컴퓨터네트워크

- 13. IP 주소 탐색 -

제 출 일	2018.12.07
학 과	컴퓨터공학과
학 번	201402391
이 름	이 병 만

※ 과제 목표 및 해결 방법

· 과제 목표

- Trie, Hash table - Nexthop IP 주소 탐색 프로그램 구현
- Trie, Hash table 성능 비교

· 해결 방법

1. trieTree 알고리즘

```
import pickle
import os

class Node(object):
    def __init__(self, key, data=None):
        self.key = key
        self.data = data
        self.children = {}

class Trie(object):
    def __init__(self):
        self.head = Node(None)

    def insert(self, insertIP, nextHop):
        curr_node = self.head

        for char in insertIP:
            if char not in curr_node.children:
                curr_node.children[char] = Node(char)

            curr_node = curr_node.children[char]

        curr_node.data = nextHop

        return self.head
```

[그림1]

➔ trieTree를 만들기 위해 두 개의 클래스를 선언하였다. Node 클래스는 트리에 사용되는 노드로서 key와 data를 필드로 가지고 있다. Trie 클래스는 트리를 생성하거나 탐색하는 기능을 할 수 있는 것을 선언하는 부분이다. 현재 파일은 트리를 생성하는 과정을 보여준다. 입력받은 String만큼 반복해서 왼쪽/오른쪽으로 이동하고 마지막 노드에 도착하면 nextHop을 삽입해주고 반환한다.

```
with open('pre.txt', 'r', encoding='utf8') as f:
    t = Trie()

    # if pickle file is exist
    while True:
        line = f.readline()
        if not line:
            with open('trietree.pickle', 'wb') as f1:
                pickle.dump(tree, f1)
                break

        insertIP = line.split(' ')[0].split('/')[0].split('.')
        mask = line.split(' ')[0].split('/')[1]
        nextHop = line.split(' ')[1]

        i_bin = ""
        for i in range(0,4):
            i_bin += bin(int(insertIP[i])).split('0b')[1].zfill(8)
        tree = t.insert(i_bin[:int(mask)], nextHop)
~
```

[그림2]

➔ 전처리한 파일을 열고, t 변수에 Trie() 생성자 저장한다. 한 줄씩 읽으면서 파일을 다 읽으면 trietree.pickle을 저장하기 위해 파일을 열고 트리를 저장한다.
한 줄씩 읽으면서 ip주소와 mask, next hop에 대해서 파싱을 하고 ip를 비트로 표현하여 트리를 생성한다.

```
import pickle
import os

class Node(object):
    def __init__(self, key, data=None):
        self.key = key
        self.data = data
        self.children = {}

class Trie(object):
    def __init__(self, object):
        self.head = object

    def search(self, searchIP):
        curr_node = self.head
        nextHop = ""

        # 하나의 입력값이 들어오면 반복을 실행
        for char in searchIP:
            if char in curr_node.children:
                curr_node = curr_node.children[char]

                if curr_node.data != None:
                    nextHop = curr_node.data
            else:
                return nextHop

        if (curr_node.data != None):
            nextHop = curr_node.data
            return nextHop
        else:
            return None
```

[그림3]

➔ 탐색을 처리하기 위한 파일이다. 삽입을 했을 때와 비교적 같지만 Trie 클래스에서 생성자 부분이 다르다. 생성자에 object를 넣어서 pickle 파일에서 불러온 트리를 사용하기 위해서 수정하였다.
탐색 함수도 삽입과 비교적 동일하다. 먼저 탐색할 IP를 입력받으면 하나 씩 트리를 확인한다. 현재 노드의 값이 있다면 nextHop을 저장하는 방식으로 반복한다. 모든 스트링을 방문했으면 마지막에 데이터가 None이 아니면 nextHop을 업데이트 해주고 그렇지 않으면 현재 저장된 nextHop을 반환해준다.

```
starttime = time.time()

# if pickle file is exist
if os.path.isfile('trietree.pickle'):
    with open('trietree.pickle', 'rb') as f:
        tree = pickle.load(f) # type tree
        t = Trie(tree)
        total_l = []
        # if pickle file is exist
        if os.path.isfile('random_ip_list.pickle'):
            with open('random_ip_list.pickle', 'rb') as f1:
                _list = pickle.load(f1) # random IP load

                for ip in _list:
                    s_bin = "" # search IP -> binary
                    _ip = ip.split('.')

                    for i in range(0,4): # create binary
                        s_bin += bin(int(_ip[i])).split('0b')[1].zfill(8)
                    nextHop = t.search(s_bin)

                    a = []
                    # [searchIP(binary), nextHop]
                    if not nextHop:
                        a = [ip, nextHop]
                    else:
                        a = [ip, nextHop.split('\n')[0]]

                    total_l.append(a)
        # save trie_ip_list
        with open('trie_ip_list.pickle', 'wb') as f3:
            pickle.dump(total_l, f3)
        print("sucess")
        finish = str(time.time() - starttime)
        print(finish)
```

[그림4]

