Assignment 3: Longest Prefix Matching (LPM) Programming report

s5524040 & s5622360

Algorithms and Data Structures 2023-2024

1 Problem description

General: Write a C program that takes as input a Routing Table that consists of the IPv4 address, subnet mask and interface index of **n** networks, as well as **m** request IP addresses and outputs the routing number of the network corresponding to the longest prefix of the requested IP address. The solution is implemented in 2 approaches:

Naive approach: For this part of implementation we are not using any Data Structures except the array. We are storing the IPs in a dynamically allocated 2D array and later locating the network IP that matches best the requested address by checking each address and returning the one with the lowest amount of different bits.

Improved approach: For this section, we are solving the same problem but in a rather faster and optimised way, by using a Trie structure which holds the binary representation of the IPv4 addresses of the routing table and runs each address-to-be-routed which through the trie, returning the number related to the best matching address from the 'routing table'. **Input-output behavior:** The input consists of a natural number n, followed by n addresses alongside theirs subnet masks and a corresponding interface number. This creates the 'routing table'. After that, another natural number m is accepted, followed by m address. The output consists of m numbers that correspond to the correct routing of each address in relation to the 'routing table' and the interface numbers.

Example input:
4
192.168.1.0/24 0
10.0.0.0/8 1
172.16.0.0/16 2
192.168.0.0/16 3
3
192.168.1.128
192.168.0.255
10.255.255.254

Example output:
0
3
1

Exaple input:
2
10.255.0.1/16 1
10.255.0.1/8 2
2
10.255.0.1
10.0.0.0

Example output:
1
2

2 Problem analysis

The implementation of the naive approach is rather straight forward, we are dynamically allocating and storing a 2D array of n rows, each row consisting of 20 chars(representing the maximum length of an IP address as input) and an interface number stored in an n-length 1D array .These 2 structures constitute the 'routing table'.A similar process is applied to the second set of addresses accepted as input. After that, every address that needs to be routed is selected in an iterative way and compared to each address of the 'routing table', with the interface number corresponding to the address with the least amount of differing bits being printed.

For the efficient implementation approach, the input is first converted to binary to ensure that every node in the Trie can have a maximum of two values (0 and 1), and then the bit pattern is inserted into the Trie. The algorithm of insertion can be described as following:

algorithm InsertInTrie(bitArray, mask, trie)
  level ← 0
  while level < mask **do**
    if index = 1
      if !trie.childRight
        trie.childRight ← makeTrie()
      trie ← trie.childRight
    else
      if !trie.childLeft
        trie.childLeft ← makeTrie()
      trie ← trie.childLeft
    level ← level + 1

After that, the addresses that need to be routed are received from the user and the matching network IP is searched in the Trie. Because each level of the trie represents a bit, we only want to go as low as the value of the mask, because the other bits represent the Host portion of the IP and aren't of our interest. As the trie is being traversed, if a node matches the current bit of the routing IP and holds information about the interface index, that index is remembered to be returned by the function, unless there exists a matching node which also has an interface index and is at a lower level, thus making it a longer matching pattern and a better fit.

3 Program design

We define the functions makeBitsInt, makeTrie, insertInTrie and searchInTrie that break down the complex task into smaller ones, making the code more readable, manageable and easier to debug. Also, we are defining the structure bitTrie that holds information about the current node and the pointers to its children.

Design choice. Saving only the bits up to the value of the mask is what prevents overuse of memory in the efficient implementation. Due to the fact that and address is considered a candidate for routing only if the number of identical bits in the address-to-be-routed is equal or greater than the value of the mask, saving any other values past the lat bit represented by the mask would be a waste of memory.

Time complexity. The time complexity of the naive approach depends both on the length of the pattern (length k, binary representation of the IPv4 address that needs to be routed) and the strings (length n, binary representation of the IPv4 addresses stored in the routing table), thus adding it up to O(kn) time. The efficient approach however, only relies on the length of the tree and has the reduced time complexity of O(n).

