**REPORT**



|  |  |
| --- | --- |
| **과목명** | 알고리즘 |
| **학과** | 경제학과 |
| **학번** | 12173082 |
| **이름** | 한규현 |
| **제출일자** | 11/21 |

**문제 제시**

‘규현’ 은 방학을 맞아 세계일주를 계획했습니다. 여행 전 비행기 항공권을 모두 구매했습니다. 하지만 고양이의 만행으로 모든 티켓이 뒤죽박죽 섞였습니다. 그래서 주어진 항공권을 모두 이용하여 다시 여행경로를 짜려고 합니다. 규현은 인천공항 ”ICN” 공항에서 출발합니다.

항공권이 2차원 배열로 제공되었을 때, 모든 항공권을 사용하는 최적 루트를 찾아 return 하십시오. Ex) 항공권 = [[“ICN”, “NRT”], [“ICN”, ”PEK”], [“NRT”, ”JFK”], [“JFK”, “BOS”]]

제한사항

1. 공항의 이름은 티켓에 영어 대문자 3글자로 주어집니다.
2. 공항의 수는 항상 3개 이상입니다.
3. Tickets의 [“A”, “B”]는 “A”에서 “B”로 향하는 항공권을 의미합니다.
4. 주어진 그래프는 연결 그래프이지만, 강한 연결 그래프는 아닐 수도 있습니다.
5. 항공권의 경로가 2개인 경우 알파벳 순서가 우선되는 경로가 정답입니다.
6. 티켓의 모든 도시는 방문할 수 있습니다.

**문제분석 및 모델링**

출발 노드는 항상 “ICN”으로 고정된다. 현재 노드와 연결된 다음 노드를 순차적으로 탐색해야 한다. 이는 효과적 탐색을 위해선 깊이 우선 탐색을 활용 해야함을 의미한다. 다만 주의할 점은 한 출발점에서 도착지가 여러 개가 들어오는 경우 즉, 출발자의 value값이 여러 개가 들어오는 경우 알파벳 순서에 의해 앞에 오는 도착역을 우선 방문해야 한다. 또한 제한조건 6에 의해 모든 도시를 방문할 수 있는지에 대한 여부는 문제가 되지 않기에 신경 쓰지 않아도 된다.

우선, DFS를 활용하여 문제를 해결하기 때문에 티켓 리스트를 그래프의 형식으로 변환시키는 과정이 필요하다. 결과값이 여러 개인 경우 알파벳 순서가 앞선 값이 정답이 되기에 변환 과정에 정렬을 추가한다. 이후 깊이 우선 탐색을 정의한다. 하지만 티켓은 방향 그래프의 형식을 가지며, 일반적인 DFS를 사용할 경우 자신이 지나온 노드를 재방문하지 못한다.

특히, 편도 티켓이 탐색의 우선순위에 있을 경우 깊이탐색이 끝나게 된다. 또한 같은 장소를 여러 번 방문하는 티켓이 존재하는 경우에도 문제점이 생긴다. 때문에, 탐색이 완료되기 위해선, 티켓을 모두 사용했음을 인지해야 한다. visit 리스트를 활용하는 기존의 방식 대신 POP()을 통해 딕셔너리에서 방문 티켓을 삭제하는 방식을 활용해야 한다. 또한, 편도 티켓의 문제를 해결하기 위해, dictionary의 key 함수에 현재 노드가 포함되었는지 확인한 뒤 편도티켓이 증명된 경우 미리 visit리스트에 추가하여 가장 마지막에 방문 되도록 한다. 방문한 노드는 모든 재귀가 완료되고 복귀하는 과정에서 answer리스트에 stack의 형식으로 저장한다.

**문제해결 알고리즘 개발**

def chanage\_tickets(tickets):

ticket\_dic = {}

for [key,item] in tickets:

if key not in ticket\_dic.keys():

ticket\_dic[key] = []

ticket\_dic[key].append(item)

for array\_d in ticket\_dic.values() :

array\_d.sort()

return ticket\_dic

def dfs(start,ticket\_dic,visited =[]):

while start in ticket\_dic.keys() and ticket\_dic[start] != []:

path = ticket\_dic[start].pop(0)

dfs(path,ticket\_dic,visited)

visited.insert(0,start)

def solution(tickets):

visited = []

tickets\_dic = chanage\_tickets(tickets)

dfs("ICN",tickets\_dic,visited)

return visited

**알고리즘 성능분석**

change\_tickets 의 시간 복잡도는 파이썬에서 제공하는 sort()의 시간복잡도가 N log N이기 때문에, O(1 + N + N \* N log N) = O(N log N)이다.

인접 리스트를 활용해 그래프를 작성한 경우, dfs는 한 정점마다 한 번씩 호출된다. 즉, |V|번 호출된다. dfs() 한 번의 수행 시간은 전체 인접 간선을 검사하는 while문에 의해 통제되는데, 모든 정점에 대해 dfs를 실행하면 방향 그래프의 경우 모든 간선을 정확히 한 번 확인한다. 깊이 우선 탐색의 시간 복잡도는 O(|V|+|E|)이다.

전체 알고리즘의 시간 복잡도는 O(N log N + N) = O (N log N)이다.

**참고문헌**

1. "Depth First Search." (2021년 11월 21일). *InterviewBit*. n.d. 수정, <https://www.interviewbit.com/tutorial/depth-first-search/>.
2. "TimeComplexity." (2021년 11월 21일). *wiki.python*. last modified Aug 18, 2020, https://wiki.python.org/moin/TimeComplexity.