09 최단 경로

01 가장 빠른 길 찾기

가장 빠르게 도달하는 방법

최단 경로 문제는 **가장 짧은 경로**를 찾는 알고리즘이며, 보통 그래프를 이용해 표현한다.

각 지점은 그래프에서 '노드'로 표현되고, 지점간 연결된 도로는 그래프에서 '간선'으로 표현된다.

최단 거리 알고리즘의 종류

- 다익스트라 최단 경로 알고리즘
- 플로이드 워셜 알고리즘
- 벨만 포드 알고리즘

다익스트라 최단 경로 알고리즘

다익스트라 최단 경로 알고리즘은 그래프에서 여러 개의 노드가 있을 때, 특정한 노드에서 출발해서 다른 노드로 가는 각각의 최단 경로를 구해주는 알고리즘이다.

매번 '**가장 비용이 적은 노드**'를 선택해서 임의의 과정을 반복하기 때문에 기본적으로 그리디 알고리즘으로 분류된다.

1. 출발 노드를 설정한다.
2. 최단 거리 테이블을 초기화한다.
3. 방문하지 않은 노드 중에서 최단 거리가 가장 짧은 노드를 선택한다.
4. 해당 노드를 거쳐 다른 노드로 가는 비용을 계산하여 최단 거리 테이블을 갱신한다.
5. 위 과정에서 3, 4번을 반복한다.

다익스트라 알고리즘을 구현하는 방법

1. 구현하기 쉽지만 느리게 동작하는 코드
2. 구현하기에 조금 더 까다롭지만 빠르게 동작하는 코드

→ **한 단계당 하나의 노드에 대한 최단 거리**를 확실히 찾을 수 있다.

따라서 마지막 노드에 대해서는 해당 노드를 거쳐 다른 노드로 가는 경우를 확인할 필요가 없다.

다익스트라 알고리즘의 특징

- **그리디 알고리즘** : **매 상황에서 방문하지 않은 가장 비용이 적은 노드**를 선택해 임의의 과정을 반복한다.
- 단계를 거치며 **한 번 처리된 노드의 최단 거리**는 고정되어 더 이상 바뀌지 않는다.
  - **한 단계당 하나의 노드에 대한 최단 거리를 확실히 찾는 것**으로 이해할 수 있다.
- 다익스트라 알고리즘을 수행한 뒤에 테이블에 각 노드까지의 최단 거리 정보가 저장된다.
  - 완벽한 형태의 최단 경로를 구하려면 소스코드에 추가적인 기능을 더 넣어야 한다.

방법 1. 간단한 다익스트라 알고리즘

다익스트라 알고리즘은 $O(N^2)$의 시간 복잡도를 가지며, 다익스트라에 의해서 처음 고안되었던 알고리즘이다.

```
import sys
input = sys.stdin.readline
INF = int(1e9)

# 노드의 개수, 간선의 개수를 입력받기
n, m = map(int, input().split())

# 시작 노드 번호를 입력받기

start = int(input())

# 각 노드에 연결되어 있는 노드에 대한 정보를 담는 리스트를 만들기
graph = [[] for i in range(n+1)]

# 방문한 적이 있는지 체크하는 목적의 리스트를 만들기
visited = [False] * (n+1)

# 최단 거리 Table을 모두 무한으로 초기화
distance = [INF] * (n+1)

# 모든 간선 정보를 입력받기
for _ in range(m):
  a,b,c = map(int,input().split())
  # a -> b 로 가는 비용이 c다.
  graph[a].append((b,c))

# 방문하지 않은 노드 중, 가장 최단 거리가 짧은 노드의 번호를 반환한다.

def get_smallest_node():
  min_value = INF
  # 가장 최단거리가 짧은 노드(인덱스)
  index = 0
  for i in range(1, n+1):
    if distance[i] < min_value and not visited[i]:
      min_value = distance[i]
      index = i
```

```
  return index

def dijkstra(start):
  # 시작 노드에 대해 초기화
  distance[start] = 0
  visited[start] = True
  for j in graph[start]:
    distance[j[0]] = j[1]
  # 시작 노드를 제외한 전체 n-1개의 노드에 대해 반복
  for i in range(n-1):
    # 현재 최단 거리가 가장 짧은 노드를 꺼내서, 방문 처리
    now = get_smallest_node()
    visited[now] = True
    # 현재 노드와 연결된 다른 노드를 확인한다.
    for j in graph[now]:
      cost = distance[now] + j[1]
      # 현재 노드를 거쳐서 다른 노드로 이동하는 거리가 더 짧은 경우
      if cost < distance[j[0]]:
        distance[j[0]] = cost

dijkstra(start)

# 모든 노드로 가기 위한 최단 거리를 출력한다.
for i in range(1,n+1):
  # 도달할 수 없는 경우, INF를 출력한다.
  if distance[i] == INF:
    print("INFINITY")
  # 도달할 수 있는 경우 거리를 출력한다.
  else:
    print(distance[i])
```

다익스트라 알고리즘 : 간단한 구현 방법 성능 분석

- 총 $O(V)$번에 걸쳐서 최단 거리가 가장 짧은 노드를 매번 선형 탐색해야 한다.
- 따라서 전체 시간 복잡도는 $O(V^2)$이다.
- 일반적으로 코딩 테스트의 최단 경로 문제에서 전체 노드의 개수가 5000개 이하라면 이 코드로 문제를 해결할 수 있다.
    - 하지만 노드의 개수가 10000개를 넘어가는 문제라면 어떻게 해야할까?

