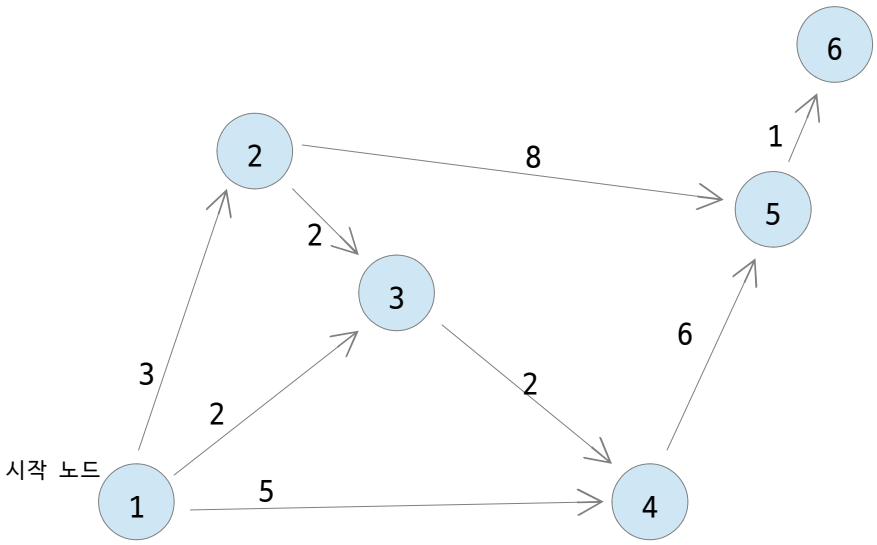
Dijkstra 최단 경로 알고리즘

sianke1991



Dijkstra 알고리즘은 하나의 시작 노드에 대해 각 노드에 도착하기 위해 필요한 최소 비용을 찾는 알고리즘이다. 위 그래프에 대해 Dijkstra 알고리즘을 적용하면 다음 결과를 얻을 수 있다. (최소 비용 테이블)

도착 노드	1	2	3	4	5	6
최소 비용	0	3	2	4	10	11

이 알고리즘은 유방향 무방향 그래프를 가리지 않고 적용할 수 있지만, 음의 가중치를 갖는 간선이 있는 그래프에는 적용할 수 없다. (cf. 벨만-포드 알고리즘)

Step a1. 최소 비용 테이블을 전부 무한대로 초기화 하고 시작 노드의 비용을 0으로 설정한다. 시작 노드 비용은 고정한다.

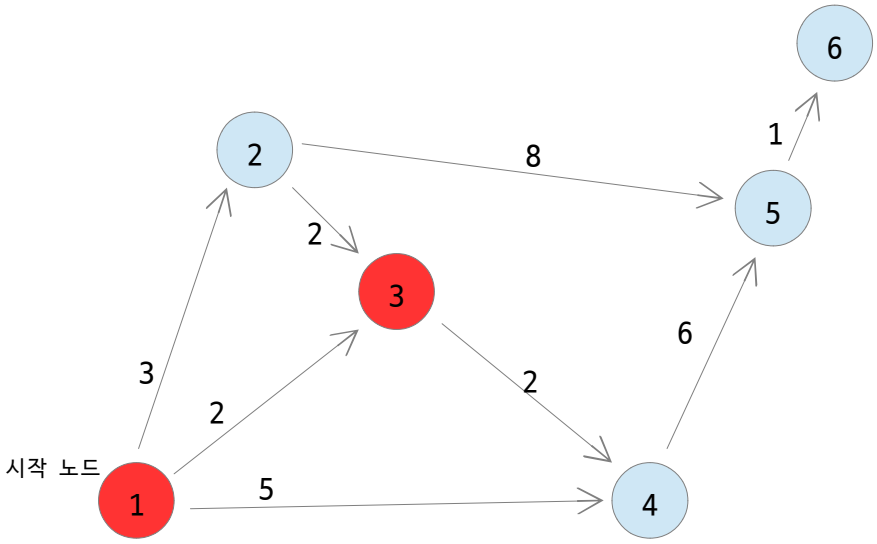
도착 노드	1	2	3	4	5	6
최소 비용	0 (고정됨)	INF	INF	INF	INF	INF

Step a2. 시작 노드인 1번 노드에서 갈 수 있는 노드의 비용 정보를 최소 비용 테이블에 넣는다. (최소 비용 테이블을 갱신한다.)

도착 노드	1	2	3	4	5	6
최소 비용	0 (고정됨)	3	2	5	INF	INF

Step a3. 최소 비용 테이블에서 고정되지 않은 값 중 최소치를 고정한다. (즉, 3번 노드로 가기 위한 최소 비용은 2로 고정된다.)

