Analysis of Algorithms

BLG 335E

Project 1 Report

HAKAN DURAN
duranh20@itu.edu.tr

Faculty of Computer and Informatics Engineering
Department of Computer Engineering
Date of submission: 26.11.2023

1. Implementation

1.1. Implementation of QuickSort with Different Pivoting Strategies

I have created a linked-list for my project. In linked list, there is city, population and forward-backward pointer information. My node class can be seen:

```
class node
{

private:
    string pcity;
    int ppopulation;
    node* pback;
    node* pforward;

public:
    node(string city, int population, node* back) : pcity{city},
        ppopulation{population}, pback {back}
    {};

    ~node(){};

    // Member functions do not listed.
};
```

The first i do in main() function is:

```
node* initial = csv_to_node(input);
int size = size_of(initial);

node** sc = new node*[1000];
shortcut_list_generate(initial, sc, size);
```

Firstly, i've used csv_to_node function in order to transform csv file to a linked-list. The linked-list's first node's pointer is initial. After determining the size of linked-list, i've created the sc array. What sc do is making linked-list traversal. I've used a special technique:

```
node** shortcut_list_generate(node* init, node** sc, int size){

    int order = 0;

    // forward_for is a function which makes forwarding traversal on
        specified pointer. It forwards for value in second argumant.
    node* mod100node = forward_for(init, 0);
    for(int i = 0; i < size; i = i + 100){
        int order = i/100;
        sc[order] = mod100node;
```

```
        mod100node = forward_for(mod100node, 100);
    }

    return sc;
}
```

mod100node is a node pointer which holds pointer that are in ranks of coefficients of 100 in the linked-list. sc array holds any mod100node value, so we can have an array of node pointers. Instead of starting every traversal from initial node, we will use this special array.

After that, we can select which algorithm is going to be used by using arguments that provided by user:

```
auto start = std::chrono::high_resolution_clock::now();

if (size > threshold){
    if(mode=="l"){
        quicksort(initial, 0, size-1, threshold, v, sc);
    }
    else if(mode=="r"){
        quicksort_random(initial, 0, size-1, threshold, v, sc);
    }
    else if(mode=="m"){
        quicksort_random3(initial, 0, size-1, threshold, v, sc);
    }
    else{
        cerr << "Unvalid mode.";
        return 1;
    }
}
else {
    insertion(initial, size, sc, 0);
}
```

The final part of main() function:

```
main(){
    .
    .
    .
    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> duration = end - start;
    std::cout << "Time taken by QuickSort with pivot strategy ' "<< mode
        << " ' and threshold " << threshold << ": "
    << std::chrono::duration_cast<std::chrono::nanoseconds>(end-start).
       count() << " ns." << std::endl;

    /* Sorted linked-list are transferring into a CSV file here */

    node_to_csv(initial, output);
```

```
13
14      /* Delete operations for deallocating memory */
15
16      node* tnode;
17      while(initial->get_forward() != nullptr){
18          tnode = initial->get_forward();
19          delete initial;
20          initial = tnode;
21      }
22      delete initial;
23      delete[] sc;
24
25      return 0;
26  }
```

1.1.1. Naive QuickSort

```
1   int partition(node* init, int low, int high, string v, node** sc){
2
3     int pivot = find(init, high, sc)->get_population();
4     int i = low - 1;
5
6     for(int j = low; j <= high; j++)
7     {
8       if(find(init, j, sc)->get_population() < pivot)
9       {
10        i++;
11        swap(init, i, j, sc);
12      }
13    }
14    swap(init, i+1, high, sc);
15    if(v == "v"){ verbose(init, pivot); }
16    return i+1;
17  }
18
19  void quicksort(node* init, int low, int high, int threshold, string v,
        node** sc){
20      if(low<high){
21          int pivot = partition(init, low, high, v, sc);
22
23          if (threshold == 1){
24              quicksort(init, low, pivot-1, threshold, v, sc);
25              quicksort(init, pivot+1, high, threshold, v, sc);
26          }
27          else {
28              // Not important for this subsection.
29          }
30      }
```

```
31 }
```

Naive QuickSort algorithm consists of two function. These functions takes a lot of arguments: node* init for linked-list, low and high values for sorting operations, v for verbose and node** sc for faster linked-list traversal:

"partition()" function makes given linked-list portion split in two and returns the pivot. In line 3, pivot is selected as the highest one in the linked-list portion. After the assigning i and j, algorithm looks the j value if it is smaller than the pivot. If it is, then i is being increased by one and, i and j swapped as it can be seen from line 6-13. In line 14, the final swap between i+1 and high values carry out so the linked-list portion can be splitted. There is also verbose option if it is wanted.

