자료구조

과제 8. 그래프 해싱

과목명 : 자료구조
담당교수 : 김승태
소속학과 : 응용통계학과
학번 : 20193011
이름 : 유승희

1. 사전 해싱하기

1.1 데이터 불러오기

제공된 사전 파일 randdict.TXT 파일을 읽기모드로 한 줄씩 불러와 랜덤 사전 dictionary 리스트에 저장해주었다. 단어들을 불러올 때 편의를 위해 줄바꿈은 제거하고 ':'를 기준으로 나누어 저장했다.

<코드>

```
# 사전 파일 불러오기
dictionary = []
with open('randdict.TXT', 'r') as file:
    for text in file:
        dictionary.append(text.strip('\n').split(' : '))
```

1.2 해시 함수 설정하기

해시 함수 만들기

dictionary에 있는 단어를 해시 테이블의 인덱스로 변환하기 위한 해시 함수를 다음과 같이 설정해주었다.

<해시 함수>
- 단어의 각 알파벳을 아스키코드로 변환 후 제곱하여 더한 값의 1~5번째 자리수 총 네자리수를 이용한다.
- 더해진 수의 10의 자릿수부터 10,000의 자릿수 까지 인덱스로 사용한다.

위와 같은 해시 함수를 만든 이유는 10,000개의 버킷과 20개의 적은 슬롯으로 해시 테이블을 사용하여 만들 수 있기 때문이다.

<코드>

```
# 해시 함수 설정
def hash_function(word):
    sum = 0
    for i in word:
        sum+=ord(i)**2
    sum = str(sum)[1:5]

    return int(sum)
```

해시 함수 테스트

만든 해시 함수로 해시 테이블을 구성했을 때 버킷 수, 슬롯 수를 확인한다.

<코드>

```
hash = []

for i in dictionary:
    ind = hash_function(i[0])
    hash.append(ind)

print('index 시작 : ', min(hash))
print('index 끝 : ', max(hash))
print('해시 테이블 버킷 수 : ', max(hash) - min(hash)+1)

cnt = []
for i in range(min(hash), max(hash)+1):
    cnt.append(hash.count(i))

print('해시 테이블 슬롯 수 : ', max(cnt))
```

<결과>

```
index 시작 :  0
index 끝 :  9999
해시 테이블 버킷 수 :  10000
해시 테이블 슬롯 수 :  20
```

- 0~9999 까지의 index를 사용할 수 있고 해당 해시 함수를 이용하면 해시 테이블의 버킷은 10,000개, 슬롯은 최대 20개로 구성된다.

1.3 단어 검색하기

정의한 해시 함수로 해시 테이블을 만들어 준 후에 검색할 단어를 입력받아 뜻을 출력하고 몇회 검색했는지 출력한다.

- 해시 테이블 초기화 : 파이썬의 append 기능으로 슬롯을 추가할 수 있으므로 버킷의 수 만큼만 2차원 배열로 해시 테이블(hash_table)을 초기화 해준다.
- 사전의 단어와 뜻을 해시 테이블에 넣기 : dictionary에 있는 단어를 해시 함수를 이용해 결과로 나온 값을 인덱스로 이용해 해시 테이블에 단어와 단어의 뜻을 넣어준다.
- 검색하기 : 검색 횟수를 세기 위한 count 변수를 0으로 초기화 한다. 무한 반복문으로 검색할 단어(word)를 입력받는다. 입력 받은 단어를 해시 함수로 나온 값을 index로 활용해 해시 테이블에서 바로 찾는다. 해당 index번째 있는 해시 테이블의 배열에 있는 단어(hash_table[ind][i][0])와 검색할 단어(word)가 일치하는지 순차적으로 탐색후 일치하면 해당 단어의 뜻(hash_table[ind][i][1])을 출력한다. 순차적으로 탐색할땐 for문을 이용해 반복할 때 마다 검색 횟수를(count)를 1 증가한다.

<코드>

```
## 단어 검색하기
# 해시 테이블 초기화
hash_table = [[] for i in range(10000)]

# 사전의 단어와 뜻을 해시 테이블에 넣기
for i in dictionary:
    ind = hash_function(i[0])
    hash_table[ind].append(i)


while 1:
    count = 0
    word = input('단어를 입력하세요 : ')
    ind = hash_function(word)
    for i in range(len(hash_table[ind])):
        count += 1
        if hash_table[ind][i][0] == word :
            print(f"{hash_table[ind][i][1]} ({count}회 검색)")
            print()
            break
```

<결과>

```
단어를 입력하세요 : add
vt.추가하다 (7회 검색)

단어를 입력하세요 : unman
vt.남자다움을 잃게 하다 (1회 검색)

단어를 입력하세요 : tippy
adj.엎어지기 쉬운 (1회 검색)

단어를 입력하세요 : apple
n.사과 (9회 검색)

단어를 입력하세요 : untrue
a.허위의 (1회 검색)

단어를 입력하세요 :
```

2. 최단 경로 찾기

2.1 다익스트라 알고리즘

도로를 도식화한 데이터에서 최단 경로와 거리를 찾기 위해 다익스트라 알고리즘을 다음과 같이 구현했다.