도착 노드	1	2	3	4	5	6
최소 비용	0 (고정됨)	3	2 (고정됨)	5	INF	INF



3번 노드에 도착하기 위해 필요한 최소 비용을 고정함으로써 (3번 노드에 도착하기 위해 필요한 비용이 2라고 단정할 수 있는가?)
3번 노드를 경유하여 다른 노드에 도달하는데 필요한 비용을 계산할 수 있다.

Step a4. 새로 고정된 3번 노드에서 갈 수 있는 노드의 비용을 계산하여 필요에 따라 최소 비용 테이블을 갱신한다.

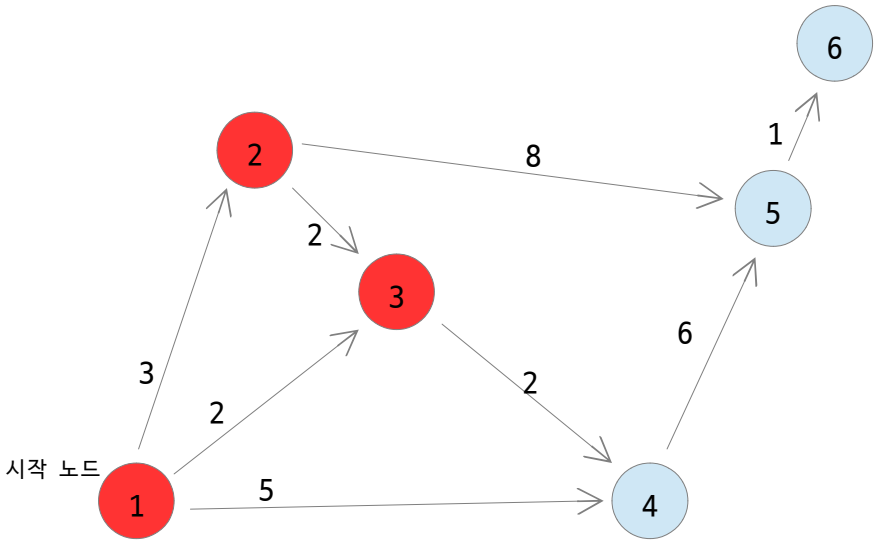
도착 노드	1	2	3	4	5	6
최소 비용	0 (고정됨)	3	2 (고정됨)	4	INF	INF

본래 4번 노드로 가기 위해 필요한 최소 비용은 5 이었지만 3번 노드를 경유하여 4번 노드로 가면 필요한 비용이 4가 되기에 최소 비용 테이블을 갱신할 수 있다. 3번 노드를 경유하여 4번 노드에 도착하기 위해 필요한 비용은 다음과 같이 구할 수 있다.

최소 비용 테이블[3] + 인접 행렬[3][4]

Step a5. 최소 비용 테이블에서 고정되지 않은 값 중 최소치를 고정한다. (즉, 2번 노드로 가기 위한 최소 비용은 3으로 고정된다.)

도착 노드	1	2	3	4	5	6
최소 비용	0 (고정됨)	3 (고정됨)	2 (고정됨)	4	INF	INF



2번 노드에 도착하기 위해 필요한 최소 비용까지 고정함으로써
2번 노드를 경유하여 5번 노드에 도착하기 위해 필요한 비용도 계산할 수 있다.

Step a6. 새로 고정된 2번 노드에서 갈 수 있는 노드의 비용을 계산하여 필요에 따라 최소 비용 테이블을 갱신한다.

도착 노드	1	2	3	4	5	6
최소 비용	0 (고정됨)	3 (고정됨)	2 (고정됨)	4	11	INF

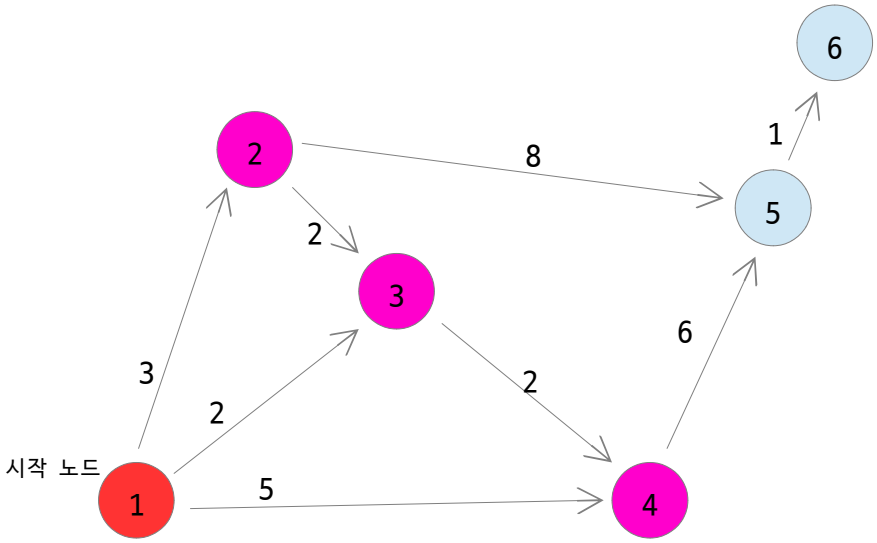
2번 노드에서 갈 수 있는 노드는 3번, 5번인데 3번 노드의 최소 비용은 고정되었으므로 5번 노드의 최소 비용만 계산한다. (설령 3번 노드의 최소 비용을 계산 하더라도 최소 비용 테이블에 기록된 값보다 작은 값은 나올 수 없을 것이다.)

