컴퓨터 네트워크

-11주차(BGP Routing Table 에서 IP주소 탐색)-

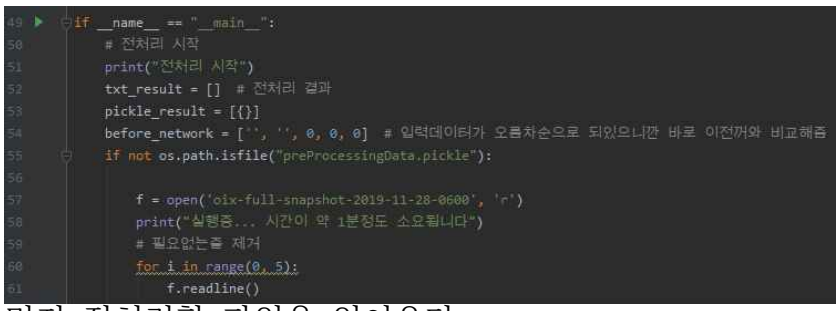
학번: 201404377
이름: 진승언

과제목표(도출해야할결과)

BGP Routing Table Data를 읽어 IP주소를 탐색한다. 먼저 제공된 BGP Routing Table Data를 읽어 전처리를 한다. 그 후 전처리된 파일을 읽어 Hash와 Trie 두 가지 알고리즘을 사용하여 input IP 주소 탐색 방법을 구현한다. 마지막으로 두 가지 알고리즘의 성능평가를 위해 그래프로 시각화하여 보여준다.

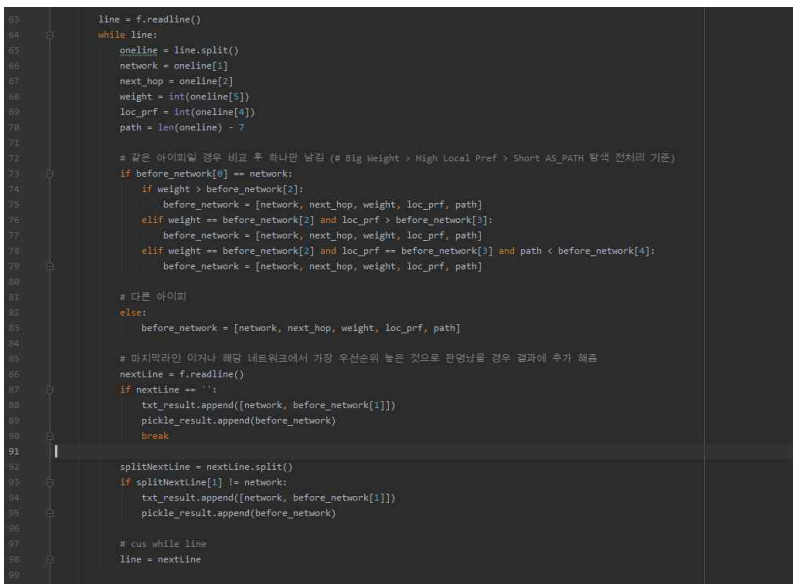
##시각화를 위해 hash와 trie 모두 하나의 파일에서 구현했습니다.

코드설명과과제해결방법



```
if __name__ == "__main__":
    # 전처리 시작
    print("전처리 시작")
    txt_result = []  # 전처리 결과
    pickle_result = [{}]
    before_network = ['', '', 0, 0, 0]  # 입력데이터가 오름차순으로 되있으니깐 바로 이전꺼와 비교해줌
    if not os.path.isfile("preProcessingData.pickle"):

        f = open('oix-full-snapshot-2019-11-28-0600', 'r')
        print("실행중... 시간이 약 1분정도 소요됩니다")
        # 필요없는줄 제거
        for i in range(0, 5):
            f.readline()
```

먼저 전처리할 파일을 읽어온다.



```
        line = f.readline()
        while line:
            oneline = line.split()
            network = oneline[1]
            next_hop = oneline[2]
            weight = int(oneline[5])
            loc_prf = int(oneline[4])
            path = len(oneline) - 7

            # 같은 아이피일 경우 비교 후 하나만 남김 (# Big Weight > High Local Pref > Short AS_PATH 탐색 전처리 기준)
            if before_network[0] == network:
                if weight > before_network[2]:
                    before_network = [network, next_hop, weight, loc_prf, path]
                elif weight == before_network[2] and loc_prf > before_network[3]:
                    before_network = [network, next_hop, weight, loc_prf, path]
                elif weight == before_network[2] and loc_prf == before_network[3] and path < before_network[4]:
                    before_network = [network, next_hop, weight, loc_prf, path]

            # 다른 아이피
            else:
                before_network = [network, next_hop, weight, loc_prf, path]

            # 마지막라인 이거나 해당 네트워크에서 가장 우선순위 높은 것으로 판명났을 경우 결과에 추가 해줌
            nextLine = f.readline()
            if nextLine == '':
                txt_result.append([network, before_network[1]])
                pickle_result.append(before_network)
                break

            splitNextLine = nextLine.split()
            if splitNextLine[1] != network:
                txt_result.append([network, before_network[1]])
                pickle_result.append(before_network)

            # cus while line
            line = nextLine
```

파싱을 하고 같은 네트워크 아이피일 경우 `Big Weight > High Local Pref > Short AS_PATH`기준으로 하나의 아이피만 남기고 리스트에 저장한다.