우선순위 큐(Priority Queue)

- 우선순위가 가장 높은 데이터를 가장 먼저 삭제하는 자료구조이다.
- 예를 들어 여러 개의 물건 데이터를 자료구조에 넣었다가 가치가 높은 물건 데이터부터 꺼내서 확인해야 하는 경우에 우선순위 큐를 이용할 수 있다.
- Pyhton, C++, Java를 포함한 대부분의 프로그래밍 언어에서 **표준 라이브러리 형태**로 지원한다.

자료구조	추출되는 데이터
스택(Stack)	가장 나중에 삽입된 데이터
큐(Queue)	가장 먼저 삽입된 데이터
우선순위 큐(Priority Queue)	가장 우선순위가 높은 데이터

- 우선순위 큐를 구현하기 위해 사용하는 자료구조 중 하나이다.
- 최소 힙과 최대 힙이 있다.
- 다익스트라 최단 경로 알고리즘을 포함해 다양한 알고리즘에서 사용된다.

다익스트라 알고리즘 : 개선된 구현 방법

```
import heapq
import sys
input = sys.stdin.readline
INF = int(1e9) # 무한을 의미하는 값을 10억으로 설정한다.
# 노드 및 간선의 개수를 입력받는다.
n, m = map(int,input().split())
# 시작 노드의 번호를 입력받는다.
start = int(input())
# 노드에 대한 정보를 담는 리스트 생성
graph = [[] for i in range(n+1)]
# 최단 거리 테이블을 모두 무한으로 초기화한다.
distance = [INF] * (n+1)

# 모든 간선 정보를 입력받는다.
for _ in range(m):
  a,b,c = map(int,input().split())
  # a 번 노드에서 b번 노드로 가는 비용이 c라는 의미이다.
  graph[a].append(b,c)


def dijkstra(start):
  q = []
  # 시작 노드로 가기 위한 최단 경로는 0으로 설정하여, 큐에 삽입한다. (우선순위 큐에 대한 데이터 삽입)
  heapq.heappush(q,(0,start))
  distance[start] = 0
  # 큐가 비어있지 않다면
  while q:
    # 가장 최단 거리가 짧은 노드에 대한 정보를 꺼낸다.
    dist,now = heapq.heappop(q)
    # 현재 노드가 이미 처리된 적이 있는 노드라면 무시
    if distance[now] < dist:
      continue
    # 현재 노드와 연결된 다른 인접한 노드를 확인한다.
    for i in graph[now]:
      cost = dist + i[1]
      # 현재 노드를 거쳐서, 다른 노드로 이동하는 거리가 더 짧은 경우 distance를 변경한다.
      if cost < distance[i[0]]:
        distance[i[0]] = cost
        heapq.heappush(q,cost(i[0]))
```

```
# 다익스트라 알고리즘을 수행한다.
dijkstra(start)

for i in range(1, n+1):
  if distance[i]  == INF:
    print("INFINITY")
  else:
    print(distance[i])


# 시작 노드로 가기 위한 최단 경로를 0으로 설정하기.
```

- 단계마다 방문하지 않은 노드 중에서 최단 거리가 가장 짧은 노드를 선택하기 위해 힙 자료구조를 이용한다.
- 다익스트라 알고리즘이 동작하는 기본 원리는 동일하다.
  - 현재 가장 가까운 노드를 저장해 놓기 위해서 힙 자료구조를 추가적으로 이용한다는 점이 다르다.
  - 현재의 최단 거리가 가장 짧은 노드를 선택해야 하므로 최소 힙을 사용한다.
- 힙 자료구조를 이용하는 다익스트라 알고리즘의 시간 복잡도는 $O(ElogV)$이다.
- 노드를 하나씩 꺼내 검사하는 반복문은 노드의 개수 V 이상의 횟수로는 처리되지 않는다.
  - 결과적으로 현재 우선순위 큐에서 꺼낸 노드와 연결된 다른 노드들을 확인하는 총 횟수는 최대 간선의 개수만큼 연산이 수행될 수 있다.
- 직관적으로 전체 과정은 E개의 원소를 우선순위 큐에 넣었다가 모두 빼내는 연산과 매우 유사하다.
  - 시간 복잡도를 $O(ElogE)$로 판단할 수 있다.
  - 중복 간선을 포함하지 않는 경우에 이를 $O(ElogV)$로 정리할 수 있다.
    - $O(ElogE)$ → $O(ElogV^2)$ → $O(2ElogV)$ -> $O(ElogV)$