"quicksort()" function is actual function for sorting. After partitioning, we have 2 sub portions, which should also be partitioned. In order to partition those, we make 2 calls as it can be seen from line 23-26. By that way, we reach every sub portion, means for we reach every value as one. An after recursively turn back, we finally reach sorted linked-list.

There is also other functions which can be investigated. swap() function is one of them:

```
1  void swap (node* init, int i, int j, node** sc){
2      node* node1 = find(init, i, sc);
3      node* node2 = find(init, j, sc);
4
5      string tcity = node1->get_city();
6      int tpopulation = node1->get_population();
7
8      node1->set_city(node2->get_city());
9      node1->set_population(node2->get_population());
10
11     node2->set_city(tcity);
12     node2->set_population(tpopulation);
13 }
```

For swap, we find wanted values by using find() function. Then, node1's values copied into tcity and tpopulation, t means temporary. After the assignments, swapping is finished. Another important function is find():

```
1  node* find(node* init, int i, node** sc){
2      int mod = i%100;
3      i = (i - mod) / 100;
4      node* mod100pointer = sc[i];
5      return forward_for(mod100pointer, mod);
6  }
```

In order to find wanted value, we firstly should find the nearest pointer in sc array. In order to find it, we find the mod of i. After subsctracting mod from i, we can divide it by 100. Using i value and sc array, we reach the nearest pointer in linked-list. After

reaching the nearest pointer, it starts to traverse forwardly from that pointer for "mod" times.

Example is the situation when i is equal to 9172. mod becomes 72 and i becomes 91. sc[91] is 91*100=9100. pointer. forward_for function traverse for 72 times and final reach is wanted 9172. pointer.

If i would just use forward_for(init, 9172), we would traverse the linked-list about eighty percent for populationx.csv. find() function would traverse half of linked-list on average. Using sc array makes traversal value maximum(size(linkedlist/100), 99), which is only "138" for populationx.csv files, and 138 is maximum value. find() function traverses roughly "69" times on average with sc array while without sc, it is 13807/2 = "6904".

1.1.2. Random QuickSort

```
1  int partition_random(node* init, int low, int high, string v, node** sc)
2  {
3      srand(time(NULL));
4      int random = low + rand() % (high - low);
5      swap(init, random, high, sc);
6      return partition(init, low, high, v, sc);
7  }
8
9  void quicksort_random(node* init, int low, int high, int threshold,
       string v, node** sc){
10     if(low<high){
11         int pivot = partition_random(init, low, high, v, sc);
12
13         if (threshold == 1){
14             quicksort_random(init, low, pivot-1, threshold, v, sc);
15             quicksort_random(init, pivot+1, high, threshold, v, sc);
16         }
17         else {
18             // Not important for this subsection.
19         }
20     }
21 }
```

In quicksort_random() function, there is no difference from the quicksort() function but the function names changes. The partition_random() function is determining the random value in line 3-4. Swap between the random and high values carry out, then, actual partition function from Section 1.1.1 carry out and returning the value of pivot to quicksort_random() function.

1.1.3. 3-median QuickSort

```
1  int partition_random3(node* init, int low, int high, string v, node** sc
       )
2  {
```

```
3       srand(time(NULL));
4       int random = high;
5
6       int r1 = low + rand() % (high - low);
7       int r2 = low + rand() % (high - low);
8       int r3 = low + rand() % (high - low);
9
10      if ((r1 <= r2 && r2 <= r3) || (r3 <= r2 && r2 <= r1)){
11          random = r2;
12      }
13      else if ((r2 <= r1 && r1 <= r3) || (r3 <= r1 && r1 <= r2)){
14          random = r1;
15      }
16      else{
17          random = r3;
18      }
19
20      swap(init, random, high, sc);
21      return partition(init, low, high, v, sc);
22  }
23
24  void quicksort_random3(node* init, int low, int high, int threshold,
        string v, node** sc){
25      if(low<high){
26          int pivot = partition_random3(init, low, high, v, sc);
27
28          if (threshold == 1){
29              quicksort_random3(init, low, pivot-1, threshold, v, sc);
30              quicksort_random3(init, pivot+1, high, threshold, v, sc);
31          }
32          else {
33              // Not important for this subsection.
34          }
35      }
36  }
```

