A0: file(1) — Unicode

Dea Felicia (dlr211), Said Said (hdn456), Kristian Jelsted (zxw463)

September 2023

1 Introduction

This report documents our solution to the A1 assignment. In this assignment we were tasked with creating programs that could locate entries in a data set, by osm-id and by coordinates. We were supposed to focus on memory, cache and the optimisation of our programs throughout the assignments.

2 Implementation and design

Osm-id

We have been tasked to create three different programs that find data by osm-ID: naive search, indexed search and binary search. In each of these 3 we have 3 functions: An mk function that makes the array, a free function that frees data and a lookup function that looks for a specific osm-id.

Naive and indexed: In naive and indexed search we use malloc() to allocate the correct amount of size bytes for the array in the mk function, in naive we then inset all the data from the array, in indexed we only inset the osm-ids thereby making it faster. In the free function we just used free(data) to free the data. In the lookup function we use a simple for-loop to go through the array and check if the needle corresponds with any osm-ID in our data set, if no match is found it simply returns NULL.

Binsort: In Binary search we have created a function that checks two osm-ids against each other and returns a value that tells us which of the osm-ids were the biggest. We use this function in our mk function, to use qsort() to sort our array of osm-ids, mk is otherwise like the mk from indexed. In our lookout function we use a while loop that runs as long as the left side of the array isn't smaller or equal to the right side. So until we've looked through the entire array. the program starts in the middle and then checks if the value we're searching for is higher or lower. It then eliminates the half of the array that is no longer needed and searches for the new middle, until it has searched the entire array.

Coordinates

We have also been tasked with creating two programs that finds a country with the coordinates closest resembling the coordinates we give it, the two programs is a naive search and a kd-tree search. These programs should also have a mk, free and lookout function each.

Naive: The naive coordinate search uses the same mk and free function as our id naive search. Because the program is created to handle coordinates that aren't a direct hit, we have created a function that estimates the distance between two coordinates. Then in our lookup function we have created a for-loop that goes through our data set and calculates the distance between our input and the current coordinates. If the distance between the two are smaller than the previous smallest difference the new coordinates are saved as the closest location to the coordinates we have given. When the loop has gone through the entire data set it returns the coordinates closest to the ones we asked for.

KD-Trees: In the implementation we have to get a bit creative compared to the other programs. Here we created a struct for the Node and a struct for the Point which saved the record and lon and lat coordinates. The node struct had a left and a right node which will either be a child or NULL if there is no more. We have a function for making the tree called kdtree which takes the points as input, n as number of elements and depth(which starts at 0) and returns a struct Node pointer. The function kdtree is simple copy paste of the pseudocode just written as C language. We use qsort to sort the elements on either lat or lon depending on the axis. And we recursively construct the tree. Our mk kdtree function just calls the recursive function kdtree. So the first node will be the root of the kdtree constructed. Our lookupkdtree uses a recursive function lookup that returns the closest point. This is also a basic copy paste of the pseudocode in C language. We use fabs to get the absolute value of the diff, we use euclidean to calculate the distance, and we use conditional operator "?" instead of if-else statements since it makes the node more readable, but uses the same logic. We recursively search through either right or left depending on the axis. But it is a pretty simple implementation of the pseudocode. Our lookupkdtree function uses the the recursive lookup function and gives the input(closest, query, and node(which is the root of the tree)). In return it gets the closest point and then we take record inside the closest point and return it.

3 Tests and evaluation

We needed to test our code to see if the speed of the programs where as expected and if any memory leaks were present.

Osm-id

For our osm-id focused programs we started by testing for memory leaks and corruption, to do this we ran Valgrind on all the index programs we created We are therefore fairly certain that no major memory leaks are present. We then moved on to testing our osm-id programs which we decided to do by testing 11 benchmarks for all three programs to test their speed. These include:

- The first and last osm-id in the data set to ensure that the begin and end of the data set was reached. First osm-id = 2202162, last osm-id = 3219806
- The biggest and smallest osm-id in our data set to ensure the first and last value was reached when the array was sorted, but also to test that

the programs could handle big and small osm-ids. The biggest osm-id = 5016646379, the smallest osm-id = 7374

- The middle osm-id in a sorted data and a couple of randomly picked osm-ids To check middle parts of the data: middle of sorted array = 1016146, index number 5000 = 1361397, index number 10000 = 43652 and index number 14998 = 470293880
- A couple of non existent osm-ids to see if the programs could handle wrong input, and because this should take the longest for our programs to get to, since they have to make sure that the osm-id is no where in the array. We have chosen: 9898, 678999, 4972144
- Finally we also tested with letters instead of numbers to make sure the programs could handle it.