4 Evaluation of the program

We test the program with the following input:
2
192.168.15.1/16 1
192.291.17.2/8 2
1
192.168.18.6
And get the expected output 1. On top of the correct output, it is also important to ensure correct memory management, which can become a tedious task with Data Structures. It is essential that only the required amount of memory is allocated and that every allocation is freed up in the end. For these purposes, we've used

Valgrind to test our program and make sure that no leaks are possible:

==2367==
==2367== HEAP SUMMARY:
==2367== in use at exit: 0 bytes in 0 blocks
==2367== total heap usage: 49 allocs, 49 frees, 3,176 bytes allocated
==2367==
==2367== All heap blocks were freed – no leaks are possible
==2367==
==2367== For lists of detected and suppressed errors, rerun with: -s
==2367== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

5 Process description

Naive implementation: After the input is stored and the process of going through the address has started, the following will happen: the loop will go through each octet from both the address-to-be-routed and the current address from the routing table, and will check whether the bit positions match. If the octets are identical, then the counter goes down by 32, to signify the fact that the address if closer to being identical, otherwise, the program checks bit by bit the two values (the current 2 octets), and and each different bit increases the counter of differences. After it runs through the 'routing table', the program the interface number related to the best matching address.

Efficient implementation: The previous implementation fails to handle big inputs from both a time and memory perspective. As such, a different approach was taken using a trie. First, each element of the 'routing table' had its bit form stored in a trie (each bit represented a node, the number of bits stored was limited by the mask) and each node had an attribute equal by default to -1 or equal to the interface number when the height is equal to the mask. Because of this approach, when searching the best possible match we can simply return the last value different from -1 encountered while going down the trie, as such guaranteeing the beast possible match.

6 Conclusions

By implementing the same problem twice, in a trivial and a more complex manner, we could see, side-by-side how a solution that makes use of the correct Data Structure can be more readable, optimised and sometimes even the only logical solution. We could clearly envision how the linear approach of the array can "balloon up" in terms of runtime if the Routing Table consists of hundreds or even thousands of IP addresses and why it only makes sense to use a Trie-like structure for these kinds of problems. The advantage of this DS for this particular problem can be observed when we're analyzing the Big-O time complexity, which decreased from O(kn) to just O(n), where n is the length of the strings. Such a reduction of time complexity, by factor k, serves a significant enhancement of runtime and wait time response for the end user.

7 Appendix: program text

Listing 1: Efficient Implementation

```c
/* file : LPM_EfficientImplementation.c */
/* authors : Vrincianu Andrei - Darius (a.vrincianu@student.rug.nl) and Vitalii Sikorski (v.sikorski@student.rug.nl) */
/* date : March 11 2024 */
/* version: 1.0 */

/* Description:
  This program accepts n addresses and their respective numbers and stores them in a trie, after which it also accepts m address which are then routed to the longest
  matching prefix using the previously mentioned trie
*/

#include <stdlib.h>
#include <stdio.h>