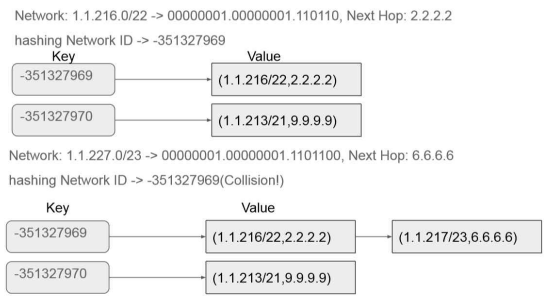
```
        # 텍스트파일로 저장 (hash와 trie에 사용하기 좋게 고침)
        with open('preProcessingData.txt', 'w') as f:
            for i in txt_result:
                tmpResult = i[0] + ' ' + i[1] + '\n'
                f.write(tmpResult)

        # pickle로 저장
        with open('preProcessingData.pickle', 'wb') as p_file:
            pickle.dump(pickle_result, p_file)
```

전처리한 ip리스트를 pickle과 txt파일로 저장한다. (txt 파일은 추후 탐색에서 읽어오기 위해 저장했다.)

```
f = open('preProcessingData.txt')
while True:
    line = f.readline()
    line = line[0:len(line) - 1]
    if not line:
        break
    # bit로 변환
    tmpId = ['', '', '']
    netwrokIdMask = line.split()[0]
    networkId = netwrokIdMask.split('/')[0]
    networkSplitId = networkId.split('.')
    networkMask = netwrokIdMask.split('/')[1]
    for i in range(0, len(networkSplitId)):
        networkSplitId[i] = str("{0:b}".format(int(networkSplitId[i])).zfill(8))
        tmpId[0] += networkSplitId[i]
    tmpId[0] = tmpId[0][:int(networkMask)]  # bit IP
    tmpId[1] = networkId  # IP
    tmpId[2] = line.split()[1]  # next hop
    bitIp = tmpId

    value = [bitIp[0], bitIp[2]]
    index = hash(bitIp[0]) % lineCount
    # 해시테이블에 데이터 추가
    if not hashIpTabel[index]:
        hashIpTabel[index] = [value]
    else:
        hashIpTabel[index].append(value)

# ip 입력파일 생성
txtFile = open("100000.txt", "r")
while True:
    line = txtFile.readline()
    if not line:
        break
    inputIpList.append(line)
```

전처리한 파일을 불러오고 해시테이블에 저장한다. 그리고 탐색할(입력한) 텍스트파일도 불러와 변수에 저장한다. 100000개의 input값이 들은 파일로 실행했다.

저장하는 방식은 밑의 참고사진과 같다.



```
    print("Hash 탐색 시작")
    # 해쉬탐색 시작
    hashStartTime = time.time()
    for ipList in inputIpList:
        mostSameHop = None

        bitIp = ''  # a.a.a.a -> 32 bit
        splitedIP = ipList.split('.')

        for i in range(0, len(splitedIP)):
            splitedIP[i] = str("{0:b}".format(int(splitedIP[i])).zfill(8))
            bitIp += splitedIP[i]
        # 8~32 비트별로 모두 탐색
        for i in range(8, 33):
            indexNum = hash(bitIp[0:i]) % lineCount
            isSearch = hashIpTabel[indexNum]
            if isSearch:  # 리스트 순회
                for kv in isSearch:  # [] in value's list
                    if kv[0] == bitIp[0:i]:
                        mostSameHop = kv[1]
                        break
        tmp = mostSameHop
        hashingResult.append(tmp)
        #시간리스트 추가
        hashEndtime = time.time()
        hashResultTime = hashEndtime - hashStartTime
        hashTimeList.append(hashResultTime)

    hashEndtime = time.time()
    hashResultTime = hashEndtime - hashStartTime
    # HASH 탐색 결과값 텍스트파일로 저장
    with open('HashSearchResult.txt', 'w') as f:
        for i in hashingResult:
            tmpResult = str(i) + '\n'
            f.write(tmpResult)
    # ----------------------------------------------------------------------위에까지
```

Hash탐색을 시작한다. 탐색과정은 위 사진과 같이 하면 된다. 탐색할 ip리스트를 하나씩 불러와 앞서 해시테이블에 저장한 거와 비교해서 찾은 ip를 결과 리스트에 추가해주면 된다.