Step a7. 최소 비용 테이블에서 고정되지 않은 값 중 최소치를 고정한다. (즉, 4번 노드로 가기 위한 최소 비용은 4로 고정된다.)

도착 노드	1	2	3	4	5	6
최소 비용	0 (고정됨)	3 (고정됨)	2 (고정됨)	4 (고정됨)	11	INF

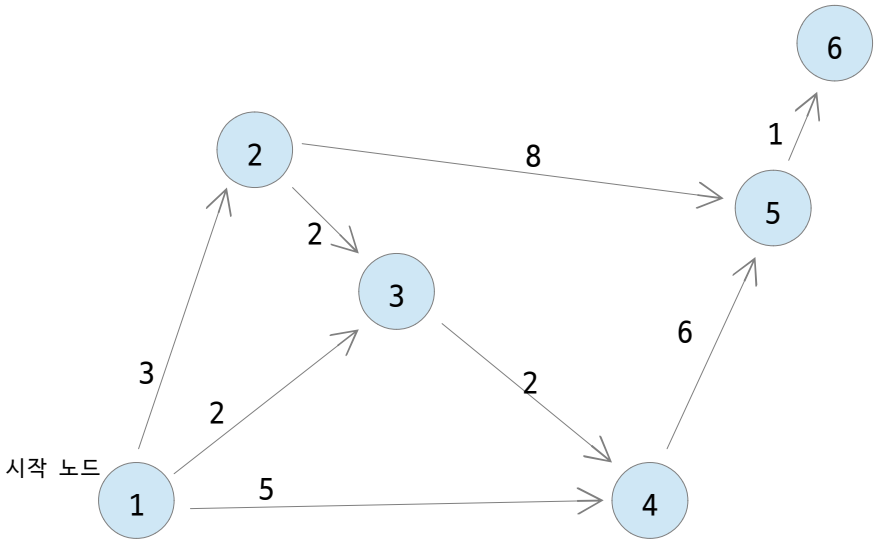
최소 비용 테이블에서 값을 고정하고, 고정된 노드에서 뻗어나가는 간선 정보를 통해 최소 비용 테이블을 갱신하는 것을 반복함으로써 최소 비용 테이블의 모든 값을 고정시키면 알고리즘이 종료된다.

도착 노드	1	2	3	4	5	6
최소 비용	0 (고정됨)	3 (고정됨)	2 (고정됨)	4 (고정됨)	10 (고정됨)	11 (고정됨)



Note 1. 시작 노드인 1번 노드에서 갈 수 있는 노드는 2, 3, 4번 노드가 있고 그 중 최소 비용으로 갈 수 있는 노드는 3번 노드이다. 이 시점에서 3번 노드 가는데 필요한 최소 비용이 2로 고정이 된다. => 1번 노드에서 뻗어 나가는 간선 중 3번 노드에 가는 것이 가중치가 가장 낮으므로. 1번 노드에서 (3번이 아닌) 다른 노드로 가는 간선은 가중치가 더 높다. 더구나 그 다른 노드에서 3번 노드로 가는 간선의 가중치도 더해져야 하므로 우회하여 3번 노드에 갈 때 필요한 비용은 2 보다 크다고 볼 수 있다. 하지만 이 설명은 음의 가중치를 갖는 간선이 있으면 적용되지 않는데, 이는 Dijkstra 알고리즘이 음의 가중치를 갖는 간선에 적용되지 않는 이유가 된다.