We have attempted to test as many edge cases as possible so we know that the programs' speed are as expected for most values in our arrays. There might still be a couple edge cases we haven't tested. For instance, we haven't been able to find to of the same osm-id and have therefore not tested what would happen if you searched for an osm-id that had a duplicate, although we assume that the first would be chosen in case of naive and indexed search.

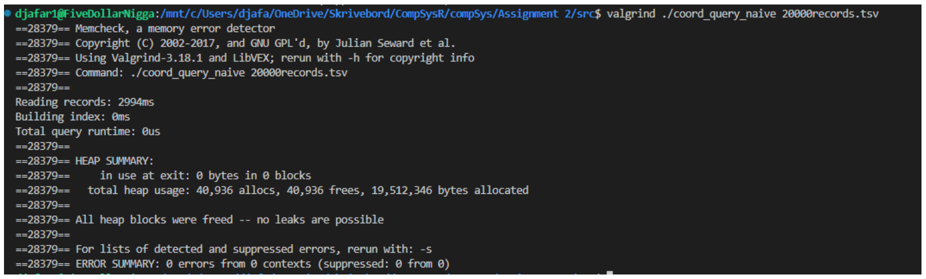
Naive: We can see from our tests that this program type is the slowest. The time goes from 1us for the first value to around 443us for a non existent id. This program starts from one end and goes through, meaning it will be very fast for the first id, but progressively become slower. The spatial locality of naive search is high, since we're going through the data set from one end to the other without making jumps, It does however have bad temporal locality since it only uses each value once and then moves on, never returning to already used values.

Indexed: The indexed search is a lot like the naive search, except the data set we go through is only osm-ids. The speed goes from 1us to roughly 152us, nearly cutting the time in half for most ids. The spatial locality of indexed-search is high as well because we go through the list from one end to the other, and the Temporal locality is again bad because it just go through the list from one end to the other.

Binsort: Binary search is the last type and by far the fastest. The time in our tests are between 1us and 3us. This is far faster than the other two programs, even when it is as it's slowest. The only point where indexed and naive are faster is when we ask for the first osm-id in the array 2202162, since this is the first value, the two other types reach, where as binsort starts at the middle. Binsort does not have a good spatial locality since we are jumping around in our array, it does however have good Temporal locality since There is a good chance the same memory locations will be reaches more than once.

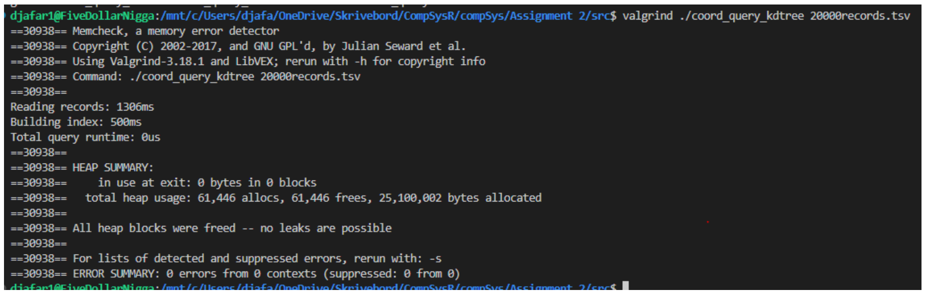
Coordinates

For the coordinates programs we also started by using valgrind to check for any major memory corruption or leaks. Since valgrind didn't find any leaks we are fairly certain that no major leaks are present. Our valgrind results can be seen here:



```
djafar1@FiveDollarNigga:/mnt/c/Users/djafa/OneDrive/Skrivebord/CompSysR/compSys/Assignment 2/src$ valgrind ./coord_query_naive 20000records.tsv
==28379== Memcheck, a memory error detector
==28379== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==28379== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==28379== Command: ./coord_query_naive 20000records.tsv
==28379==
Reading records: 2994ms
Building index: 0ms
Total query runtime: 0us
==28379==
==28379== HEAP SUMMARY:
==28379==     in use at exit: 0 bytes in 0 blocks
==28379==   total heap usage: 40,936 allocs, 40,936 frees, 19,512,346 bytes allocated
==28379==
==28379== All heap blocks were freed -- no leaks are possible
==28379==
==28379== For lists of detected and suppressed errors, rerun with: -s
==28379== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Figure 1: Valgrind coord naive