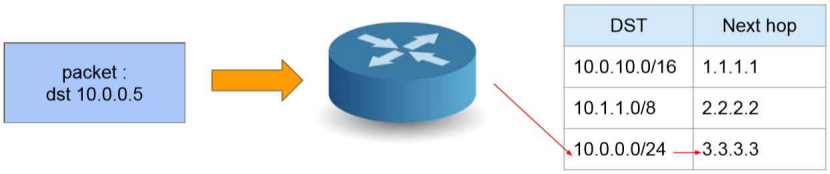
단, Longet Prefix Matching 방법에 의하여 IP 주소 8부터 31까지 Masking 수행하고 Subnet Mask한 결과를 Hash화 하여 Hash 키에대한 모든 Value를 찾아온 후 그 중 가장 긴 Subnet Mask를 가진 Network 주소의 Next Hop으로 결정하면 된다.

Longet Prefix Matching 방법은 다음 사진과 같다.

결과값은 텍스트 파일로 저장하였다.

Longest prefix Matching

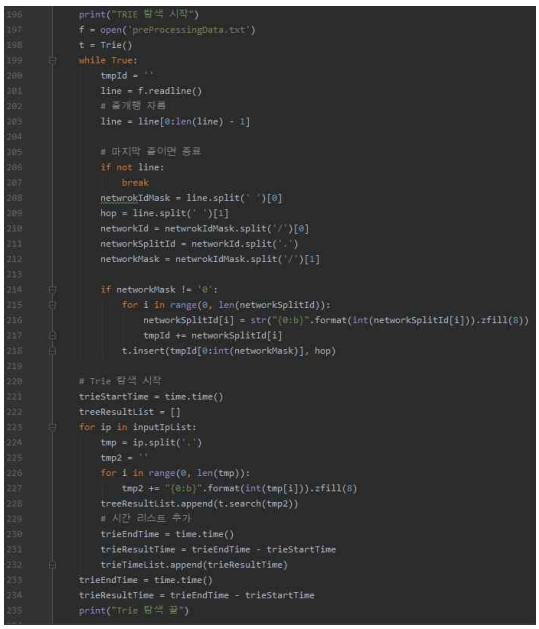
- 라우터가 패킷을 포워딩할때 Next Hop을 결정하기 위해 패킷의 목적지 주소와 라우팅 테이블의 각 엔트리에 있는 prefix와 비교 수행
- subnet mask 중 가장긴 mask를 선택하여 경로를 설정하는 것



DST	Next hop
10.0.10.0/16	1.1.1.1
10.1.1.0/8	2.2.2.2
10.0.0.0/24	3.3.3.3

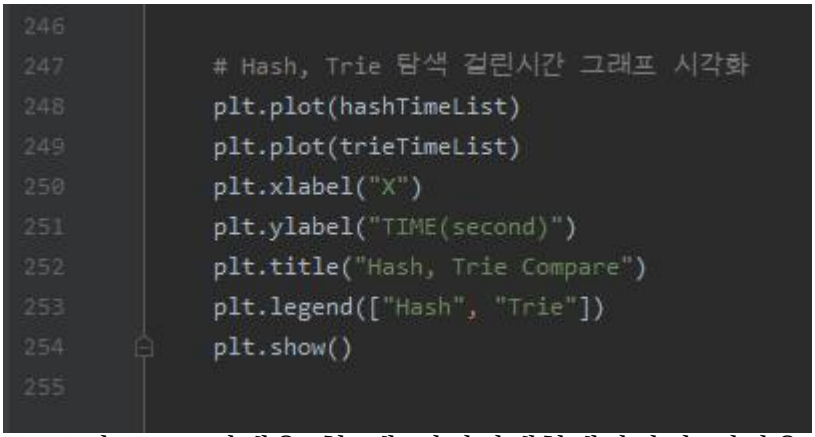
```python
class Trie(object):
    def __init__(self):
        self.head = Node(None)

    def insert(self, string, data):
        currentNode = self.head
        for char in string:
            if char not in currentNode.children:
                currentNode.children[char] = Node(char)
            currentNode = currentNode.children[char]

        if not currentNode.data:
            currentNode.data = data

    def search(self, string):
        currentNode = self.head
        rdata = None
        for char in string:
            if char in currentNode.children:
                currentNode = currentNode.children[char]
                if currentNode.data:
                    rdata = currentNode.data
            else:
                break

        if currentNode.data != None:
            return rdata
        else:
            return rdata
```