➔ 먼저 pickle 파일이 있는지 확인한다. 있다면 파일을 열고 파일을 로드해서 저장한다. random으로 ip가 100만개가 저장된 파일을 열어서 저장한다. ip를 한줄 씩 읽으면서 비트로 변환시키고 그 값으로 탐색을 한다. 탐색 후 반환받은 nextHop이 None이면 그냥 그 값을 넣어주고 그렇지 않으면 개행문자를 제거하고 리스트에 저장한다. 저장한 리스트들을 total_l이라는 리스트에 넣어준다. 모든 리스트에 결과가 들어가면 trie에서 탐색한 ip 리스트와 nextHop이 있는 데 pickle을 사용해서 이 값을 저장해준다.

```
with open('oix-full-snapshot-2018-11-01-2200') as f:
    with open('pre.txt', 'w', encoding='utf8') as f1:
        tmp = []
        count = 0
        while True:
            count = count + 1
            line  = f.readline()
            if not line: # empty -> finish
                break
            if count > 5:# data filtering
                s = line.split()
                if len(tmp) == 0: # tmp is empty
                    tmp = s
                else:
                    if tmp[1] == s[1]: # network is same
                        if int(tmp[5]) < int(s[5]): # weight
                            tmp = s
                            continue
                        else:
                            if int(tmp[4]) < int(s[4]): # local
                                tmp = s
                                continue
                            else:
                                if len(tmp) > len(s): # path length
                                    tmp = s
                                    continue
                    else:
                        f1.write(tmp[1]+" "+tmp[2]+"\n")
                        print(tmp)
                        tmp = s
```

[그림5]

➔ **받은 데이터를 전처리해주는 기능이다. 먼저 전처리를 진행할 파일의 유무를 확인 후 파일을 연다.**
반복문을 진행해서 count가 5가 될 때까지 기다린다. 그 이유는 위에 필요없는 데이터가 존재하기 때문이다.
한줄 씩 읽으면서 tmp 배열이 비어 있으면 현재 읽은 값을 넣어주고 있다면 비교를 통해서 path가 가장 짧은IP를 가져와 데이터를 선정하여서 저장한다.

2. HashTable

```
import hashlib
import pickle
import time
```

```
def int_to_bit(ip):
    ip = ip.split('.') # 150.23.14.42
    i_bin = ""
    for i in range(0,4):
        i_bin += bin(int(ip[i])).split('0b')[1].zfill(8)
    return i_bin

with open('../trie/pre.txt', 'r', encoding='utf8') as f:
    hashTable = {}
    while True:
        line = f.readline()
        if not line:
            break

        network = line.split(' ')[0] # ex) 150.23.14.42/24
        ip = network.split('/')[0] # ex) 150.23.14.42
        mask = int(network.split('/')[1]) # ex) 24
        nextHop = line.split(' ')[1].split('\n')[0]

        bit = int_to_bit(ip)
        hash_key = hash(bit[:mask])

        value = [bit, nextHop]

        if hash_key in hashTable:
            print('Hashkey is exist')
            break
        else:
            hashTable[hash_key] = value
```

[그림6]

➔ 사용할 모듈을 선언해준다. 인트형을 비트로 만들어주는 함수를 선언한다. " . "으로 파싱해서 각 숫자를 비트로 변환 후 합쳐서 반환하였다.
전처리한 파일을 연다. HashTable을 사용할 딕셔너리를 선언해준다. 반복문을 실행하면서 전처리한 데이터를 일고 읽은 값을 이전과 동일하게 파싱하여 데이터를 저장한다. 저장한 비트 마스크만큼 해쉬함수에 넣어서 해쉬화한다. key의 value에 들어가는 value를 만들어준다. 만약 hashTable에 key가 같은 것이 없다면 빠져나온다.
없다면 Table에 value값을 넣어준다.

```
outputList = list()
starttime = time.time()

with open('../trie/random_ip_list.pickle', 'rb') as f1:
    with open('hashtable_result.txt', 'w') as fw:
        ip_list = pickle.load(f1)
        for ip in ip_list:
            nextHop = ""
            for prefix in range(8, 33):
                ip_add = int_to_bit(ip)[:prefix]
                hash_code = hash(ip_add)
                if hash_code in hashTable:
                    nextHop = hashTable[hash_code][1]
            if nextHop == "":
                value = (ip, None)
            else:
                value = (ip, nextHop)
            outputList.append(value)

        for i in range(len(outputList)):
    #        print(outputList[i][0]+" "+outputList[i][1])
            fw.write(outputList[i][0]+" "+str(outputList[i][1])+"\n")
        fw.write("-----------------------------------------\n")
        fw.write("Time : " + str(time.time() - starttime))
```

[그림7]

➔ 출력을 저장할 리스트를 선언한다. 검증할 데이터를 열어서 저장한다. 각 ip마다 반복문을 실행하고 prefix를 8부터 32까지 반복하면서 네트워크 id 추출하고, 그 값을 해시화한다. 길이 8의 네트워크 id가 일치하는 슬롯이 있는지 확인하고 있다면 nextHop을 저장한다. nextHop이 없다면 value에 None을 저장하고 그렇지 않으면 nextHop을 저장한더, 다음은 리스트에 저장한 값들을 파일에 작성한다. 마지막에 실행 시간을 출력해준다.

※ 실행 결과

TrieTree

```
mani@mani-VirtualBox:~/CN/trie$ python3 search_trie.py
sucess
33.59189701080322
```

HashTable

```
Time : 220.650160074234
```