```

```c
typedef struct bitTrie {
  int isLeaf;
  //routingNumber is the number taken alongside the address in the input
  int routingNumber;
  struct bitTrie *childLeft, *childRight;
} bitTrie;

// Creates an empty trie
bitTrie* makeTrie (void) {
  bitTrie* node = (bitTrie*)malloc(sizeof(bitTrie));
  // Checking if memory allocation was successful
  if (node == NULL) {
    printf("malloc failure");
    exit(EXIT_FAILURE);
  }
  if (node) {
    node->childLeft = NULL;
    node->childRight = NULL;
    node->isLeaf = 0;
    node->routingNumber = -1;
  }
  return node;
}

// Releases the allocated memory
void freeTrie (bitTrie* node) {
  if (node == NULL) {
    return;
  }
  freeTrie(node->childLeft);
  freeTrie(node->childRight);
  free(node);
}

// Transforms the IP address into a bit array
void makeBitsInt(int p1, int p2, int p3, int p4, int* bitArray) {
  int x = 0;
  int binaryImp = 7;
  // Starting from the least significant bit, the bit value is the remainder
     of division to 2
  for (int j = binaryImp; j >= x; j--) {
    bitArray[j] = p1 % 2;
    p1 /= 2;
  }
  // Adjusting the indexes
  x += 8;
  binaryImp += 8;
  for (int j = binaryImp; j >= x; j--) {
    bitArray[j] = p2 % 2;
    p2 /= 2;
  }
  x += 8;
  binaryImp += 8;
  for (int j = binaryImp; j >= x; j--) {
    bitArray[j] = p3 % 2;
    p3 /= 2;
  }
  x += 8;
  binaryImp += 8;
  for (int j = binaryImp; j >= x; j--) {
    bitArray[j] = p4 % 2;
```

```
    p4 /= 2;
  }
}

// Inserts the first n-bits of an address into the trie, where n is the the
   mask of said address, and sets the value of the last bit to the number
   relevant to the address
// taken from the input
void insertInTrie(bitTrie *root, int* bitArray, int maskNumber, int
   routeNumber) {
  int level;
  int index;
  int mask = maskNumber;
  int counter = 0;
  bitTrie *newtrie = root;
  for (level = 0; level < mask; level++) {
    index = bitArray[level];
    if (index) {
      if (newtrie->childRight == NULL) {
        newtrie->childRight = makeTrie();
      }
      newtrie = newtrie->childRight;
    } else {
      if (newtrie->childLeft == NULL) {
        newtrie->childLeft = makeTrie();
      }
      newtrie = newtrie->childLeft;
    }

  }
  newtrie->routingNumber = routeNumber;
  newtrie->isLeaf = 1;
}

// This searches the trie for the best match for the given address by
   returning the number relevant to the address with the last mask
   encountered,
int searchInTrie(bitTrie *trie, int* bitArray) {
  bitTrie *newtrie = trie;
  int level = 0;
  int index;
  int lastMask = -1;
  //The number length is fixed as 32 due to the fact that an address has
     exactly 32 bits
  int length = 32;
  for (level = 0; level < length; level++) {
    index = bitArray[level];
    // Upon discovering a value different from -1, the fact that a relevant
       value has been discovered is flagged
    if (newtrie->routingNumber != -1) {
      lastMask = newtrie->routingNumber;
    }
    if (index) {
      newtrie = newtrie->childRight;
    } else {
      newtrie = newtrie->childLeft;
    }
    if (newtrie == NULL) {
      break;
    }
  }
```

```
  return lastMask;
}

int main(int argc, char* argv[]) {
  int n;
  scanf("%d", &n);
  int p1, p2, p3, p4, mask, routingNumber;
  int bitArray[32];
  // Creating the root of the bit trie
  bitTrie* root = makeTrie();
  while (n) {
    // Getting the user input
    scanf("%d.%d.%d.%d/%d %d", &p1, &p2, &p3, &p4, &mask, &routingNumber);
    // Transforming the input address into an array of bits
    makeBitsInt(p1, p2, p3, p4, bitArray);
    // Adding the array of bits to the trie
    insertInTrie(root, bitArray, mask, routingNumber);
    n--;
  }
  int m;
  scanf("%d", &m);
  while (m) {
    scanf("%d.%d.%d.%d", &p1, &p2, &p3, &p4);
    makeBitsInt(p1, p2, p3, p4, bitArray);
    // Outputting the interface index of the matching network
    printf("%d\n", searchInTrie(root, bitArray));
    m--;
  }
  // Freeing up the memory
  freeTrie(root);
  return 0;
}
```