In quicksort_random3() function, there is no difference from the quicksort() function but the function names changes. The partition_random3() function determines 3 random value in line 3-8. Median of those 3 values determined in line 10-18. Then, random value swaps with highest value. Actual partition() function is called and returning the value of pivot to quicksort_random3() function.

1.1.4. Results

	Population1	Population2	Population3	Population4
Last Element	3927881758	49603578115	34144508465	102995725
Random Element	101491100	97227717	121683844	100053943
Median of 3	105656033	86001286	101786627	110633934

Table 1.1: Comparison of different pivoting strategies on input data.

The function which takes the most of time is last element function. It is because of the fact that randomized quicksort algorithms works faster when the input data is sorted or reverse sorted than naive quicksort algorithm. It is remarkable to see there is no difference for naive quicksort algorithm when the input data is randomized, which is population4.csv.

Another thing that is noticeable is median-of-3 is faster than randomized algorithm. It is because randomized algorithm can select its pivot too close to the edges. It is harder for median-of-3 algorithm.

1.2. Hybrid Implementation of Quicksort and Insertion Sort

```
1  void quicksort(node* init, int low, int high, int threshold, string v,
       node** sc){
2      if(low<high){
3          int pivot = partition(init, low, high, v, sc);
4
5          if (threshold == 1){
6              // Not important for this subsection.
7          }
8          else {
9              if(pivot-1-low > threshold){
10                 quicksort(init, low, pivot-1, threshold, v, sc);
11             }
12             else{
13                 insertion(find(init, low , sc) , pivot-low , sc, low);
14             }
15             if(high-pivot-1 > threshold){
16                 quicksort(init, pivot+1, high, threshold, v, sc);
17             }
18             else{
19                 insertion(find(init, pivot+1 ,sc) , high-pivot , sc,
                       pivot+1);
20             }
21         }
22     }
23 }
24
```

```
25 void insertion(node* init, int size, node** sc, int order){
26
27     int i, pkey, j;
28     string ckey;
29
30     for (i = 1; i < size; i++) {
31
32         pkey = find(init, i + order, sc)->get_population();
33         ckey = find(init, i + order, sc)->get_city();
34         j = i - 1;
35
36         while (j >= 0 && find(init, j + order, sc)->get_population() >
               pkey) {
37             find(init, j+1 + order, sc)->set_population( find(init, j +
                   order, sc)->get_population() );
38             find(init, j+1 + order, sc)->set_city( find(init, j + order,
                    sc)->get_city() );
39             j = j - 1;
40         }
41
42         find(init, j+1 + order, sc)->set_population( pkey );
43         find(init, j+1 + order, sc)->set_city( ckey );
44
45     }
46 }
```

If threshold is not 1, then hybrid algorithm takes its place. As it can be seen from line 8-21, there is a control of size if size is larger than threshold. If sub portion of linked-list's size is larger than threshold value, quicksort algorithm works. If it is not, then insertion sort algorithm works.

Insertion algorithm is efficient than quicksort when sub portions are smaller. The code starts with assigning 1 to i. Values of i. order of linked-list is assigning to pkey and ckey. j is also determined in the line of 34. In the while section (line 36-40), after the control, necessary assignments are carrying out and j is decremented by one. After enough reducing, when it finally get out of while section, j+1. value takes pkey and ckey.

1.2.1. Results

Threshold (k)	0.5	1	2	3	4
Population4	2046459787	1841856868	4072646049	5585789466	7132162103
Threshold (k)	5	6	7	8	9
Population4	10878577063	12590893241	12081954187	13933024940	18793503497

Table 1.2: Comparison of different thresholds on the hybrid QuickSort algorithm on input data.

As it can be seen, around the threshold value of 1k, algorithm is efficient. There is decrease of duration until the optimum value, and then duration increases again.