Trie는 먼저 트리구조를 만들어 주고 초기화, 삽입, 탐색하는 기능을 구현해줬다.



```python
print("TRIE 탐색 시작")
f = open('preProcessingData.txt')
t = Trie()
while True:
    tmpId = ''
    line = f.readline()
    # 줄개행 자름
    line = line[0:len(line) - 1]

    # 마지막 줄이면 종료
    if not line:
        break
    netwrokIdMask = line.split(' ')[0]
    hop = line.split(' ')[1]
    networkId = netwrokIdMask.split('/')[0]
    networkSplitId = networkId.split('.')
    networkMask = netwrokIdMask.split('/')[1]

    if networkMask != '0':
        for i in range(0, len(networkSplitId)):
            networkSplitId[i] = str("{0:b}".format(int(networkSplitId[i])).zfill(8))
            tmpId += networkSplitId[i]
        t.insert(tmpId[0:int(networkMask)], hop)

# Trie 탐색 시작
trieStartTime = time.time()
treeResultList = []
for ip in inputIpList:
    tmp = ip.split('.')
    tmp2 = ''
    for i in range(0, len(tmp)):
        tmp2 += "{0:b}".format(int(tmp[i])).zfill(8)
    treeResultList.append(t.search(tmp2))
    # 시간 리스트 추가
    trieEndTime = time.time()
    trieResultTime = trieEndTime - trieStartTime
    trieTimeList.append(trieResultTime)
trieEndTime = time.time()
trieResultTime = trieEndTime - trieStartTime
print("Trie 탐색 끝")
```

앞서 Hash에서 했던 것처럼 ip 리스트들을 Trie에 넣어주고 찾을 ip를 search()를 통해 탐색하였다. 그리고 Trie또한 결과를 텍스트 파일로 출력하였다. Trie 노드는 비트화한 ip를 차례대로 추가해주고 nextHop을 최종 노드에 저장해주면 된다. 탐색은 탐색할 ip를 비트로 변환하여 순차적으로 대입하고 찾아 맞는 일치한 비트까지의 노드를 탐색하여 nextHop을 알아낸다.

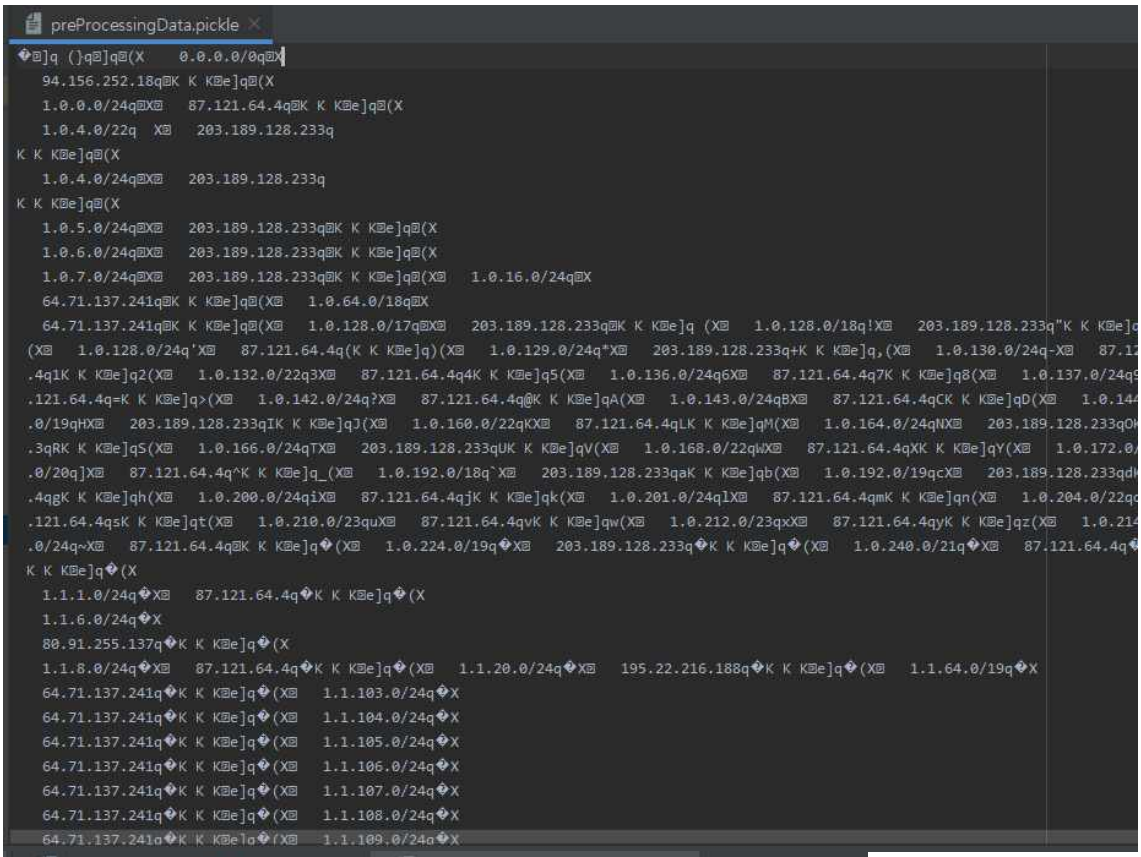


```
246
247     # Hash, Trie 탐색 걸린시간 그래프 시각화
248     plt.plot(hashTimeList)
249     plt.plot(trieTimeList)
250     plt.xlabel("X")
251     plt.ylabel("TIME(second)")
252     plt.title("Hash, Trie Compare")
253     plt.legend(["Hash", "Trie"])
254     plt.show()
255
```

