Computer Systems - A1: Dynamic Memory and Cache Optimisations

Aditya Fadhillah (hjg708), Usama Bakhsh (mhw630), Nikolaj Schaltz (bxz911)

October 9, 2022

3

3.5

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

id_query_naive

data from the function *mk_naive* is an example of spatial locality as *data* access the data element of *rs* and *n*. In the for-loop of the function lookup_naive there is an example of temporal locality as the data elements $data \rightarrow n$ are accessed in a loop, meaning that the same data elements are accessed multiple times.

id_query_indexed

The struct that is created from the function *build_index* is used in the function *mk_indexed*, where the value of that struct is the stored in a different struct. This is an example of a spatial locality As for temporal locality, the value of n in the for-loop of the function *build_indexed* is being compared with i for every iteration of the loop. The value of n remained unchanged for all iterations.

id_query_binsort

An example of spatial locality can be found in the function *lookup_indexed*, where the value of *irs[mid].osm_id* changes, but it changes into values of *osm_id* in the same struct.
An example of temporal locality is the *needle* in which *irs[mid].osm_id* is being compared with. During the while-loop needle is likely to be accessed multiple times.

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

By using the *free* function we have prevented memory leaks in our implementation. In our implementation of *id_query_indexed* we have also taken into consideration of freeing the allocated memory when we created the pointer *irs* in the function *build_index*. To see whether or not we actually succeed in freeing the allocated memory, we used an external program called Valgrind that detects and report any memory leaks that might occur when we run the program.

How confident are you that your programs are correct?

We are sure our programs are correct, because of how the code procedure works. For *id_query_naive* the code works by using linear search through the entire *20000records.tsv* file to find a matching oms-id. It does that by calling three functions.

- *mk_naive*. It takes a record pointer that is called *rs* and an integer as parameters. *malloc()* gets used to allocate space in the memory and returns a pointer to it. A test variable gets created and returned in the end of *mk_naive*. test is a pointer name to the variable *rs* and *n*

- *free_naive*. By using the build in *free()* function we can free up data to prevent memory leak.
- *lookup_naive*. This function takes the data pointer an a given interger *needle* as input. The function uses a for-loop from *1 to data pointer* to size n times to execute an if-statement. Inside the if-statement the function the *needle* gets compared to a given *osm-id* in the i item. If the comparison is true the address to the given data is returned.

This is basically how our linear search works. For *id_query_indexed* works almost the same way. To cut corners and make it faster *id_query_indexed* works by constructing a smaller array that contains all the *oms_id*'s and records. This is done by adding a another function.

- *build_index*. That uses the parameters, the record pointer *rs* and an interger size *n*. It uses *calloc()* instead of *malloc()*, because *calloc()* sets the allocated memory to zero. A for-loop gets used to for setting the data *rs* to be equal to the adress of *rs* which is the same as record from the *20000records.tsv* and also sets the *osm_id* to be equal to the *osm_id* in the index.

The last program is *id_query_binsort*. We are confident that the program is correct. We reused most of the code from *id_query_indexed* when making *id_query_binsort*. The only differences being that we added *qsort()* to sort the data in the smaller index, and that we changed the code in the *lookup* function from linear search to binary search. The code for binary search works by using a while-loop where we define the values *high* equal to n-1 and *low* equal to zero. While *low* is smaller or equal to *high*, the value *mid* gets defined or redefined as the middle point between *low* and *high*. An if-statement compares the *needle* to *osm_id*. If they are equal then we have found our correct location. Else if the given *oms_id* is smaller than the *needle*, then we redefine *low* to be *mid*+1 else if the given *oms_id* is higher then we redefine *high* as being *mid*+1. This loop continues until we have found the correct *oms_id*.

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

We can clearly see the differences in how each function take by comparing the time it takes to find the locations when running the program. Down below is a table over how much time each of them took to search for Guéthary with the osm_id of 166719.

Guéthary / 166719	id_query_naive	id_query_indexed	id_query_binsort
Reading records	24ms	22ms	28ms
Building index	0ms	0ms	2ms
Query time:	267us	51us	2us

We can see that in the case of finding Guéthary with the *osm_id* of 166719, it takes longer for the *id_query_naive* to find the record with the *osm_id* of 166719. This is of course because the naive code search through the data from top to bottom, and it also reads through all the content of the record and not just its index. And while *id_query_indexed* also search through the data from top to bottom, it only look at the index and the *osm_id* of the records. So the asymptotic running time for *id_query_naive* and *id_query_indexed* is $\mathcal{O}(n)$.

With *id_query_binsort* the time it takes to find the record with the *osm_id* of 166716 is much quicker. While it takes time to first sort the data, it is much faster to search through the data with binary search in comparison with the two previous *osm_id* search. The asymptotic running time for the binary search part of id_query_binsort is $\mathcal{O}(log(n))$. But the part of the implementation where we sort the data using quicksort, have an asymptotic running time of $\mathcal{O}(n^2)$.

In the case of finding the record with the *osm_id* of 166719, it is binary search that is the fastest. But it might not always be the case. As an example lets say that we have to find the record with *osm_id* of 2202162. In this case indexed search will be the fastest simply because that is the *osm_id* of the first record.

4

4.3

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

We avoid leak memory because we remember to free our memory after using it. We also avoid memory corruption because we do not need to access memory that we have freed.

How confident are you that your programs are correct?

The program *coord_query_naive* is supposed to be given a longitude and a latitude give back the closest location from *20000records.tsv*. We have done this in the *lookup* function where we construct the double *lowestDist* which is an arbitrary high value of 1000000 made to be overwritten by the first index. The line after that we create lowestData that is recordpointer that is set to be second line in *20000records.tsv*, since the first line does not contain data. We are using a for-loop to create the value *currentDist*, which is the current distance from the inputted longitude and latitude and rs[i], and also to traverse through the data set. *currentDist* is defined using the Euclidean distance. We use an if statement to check if currentDist is smaller than lowerDist, if that is the case then we update lowestData and lowestDist to the data and distance for the current index. The function returns the lowestData which should be the data of the closest place in the dataset.
We've also tested that the code works by copy pasting specific coordinates to check that they give the correct location, and by changing those coordinates a tiny bit so that they are not correct but close enough. These tests showed us that the program is able to find the closest location to the given longitude and latitude.

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?

The asymptotic running time of *coord_id_naive* is $\mathcal{O}(n)$ as it is a naive search meaning that the code goes through all the data from the top to the bottom to see which record have the closest latitude and latitude to the value that was inputted in the terminal.
While we have not been successful in implementing *coord_id_kdtree* we have surmise that the asymptotic running time for searching a kd-tree is $\mathcal{O}(log(n))$.

While *coord_id_kdtree* have the faster average running time, there are still cases where *coord_id_naive* is faster, and that is when the record that we are trying to find the first record in the data, meaning that the running time for *coord_id_naive* is $\mathcal{O}(1)$, which is the best-case time complexity of *coord_id_naive*.
The best-case time complexity of *coord_id_kdtree* would also be $\mathcal{O}(1)$, and that is when the top most index of the tree would directly match the desired value.