Listing 2: Naive Implementation

```
/* file : NaiveApproach.c */
/* authors : Vrincianu Andrei - Darius (a.vrincianu@student.rug.nl) and
   Vitalii Sikorski (v.sikorski@student.rug.nl) */
/* date : March 11 2024 */
/* version: 1.0 */

/* Description:
  This program accepts n addresses and their respective numbers and stores
     them in a trie, after which it also accepts m address which are then
     routed to the longest
  matching prefix using the previouslt mentioned trie.
*/



#include <stdlib.h>
#include <stdio.h>
#include <string.h>

//This function returns the value corresponding to the address with the
   least differing bits to the address which needs to be routed
void differencesOfAddresses(char** routingAddresses, int n, char**
   addressesToRoute, int address, int* correctRouting) {
  //In order to find the lowest value, we initialize this with a value that
     cannot be lower than any other possible one
  int maxAnswer = 33;
```

```c
  int counter = 0;
  while(n>counter) {
    int current = 0;
    int ipVal1 = 0, ipVal2 = 0;
    int i = 0;
    int j = 0;
    //This goes through all the addresses store in the routing table, converts them into integers, which are then converted into bits and the number of differences
    //between each address and the one we're currently checking is stored
    while(addressesToRoute[address][j]!='\0' && routingAddresses[counter][i]!='/') {
      ipVal1 = 0;
      ipVal2 = 0;
      while(routingAddresses[counter][i] != '.' && routingAddresses[counter][i] != '/') {
        ipVal1 = ipVal1*10 + (routingAddresses[counter][i] - '0');
        i++;
      }
      while(addressesToRoute[address][j]!='.' && addressesToRoute[address][j]!='\0') {
        ipVal2 = ipVal2*10 + (addressesToRoute[address][j] - '0');
        j++;
      }
      if(ipVal1 != ipVal2) {
        for(int l = 0; l<32;l++) {
        if(((ipVal2>>l) & 1) != ((ipVal1>>l)&1)) {
          current++;
        }

      }
      } else {
        //If the values are identical, then we artificially decrease the value to make sure we find the better match
        current-=32;
      }
      if(routingAddresses[n-1][i] == '.') {
        i++;
      }

      if(addressesToRoute[address][j] == '.') {
        j++;
      }
    }
    if(current < maxAnswer) {
      maxAnswer = current;
      *correctRouting = counter;
    }
    counter++;
  }

}



int main(int argc, char* argv[]) {
  int n;
  scanf("%d", &n);
  char** routingAdresses = malloc(n*sizeof(char*));
  int* routeNumbers = malloc(n*sizeof(int));

```

```
  for(int i = 0; i<n; i++) {
    //We are assigning a size of 20 characters because that is the aboslute maximum number of characters an address may hold
    routingAdresses[i] = (char*)malloc(20*sizeof(char));
  }

  //This accepts the first input addresses and stores them in an array
  for(int i = 0; i<n; i++) {
    scanf("%s %d", routingAdresses[i], &routeNumbers[i]);
    routingAdresses[i][strlen(routingAdresses[i])] = '\0';
  }

  int m;
  scanf("%d", &m);
  char** addressesToRoute = malloc(m*sizeof(char*));

  for(int i = 0; i<m; i++) {
    addressesToRoute[i] = (char*)malloc(20*sizeof(char));

  }

  //This accepts the following set of addresses and store them into an array
  for(int i = 0; i<m; i++) {
    scanf("%s", addressesToRoute[i]);
    addressesToRoute[i][strlen(addressesToRoute[i])] = '\0';
  }
  int correctRouting;
  int k = 0;
  //This goes through all the addresses that need ot be routed and prints the number associated with the desired address.
  while(m!=k) {
    differencesOfAddresses(routingAdresses, n, addressesToRoute,k,&correctRouting);
    printf("%d\n", routeNumbers[correctRouting]);
    k++;
  }

  free(routeNumbers);
  for(int i = 0; i < n; i++) {
    free(routingAdresses[i]);
  }
  for(int i = 0; i<m; i++) {
    free(addressesToRoute[i]);
  }
  free(routingAdresses);
  free(addressesToRoute);
}
```