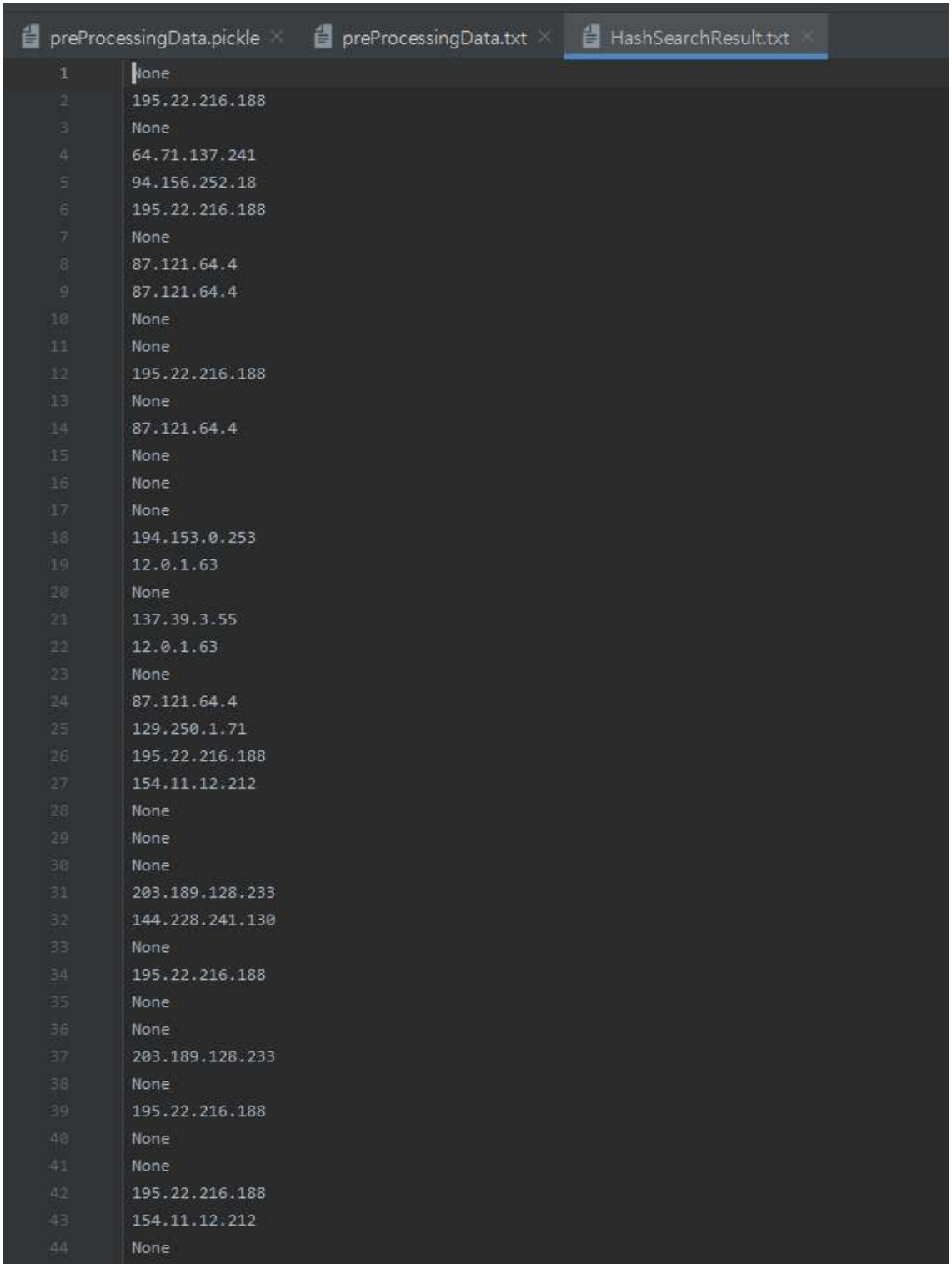
Hash와 Trie 탐색을 할 때 하나탐색할때마다의 시간을 time 리스트에 넣어주었다. 그것을 시각화한 코드이다.