```
djafar1@FiveDollarNigga:/mnt/c/Users/djafa/OneDrive/Skrivebord/CompSysR/compSys/Assignment 2/src$ valgrind ./coord_query_kdtree 20000records.tsv
==30938== Memcheck, a memory error detector
==30938== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==30938== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==30938== Command: ./coord_query_kdtree 20000records.tsv
==30938==
Reading records: 1306ms
Building index: 500ms
Total query runtime: 0us
==30938==
==30938== HEAP SUMMARY:
==30938==     in use at exit: 0 bytes in 0 blocks
==30938==   total heap usage: 61,446 allocs, 61,446 frees, 25,100,002 bytes allocated
==30938==
==30938== All heap blocks were freed -- no leaks are possible
==30938==
==30938== For lists of detected and suppressed errors, rerun with: -s
==30938== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
djafar1@FiveDollarNigga:/mnt/c/Users/djafa/OneDrive/Skrivebord/CompSysR/compSys/Assignment 2/src$
```

Figure 2: Valgrind coord kdtree

Afterwards we tested our two programs to see if they worked as expected. Since the program estimates distance, we've both tested direct hits and non direct hits

- We tested the first and last coordinates in the data set, to see if the naive goes through the entire data set: First coordinates = 1.8753098 46.7995347, last coordinates = 3.7892891 47.4323246
- We tested a couple not specif values to see if the programs came to the same conclusion. We've tested for: 5 5, 45 45, 12 23, 52 1
- To test our Kd-tree we printed our kd-tree to a different file where we could see the different depths, we then tested them all. Kd-tree k=2 whit

20.000 records has a depths of 14, so we tested from depth 0 to depth 14 and then compared with our naive search to see if it matched up

We have attempted to find edge cases with the two programs. They both work well with direct hits, but it is a lot harder to verify the results of these two programs when they are estimating. With the osm-id there was one easy to check correct answer, but with these programs we would have to manually check that no coordinates were closer than what the programs gave us, which wasn't a possibility. We do know that our kdtre in general works a lot faster than our naive search, and that both our programs make the estimations, which makes us fairly certain that our programs serve their purpose with most input.

Naive: Has a average runtime of O(n) and since the for-loop goes through the entire list of n elements, we will always have to go through the entire list and will therefor have runtime O(n). Since the program goes from one end to the other of our data set that also means that if two location has the same distance to our needle, we will only return the first one, since the second one won't have a smaller distance to our input. This could be an instance where we won't get the result we were hoping for, but we haven't been able to find two coordinates with equal distance to a number and therefor haven't tested this.

KD-trees: Under most circumstance our kd-tree will be a lot faster than our naive search since Kd-trees have an average runtime of O(k log N) while naive has a runtime of O(N). If the data set had been small then the naive search would be faster since it takes time to build a kd-tree. Kd-tree will have a worst runtime of O(n), meaning worst case it might be as slow as naive, but in most cases kd-tree will be a lot faster. But as the dataset grows and grows, kd-tree will out perform naive more and more. But kd-tree overall is much faster than naive, but some cases it might be close to same running time as naive, and this as we clarified earlier. This will occur if the element searched for is deep in the kd-tree and can and will depend on the computer.

4 Conclusion

We have created 5 programs that go through a data set and find a specific country. The first 3 looks for osm-ids, the other 2 looks for specific coordinates. We have had focused on testing if these programs were as optimised as we assumed and if they worked as intended. There are a couple of things we haven't been able to test for, but all in all we can conclude that the speed of our programs are as expected. We have used Valgrind on all our programs and been told that no leaks are possible. Where the code might be more difficult to understand, we have written some code description to make it more understandable for the average computer science student.