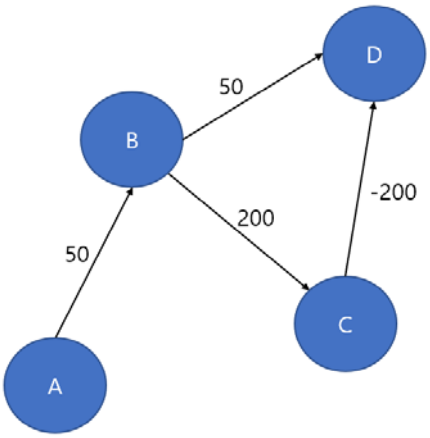
사범대학 과학교육과
1748022 김유진

중간과제 – "다익스트라 알고리즘"

음수 가중치가 있는 그래프에서 다익스트라 알고리즘이 제대로 동작하지 않는 예)



다음과 같은 그래프가 주어졌을 때, A에서 D로 가는 최단 경로를 다익스트라 알고리즘을 통해 구한다고 가정한다. 다익스트라 알고리즘에 의하면 A-> B-> D의 경로가 선택될 것이다. 왜냐하면 경로 A->B가 선택된 상황에서 B->D 로 가는 가중치는 50인 반면 B->C로 가는 가중치 200으로, 이를 고려하면 A->B->C는 A->B->D 가 가지는 가중치를 초과하기 때문이다. 하지만 C->D로 가는 경로에 음수 가중치 -200이 있으므로 실제 최단 경로를 구하면 A->B->C->D가 된다. 따라서 다익스트라 알고리즘은 음수 가중치가 있을 때 최단 경로를 보장하지 않는다.

이유)

다익스트라는 한 번 경로를 확정한 정점에 대해서 다시는 갱신이 일어나지 않는다. 하지만 음수인 경로를 가지고 있다면 경로를 확정했음에도 불구하고 가중치를 갱신을 할 수가 있게 된다.(더 짧은 경로가 존재할 수 있다) 이는 다익스트라의 '최단경로가 보장되지 않은 남아있는 정점들 중, 최소거리를 갖는 정점은 최단경로이다'가 보장되지 않으면서 발생한다. 최단경로임이 보장되지 않는 정점에서, 음수 간선을 타고 다른 정점으로 이동한 경로가 현재 남아있는 정점들 중 최소거리보다 더 작아질 수 있기 때문이다.

개선방안)

이러한 문제점을 해결할 수 있는 방법은 **벨만-포드 알고리즘을 사용하는 것**이다. 벨만-포드 알고리즘은 다익스트라와 같은 방식으로 동작하지만, 모든 노드를 바탕으로 순환을 진행함으로써, 음의 가중치도 계산이 가능하며 경로 내에 음의 순환이 존재하는지 체크할 수 있는 기능을 가지고 있다.

다익스트라가 한 번 경로를 확정한 정점에 대해서 다시 고려하지 않는다면, 이 알고리즘의 경우

경로를 확정할 때마다 모든 경우의 수를 다 탐색하면서 최소비용을 찾게 된다. 모든 간선들을 탐색하면서, '한번이라도 계산 된 정점' 이라면 해당 간선이 잇는 정점의 거리를 비교해서 다시 업데이트 한다. 이를 모든 간선들이 모든 정점들을 최단거리로 이을 수 있을 만큼 반복한다.
이 경우 시간복잡도는 다익스트라 알고리즘보다 커지지만, 엣지의 가중치가 음수일 때에도 최단경로를 구할 수 있다는 장점이 있다.

참고자료 - 의사코드 첨부

```
{{{#!syntax java
BellmanFord(G,w,s):
//초기화 과정
for each u in G.V:     //노드를 초기화 하기
      distance[v] = inf      //모든 노드의 최단거리를 무한으로 지정
      parent[v] = null       //모든 노드의 부모 노드를 null 값으로 지정
distance[s] = 0 //출발점의 최단거리는 0 으로 지정한다
//거리측정 과정
for i from 1 to len(G.V):   //노드의 숫자만큼
     for each (u,v) in G.E:   //모든 변을 체크해 최단 거리를 찾아본다.
          if distance[u] + w[(u,v)] < distance[v]:
          //만약 u 를 경유하여 v 로 가는 거리가 현재 v 의 최단 거리보다 짧으면
               distance[v] = distance[u] + w[(u,v)]  //그 거리를 v 의 최단거리
로 지정
               parent[v] = u   //u 를 v 의 부모 노드로 지정
//음수 사이클 체크 과정
for each (u,v) in G.E:
     if distance[u] + w[(u,v)] < distance[v]:
          return false //음수 사이클을 확인하고 알고리즘을 정지
return distance[], parent[]
}}}
```

#1916번 : 최소비용 구하기

① 알고리즘 설명

1) 도시의 개수(정점의 개수)와 버스의 개수(간선의 개수), 그리고 간선의 가중치를 입력받아 인접배열 형태로 저장한다.