<다익스트라 알고리즘>

시작 정점에서 다른 모든 정점까지의 최단 경로를 찾는 알고리즘을 구현하기 위해 다음과 같은 배열을 이용한다.
- 시작정점으로부터의 최단경로 거리를 저장하는 dist[] 배열
- 방문한 정점 표시를 하기 위한 found[] 배열, 초기화 시 모든 항목이 False
- 바로 이전 정점을 저장해서 이전 정점을 따라 시작하는 정점까지 가는 경로가 최단 경로가 될 수 있도록 하는 path[] 배열

방문하지 않은 정점들 중 가장 거리가 짧은 정점을 찾는 함수

choose_vertex(dist, found) :
INF를 10,000로 설정한다. 거리의 최소(min)를 INF로, 최소 거리 정점의 인덱스(minpos)를 -1로 초기화 한다. 반복문을 이용해 모든 정점 중에서 방문하지 않은 최소 dist 정점을 찾는다. 해당 정점을 찾으면 최소 dist 정점의 인덱스를 반환한다.

<코드>

```
#최소 dist 정점을 찾는 함수
INF = 10000
def choose_vertex(dist, found):
    min = INF
    minpos = -1
    for i in range(len(dist)):
        if dist[i]<min and found[i] == False :
            min = dist[i]
            minpos = i
    return minpos;
```

시작 정점으로부터 다른 모든 정점까지의 최단 경로를 계산

shortest_path(vtx,adj, start):
정점리스트(vtx)와 인접 행렬(adj), 시작 정점의 인덱스(start)를 매개변수로 입력받는 함수이다.
정점리스트의 길이는 정점수(vsize)로, 인접 행렬의 start를 인덱스로 가지는 배열을 dist배열로, 시작 정점을 정점수만큼 path 배열로, False를 정점수 만큼 found 배열로, dist배열의 start번째는 0으로 (자기자신과의 거리는 0) 초기화 한다.
정점수(vsize)만큼 반복문을 돌아 choose_vertex함수를 이용해 최소 거리의 정점 u를 탐색한다. found배열의 u번째 요소를 True로 바꾸어 정점 u는 찾았다고 바꿔준다.
아직 찾아지지 않은 모든 정점에 대해 반복문을 돌며 더 짧은 경로가 있으면 dist배열을 업데이트 해주고 path배열에 이전 정점을 업데이트 해준다.
모든 계산을 끝내고 경로를 담고 있는 path 배열과 거리를 담고 있는 dist배열으르 반환한다.

<코드>

```python
# start에서부터 다른 모든 정점까지의 최단 경로 계산

def shortest_path(vtx, adj, start):
    vsize = len(vtx)
    dist = list(adj[start])
    path = [start]*vsize
    found = [False]*vsize
    found[start] = True
    dist[start] = 0

    for i in range(vsize):
        u = choose_vertex(dist, found)
        found[u] = True

        for w in range(vsize) :
            if not found[w]:
                if dist[u] + adj[u][w] < dist[w]:
                    dist[w] = dist[u] + adj[u][w]
                    path[w] = u

    return path, dist
```

2.2 최단경로 탐색하기

데이터 생성

도로를 도식화한 데이터를 입력해준다.

<코드>

```
# 데이터 생성
vertex = [0,1,2,3,4,5,6]
weight = [[0, 4, 8, INF, INF, INF, 10],
          [4, 0, 2, 5, 11, INF, INF],
          [8, 2, 0, INF, 9, 4, 5],
          [INF, 5, INF, 0, 7, INF, INF],
          [INF, 11, 9, 7, 0, 2, 8],
          [INF, INF, 4, INF, 2, 0, INF],
          [10, INF, 5, INF, 8, INF, 0]]
```

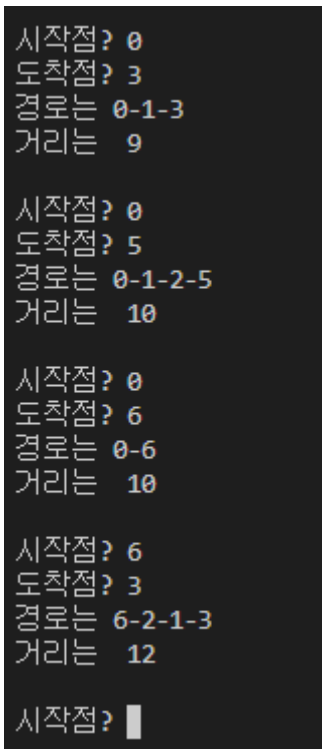
최단 경로 찾기

시작점과 끝점을 입력 받아 시작점으로부터 모든 정점까지의 최단 경로를 찾고 입력받은 끝점에 해당하는 최단 경로와 거리를 출력한다.

<코드>

```
# 최단 경로 탐색
while 1:
    start = int(input("시작점? "))
    end = int(input("도착점? "))

    path, dist = shortest_path(vertex, weight, start)

    e = end
    p = []
    p.append(vertex[e])
    while (path[e] != start):
        p.append(vertex[path[e]])
        e = path[e]
    p.append(vertex[path[e]])


    print("경로는 ", end = '')
    print(p[len(p)-1] , end = "-")
    for i in range(len(p)-2,0 ,-1):
        print(p[i], end = "-")
    print(p[0])

    print("거리는 ", dist[end])
    print()
```

<결과>


시작점? 0
도착점? 3
경로는 0-1-3
거리는 9
시작점? 0
도착점? 5
경로는 0-1-2-5
거리는 10
시작점? 0
도착점? 6
경로는 0-6
거리는 10
시작점? 6
도착점? 3
경로는 6-2-1-3
거리는 12
시작점?