[실행결과]

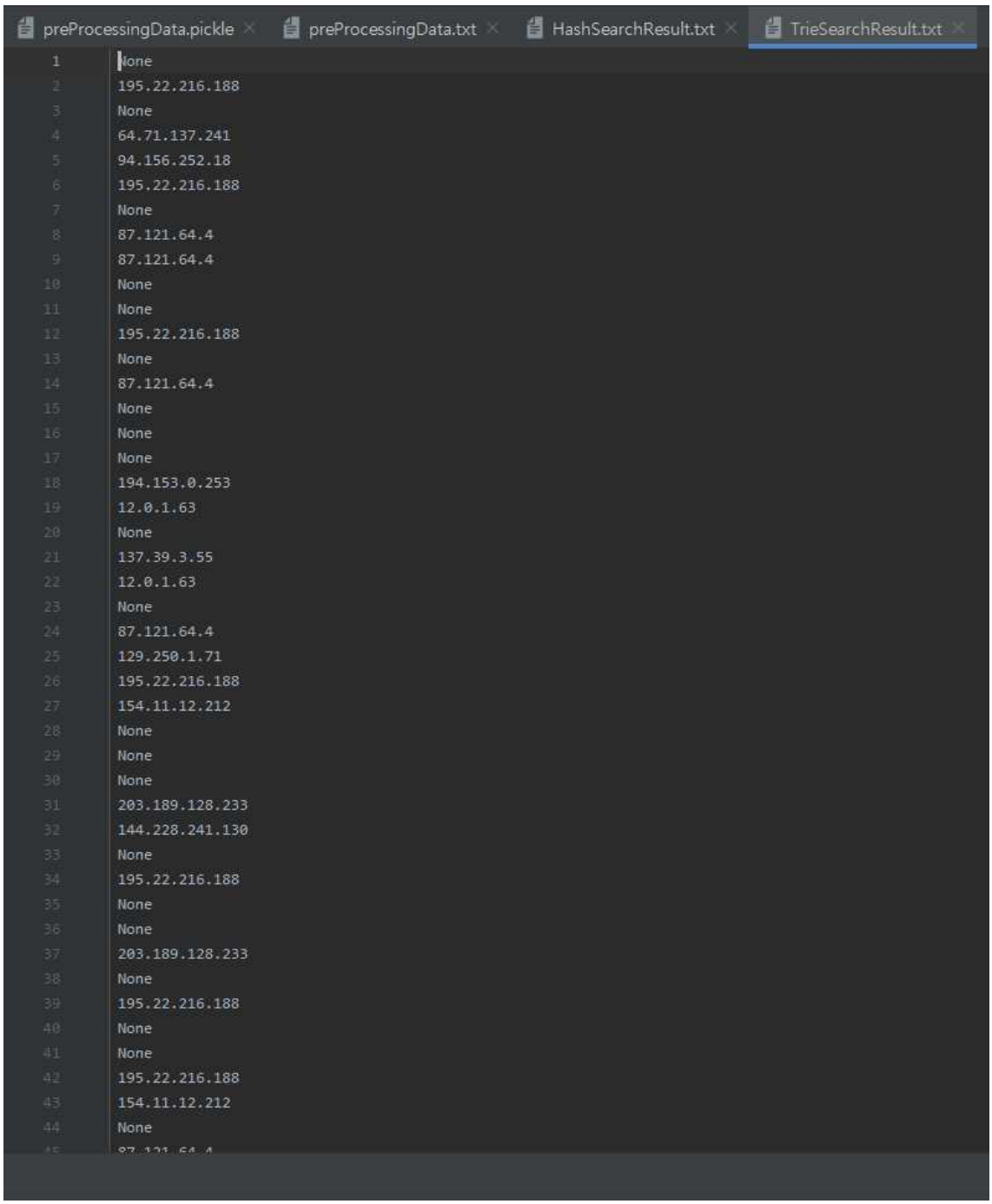
전처리 파일


preProcessingData.pickle
preProcessingData.txt
31.18.110.0/24 94.156.252.18
31.18.111.0/24 94.156.252.18
31.18.112.0/24 94.156.252.18
31.18.113.0/24 94.156.252.18
31.18.114.0/24 94.156.252.18
31.18.115.0/24 94.156.252.18
31.18.116.0/24 94.156.252.18
31.18.117.0/24 94.156.252.18
31.18.118.0/24 194.153.0.253
31.18.120.0/24 94.156.252.18
31.18.121.0/24 94.156.252.18
31.18.122.0/24 94.156.252.18
31.18.123.0/24 94.156.252.18
31.18.124.0/24 94.156.252.18
31.18.125.0/24 94.156.252.18
31.18.126.0/24 94.156.252.18
31.18.127.0/24 94.156.252.18
31.18.128.0/17 64.71.137.241
31.18.128.0/24 94.156.252.18
31.18.129.0/24 94.156.252.18
31.18.130.0/24 94.156.252.18
31.18.131.0/24 94.156.252.18
31.18.132.0/24 94.156.252.18
31.18.133.0/24 94.156.252.18
31.18.134.0/24 94.156.252.18