2) 출발 노드를 설정하고, 출발점에서부터 연결된 각 노드의 가중치를 비교하여 아직 방문하지 않은 노드 중 가장 최소 비용을 가지는 vnear 노드를 찾는다.

3) 해당 노드를 이미 선택된 노드들의 집합에 추가하고, 이를 고려해서 다른 노드로 가는 최단거리 값을 갱신한다.

4) 이 과정을 출발점으로부터 모든 정점이 연결될 때까지 n-1번 반복한다.

5) touch[i] 배열을 이용하여 도착지점까지 최단경로를 출력한다.

② 소스코드 설명

```
import sys

def dijkstra(start,stop) :  #다익스트라 알고리즘을 수행하는 함수 정의

    for connect in range (n-1) :    #모든 정점이 선택될 때 까지 n-1 번 반복
        min = float('inf')  #초기값을 매우 큰 값으로 할당

        #length[i] 를 모두 검사하여, Y 로부터 최소 비용을 갖는 노드 vnear 을 찾는다.
        for i in range (1,n+1) :
            if (0<length[i]<min) :  #이미 선택된 값은 고려하지 않는다.
                min = length[i]
                vnear = i

        #현재까지의 가중치와 새로 추가된 vnear 에서 갈 수 있는 가중치를 비교한다.
        for i in range (1,n+1) :
            # 만약 현재 값보다 더 작은 값을 가진다면, length 와 touch 배열을
업데이트한다.
            if(length[vnear]+adj[vnear][i] < length[i]) :
                length[i] = length[vnear] + adj[vnear][i]
                touch[i] = vnear

        length[vnear] = -1  #vnear 의 인덱스를 음수 값으로 바꿔, 노드를 Y 에 추가한다.

    print(totalcost(start,stop))    #결과값 출력

def totalcost(start, stop) :    #결과값을 계산하는 함수 정의
    dist = 0

    # 도착점까지의 최단 경로에 존재하는 가중치를 모두 더함.
    while (stop != start) :     #출발지점으로 돌아올 때 까지 반복
        dist += adj[touch[stop]][stop]
        stop = touch[stop]

    return dist     #결과값 리턴

#main()
n = int(sys.stdin.readline())    #도시 개수 n
m = int(sys.stdin.readline())    #버스 개수 m

# <인접행렬 초기화>
#인접행렬 adj, 도시 번호가 1 부터 주어지므로 n+1 개의 배열 생성
adj= [[float('inf') for col in range(n+1)] for row in range(n+1)]
#1 번부터 n 번 도시까지, i ==j 일 경우에는 거리에 0 을 대입, 그렇지 않을 경우 무한대로
초기화
for i in range (1,n+1) :
    for j in range(1,n+1) :
        if i == j:
            adj[i][j] = 0
```

```
#인접행렬 값을 입력받아 저장함.
for i in range (1,m+1) :
    i, j, w = map(int, sys.stdin.readline().split())
    if adj[i][j] > w :
        adj[i][j] = w

#시작점 start, 도착점 stop 입력 받음.
start , stop = map(int, sys.stdin.readline().split())

#이미 선택된 노드들의 집합을 Y 라고 할 때
#touch 배열 : Y 에 속한 노드들만 거쳐서 i 번째 도시로 가는 최단 경로 상의 마지막
이음선을 만들 수 있는 인덱스를 저장
touch = [start for i in range(n + 1)]   #시작점으로 초기화

#length 배열 : Y 에 속한 노드들만 거쳐서 i 번째 도시로 가는 현재 최단경로의 길이
length = [-1 for i in range(n + 1)]
# 현재는 Y 에 시작점만 들어있으므로, 시작점과의 가중치로 초기화
for i in range(1, n + 1):
    length[i] = adj[start][i]

dijkstra(start,stop)    #다익스트라 알고리즘을 수행하여 결과 출력
```

#1753번 : 최단경로

① 알고리즘 설명

힙(우선순위 큐)을 이용하여 앞서 구현한 다익스트라 알고리즘의 성능을 개선할 수 있다.

1) 해당 인덱스의 노드(u)로부터 갈 수 있는 정점들(v)을 그래프 G에 가중치(w)와 함께 [w,v] 형태로 쌍을 이뤄 저장한다.

2) 파이썬의 최소 힙의 성질을 이용하여 가중치(w)를 기준으로 노드를 정렬하여 뽑아낼 수 있다. 이를 통해 앞에서 vnear를 하나씩 탐색했던 과정을 효율적으로 처리할 수 있게 된다. 정렬된 상태의 힙에서 출발 지점으로부터 가중치의 합이 가장 작은 정점부터 하나씩 뽑아내면서, 해당 정점을 포함할 때의 거리를 비교하여 최단거리를 result 배열에 업데이트 시킨다.

