_naive` is correct, but we are less confident as far as `coord_query_kdtree` is concerned. As already said, we have performed a lot of manual tests to ensure the correctness of the output, but our query time for `coord_query_kdtree` is slower than that of `coord_query_naive`.

Because of that we went as far as to make an additional control version of `coord_query_kdtree2` to try a different approach with a searching algorithm. But even that new version was slow in comparison to `coord_query_naive`.

We believe that part of the reason is that we first build the kd_tree in our search function, which might add some extra time to the query time.

1.3 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?

Sadly, our solution in 4.2 appeared to be slower than our solution in 4.1. For instance, when searching the place with coordinates 40.00 40.00, it took `coord_query_naive` around 2800us to produce the result, while it took `coord_query_kdtree` around 10200us to find the target, which is a big difference.

As said above, we believe the reason behind the slow query time might be due to building the kd-tree structure. Besides, it could also be that we are going through the kd_tree in a wrong way, or it could be that we have structured the tree wrongly when building it, which is the same in both our versions.

However, we did mange to build two versions of `coord_query_kdtree`, which both are able to find the closest coordinates for a given query.

2 Querying by ID

2.1 Evaluate the temporal and spatial locality of the three programs you have implemented.

1. Speaking about the locality properties of the `id_query_naive.c`, it is relevant to focus on the function `lookup_naive()` because it usually makes sense to improve the code within loops.

   We evaluate that it has good locality with respect to the array `rs` in that it loops through all adjacent data elements in `rs` and searches for the desired ID by comparing the ID of each element with that of `needle`. And since it references `rs` elements in succession, we can say that it has spatial locality.

2. Speaking about the locality properties of the `id_query_indexed.c`, it also has good locality with respect to its two functions `lookup_indexed()` and `map()`. The former is designed in a similar manner as the `lookup_naive()` and exhibits spatial locality.

   The latter is meant to copy the ID data from the original array of records - `rs` - into the new array `irs` that merely contains ID data (copied from `rs`) and a pointer to the one cell/record in the array of records `rs`.

   We evaluate that `map()` has good spatial locality in that it iterates through all the elements in both arrays `irs` and `rs` in succession. On the other hand, when copying the value of ID from `rs` into the corresponding cell of `irs`, the function at hand might have bad locality because the two arrays might not be placed close to each other in the virtual memory.

3. We evaluate that the program `id_query_binsort` has good locality. As far as the sorting function `irs_sort()` is concerned, its locality depends on the locality properties of both `id_comparator` and `qsort`. While it is difficult to make any qualified evaluations of the latter as it is a library function, `id_comparator` has good spatial locality because it is designed so that it references and compares two adjacent elements in the array.

   Finally, the locality of the `lookup_indexed` depends on the locality properties of `binary_search`. In particular, it is relevant to analyse the body of the while-loop. The `mid`, `low` and `high` values have temporal locality as their values are being referenced per each iteration of the loop. At the same time, `binary_search` exhibits bad spatial locality, because the given values are not adjacent.

2.2 Do your programs corrupt memory? Do they leak memory? How do you know?

We believe that our programs do not corrupt memory due to the following two "tests". Firstly, we added additional sanitize flags (-fsanitize=address -fno-omit-frame-pointer) to our `Makefile` and compiled the program. After having run the programs, the compiler did not produce "3911==ERROR: LeakSanitizer: detected memory leaks".

Secondly, we ran the `pstree` command to see all running processes as a tree. We ran the command three times. First, before running the programs, second, while running each program individually and, third, after each program had terminated. The first output did not contain any process ID of the program. In the second output, we could identify the process ID of the running program, while the third output no longer contained the program's ID. That means that the allocated memory has been freed correctly after the programs have terminated.

2.3 How confident are you that your programs are correct?

To secure the correctness of our three programs, we have written three tests — one per program (cf. the code in c-files) — and none of them failed.

That means that the `lookup` function in the three programs never returns NULL.

2.4 How confident are you that your optimised programs are fast? How did you pick the benchmarking data?

We are quite confident in our optimised programs, as we have seen the times of the different programs, where they go in the expected order, of which is the following (from slowest to fastest build time):

1. `id_query_naive` had the fastest building time, but slowest search time. This is because it does not have to build any structures behind the scenes, but as a consequence, it has to look through every line of data in the file.

2. `id_query_indexed` had the middlemost build and search. It has to create the `indexed_data` and `index_record` structures to optimize the searches. This doubles the build time, but makes the search practically instant.

3. `id_query_binsort` has the slowest build time, but the fastest search time. This is because it builds upon `id_query_indexed`, sorts it and uses binary search as its method of searching

It was decided that the benchmarking data would by default being the last element in every record, as this gives the most accurate reflection of the process' runtime.