Hash탐색 결과

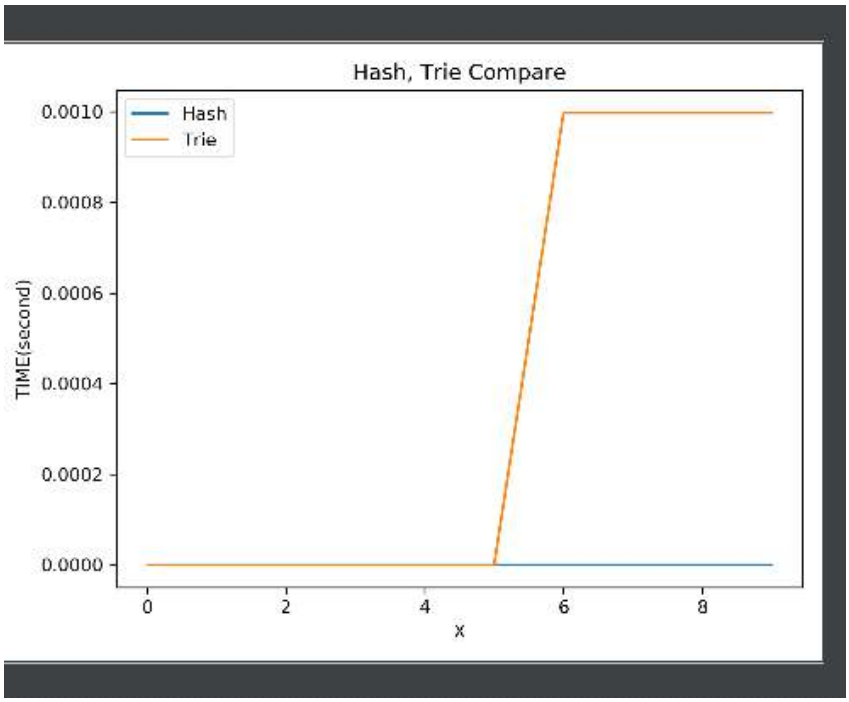

preProcessingData.pickle
preProcessingData.txt
HashSearchResult.txt
None
195.22.216.188
None
64.71.137.241
94.156.252.18
195.22.216.188
None
87.121.64.4
87.121.64.4
None
None
195.22.216.188
None
87.121.64.4
None
None
None
194.153.0.253
12.0.1.63
None
137.39.3.55
12.0.1.63
None
87.121.64.4
129.250.1.71
195.22.216.188
154.11.12.212
None
None
None
203.189.128.233
144.228.241.130
None
195.22.216.188
None
None
203.189.128.233
None
195.22.216.188
None
None
195.22.216.188
154.11.12.212
None

Trie탐색 결과(Hash와 일치함을 볼 수 있다.)

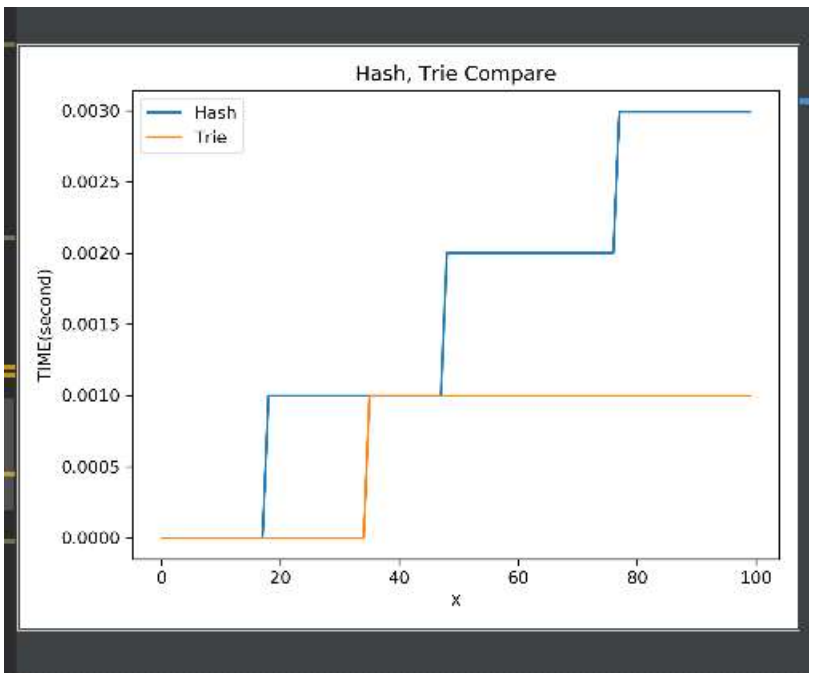


두 알고리즘 시각화 비교

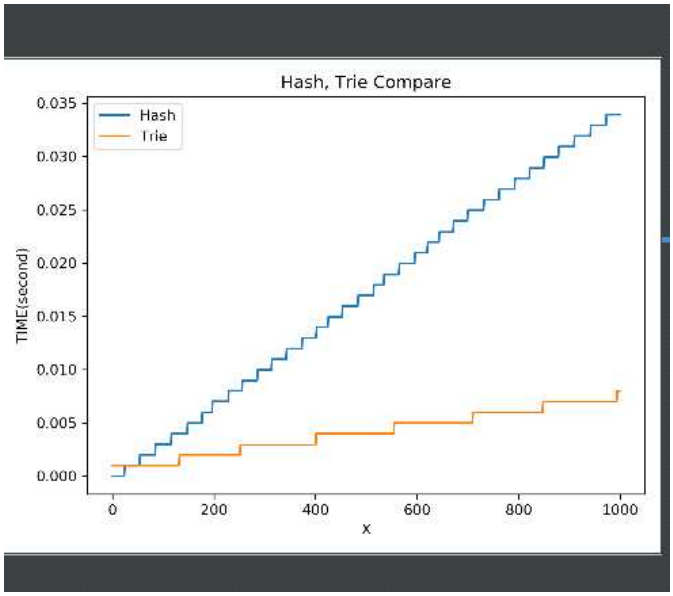
(10개)