3) 최단 거리 값을 업데이트 할 때마다 힙에 정점이 추가된다.

4) 더 이상 업데이트 할 값이 없어 힙의 모든 원소가 빠져나갈 때까지 과정을 반복한다.

5) 최단거리 배열 result를 출력한다.

② 소스코드 설명

```
import sys
import heapq    #우선순위 큐(최소 힙)을 사용하기 위하여 heapq 를 import

def dijkstra(V,start,G):    #우선순위 큐를 이용하여 다익스트라 알고리즘을 수행하는
함수 정의
```

```
    result = [INF for i in range(V+1)]  # 최단 거리 계산 결과를 저장할 result 배열
    result[start] = 0   #출발 지점 주어짐 : 출발지점 -> 출발지점 까지 거리 0 으로
초기화

    q = []  # 우선순위 큐 (최소 힙 사용)
    heapq.heappush(q, [0, start])  # 정점 사이의 거리값을 기준으로 우선순위 큐 정렬

    while q:    #큐에 원소가 없을 때 까지 반복
        dis, end = heapq.heappop(q)  # 큐에서 pop, vnear 를 선택하는 작업과 동일

        for d, x in G[end]:  # end 인덱스를 이용하여 연결된 노드만 탐색
            d += dis  # 이전거리와 현재 연결된 노드의 거리를 더해서
            if d < result[x]:  # 거리비교 : 거리가 이전보다 짧으면
                result[x] = d  # 결과값으로 출력될 result 배열의 거리를 갱신시키고
                heapq.heappush(q, [d, x])  # 해당 간선과 노드를 우선순위 큐에
넣어준다.

    for i in range(1, V + 1):   #결과 출력
        # i 번 정점까지 최단 경로의 경로값을 출력한다. 시작점 자신은 0 으로 출력하고,
경로가 존재하지 않는 경우에는 INF 를 출력
        print(result[i] if result[i] != INF else "INF")


#main()
INF = sys.maxsize   #INF 변수에 최댓값 저장
V, E = map(int, sys.stdin.readline().split())   #정점의 개수 V, 간선의 개수 E

K = int(sys.stdin.readline())   #시작점 K
G = [[] for i in range(V + 1)]  # 그래프 G

for i in range(E):  # 간선 정보를 그래프에 저장
    u, v, w = map(int, sys.stdin.readline().split())
    G[u].append([w, v])

dijkstra(V,K,G)     #다익스트라 알고리즘 실행
```

1916번 방법과 1753번 방법의 비교설명 및 시간복잡도 분석

1916번의 다익스트라 알고리즘은 모든 인접 노드의 값을 선형 탐색하여 vnear을 구하고, 이를 현재 값과 비교하여 update 하는 과정을 (n-1)번 반복한다. 이는 2중 for문으로 구현되어 있으며, 바깥 for문이 총 (n-1)번, 안의 for문이 n번씩 반복되므로 입력된 정점의 개수를 n이라고 했을 때, 알고리즘의 시간복잡도가 $\mathbf{O(n^2)}$ 으로 형성됨을 알 수 있다. 이 경우 정점의 개수는 많은데 간선은 적은 그래프를 탐색할 때 비효율적일 수 있다.

1753번의 알고리즘에서는 각 정점은 정확히 한 번씩 방문되고, 따라서 모든 간선은 한 번씩 검사된다는 사실을 떠올리면 간선들을 검사하는 데는 전체 O(|E|)의 시간이 걸린다. 우선순위 큐가 가장 커지는 최악의 시나리오는 그래프의 모든 간선이 검사될 때마다 result[ ]가 갱신되고 우선순위

큐에 정점의 번호가 추가되는 것이다. 따라서 최대로 heap에 들어갈 수 있는 원소의 수는 E개이며 heap의 시간복잡도는 heap에 들어간 원소의 수에 비례하기 때문에 최종 시간복잡도는 $O(|E|\log|E|)$이다. 그래서 이 두 작업에 걸리는 시간을 더하면 $O(|E|\log|E|)$가 되지만, 대개의 경우 그래프에서 간선의 개수 |E|는 $|V|^2$보다 작기 때문에 O(log|E|)=O(log|V|)라고 볼 수 있다. 따라서, 위에서처럼 정점의 개수를 n으로 놓았을 때, 1753번의 시간복잡도는 $\mathbf{O(n\log|n|)}$을 가진다. 그러므로 힙(우선순위 큐)을 이용한 다익스트라 알고리즘이 더욱 효율적임을 알 수 있다.