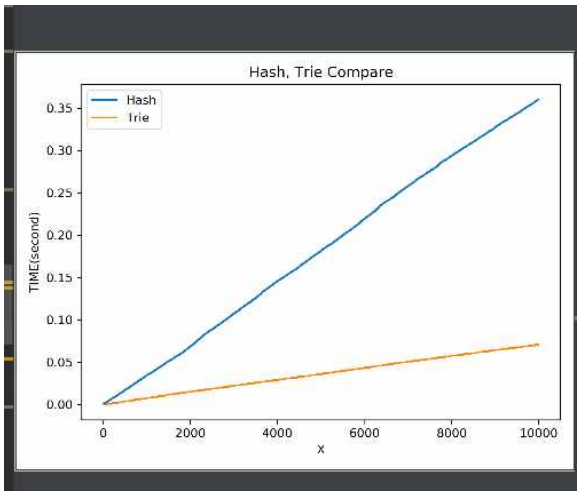
(100개)



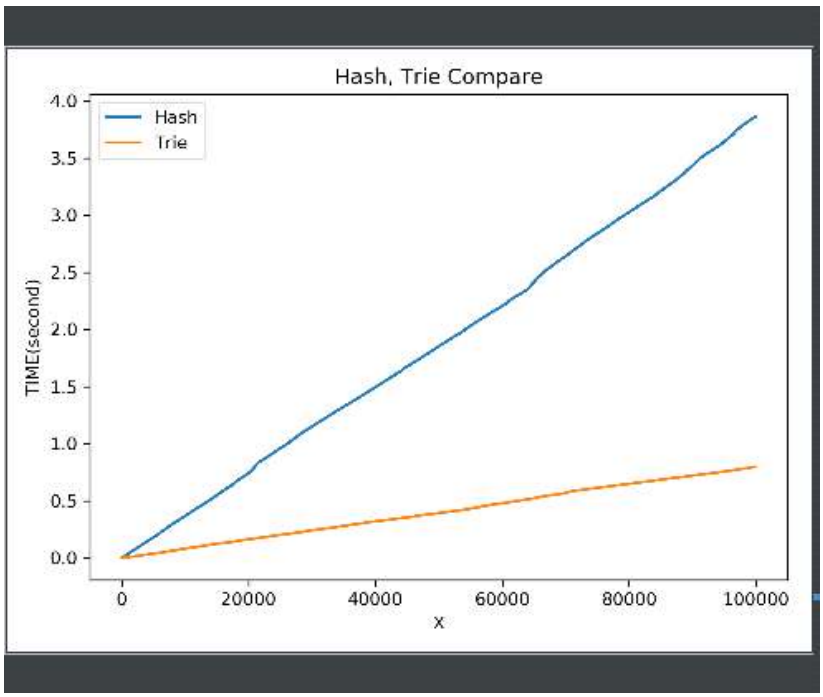
(1000개)



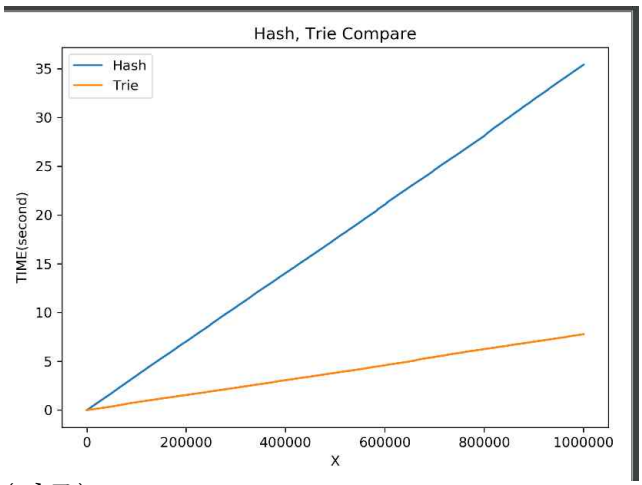
(10000개)



(100000개)



(1000000개)



(결론)

10개일 때 빼고 Trie가 우수함을 알 수 있다.

입력데이터 100000개를 기준으로 실험하였다.

Hash 탐색 시간 복잡도 : O(1)

Trie 탐색 시간 복잡도 : O(m), m은 탐색하려는 문자열 길이