Note 2. 노드가 많은 그래프에서 매 단계마다 고정되지 않는 최소 비용 중 최소치를 고정시키는 이 방법을 사용하면 시간이 매우 오래 걸리게 된다. $O(V^2+E)$ 각 단계마다 노드에 갈 수 있는 비용을 계산하여 {최소 비용, 도착 노드} 페어를 컬렉션에 저장하고, 컬렉션에서 최소 비용 값이 가장 낮은 페어를 꺼내서 다음 단계를 수행할 수 있으면 알고리즘을 적용하는데 걸리는 시간을 크게 줄일 수 있다. 컬렉션에 값을 넣고 컬렉션에 들어있는 값 중 가장 낮은 값을 꺼낼 수 있는, 그런 연산을 빠르게 할 수 있는 컬렉션인 우선순위 큐를 도입해 본다.



Step b1. 최소 비용 테이블을 전부 무한대로 초기화 하고 시작 노드의 비용을 0으로 설정한다.

도착 노드	1	2	3	4	5	6
최소 비용	0	INF	INF	INF	INF	INF
PQ						

Step b2. 시작 노드인 1번 노드에서 갈 수 있는 노드의 비용 정보를 최소 비용 테이블에 넣고 우선순위 큐에 넣는다.
우선순위 큐에서 값을 꺼낼 때 비용이 가장 낮은 값을 꺼내야 한다. 따라서 우선순위 큐에 값을 넣을 때 {최소 비용, 도착 노드}의 페어를 구성하여 넣는다.

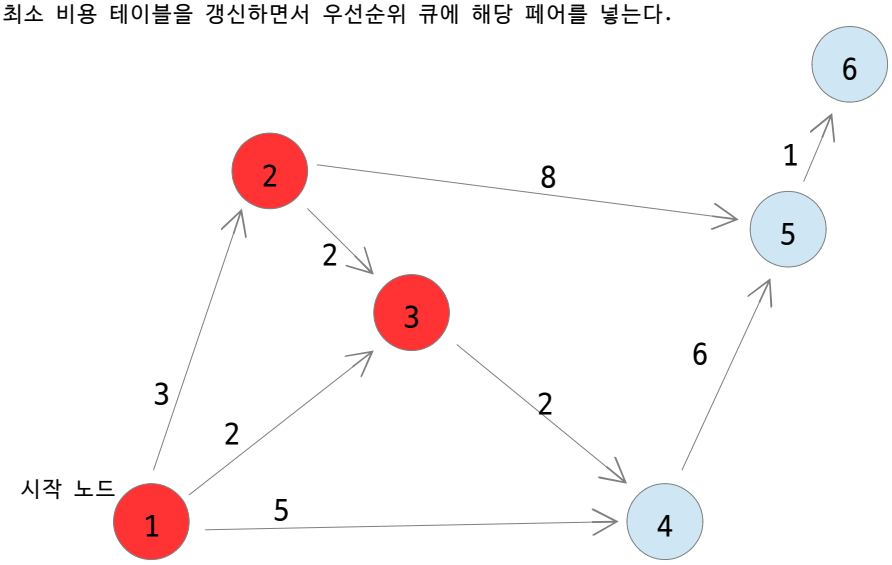
도착 노드	1	2	3	4	5	6
최소 비용	0	3	2	5	INF	INF
PQ	{2, 3}	{3, 2}	{5, 4}			

Step b3. 우선순위 큐에서 값을 꺼낸다. 꺼낸 값은 도착 노드: 3, 최소 비용: 2인 페어다. 이 정보가 최소 비용 테이블과 일치하는지 확인한다. 일치하지 않으면 이번 b3의 남은 과정은 스킵하고 다음 b3 과정으로 진행한다. (하지만 이번에는 우선순위 큐에서 꺼낸 값이 최소 비용 테이블과 일치한다.) 일치하는 경우 우선순위 큐에서 꺼낸 값의 도착 노드 (이 경우 3번 노드)에서 갈 수 있는 노드의 최소 비용을 계산하고 필요에 따라 최소 비용 테이블을 갱신하고 우선순위 큐에 넣는다.

도착 노드	1	2	3	4	5	6
최소 비용	0	3	2	4	INF	INF
PQ	{3, 2}	{4, 4}	{5, 4}			

PQ에서 꺼낸 값: {2, 3}

본래 4번 노드로 가기 위해 필요한 최소 비용은 5 이었지만 3번 노드를 경유하여 4번 노드로 가면 필요한 비용이 4가 되기에 최소 비용 테이블을 갱신할 수 있다. 최소 비용 테이블을 갱신하면서 우선순위 큐에 해당 페어를 넣는다.



Step b3. 우선순위 큐에서 값을 꺼낸다. 꺼낸 값은 도착 노드: 2, 최소 비용: 3인 페어다. 이 정보가 최소 비용 테이블과 일치하는지 확인한다. 일치하지 않으면 이번 b3의 남은 과정은 스킵하고 다음 b3 과정으로 진행한다. (하지만 이번에는 우선순위 큐에서 꺼낸 값이 최소 비용 테이블과 일치한다.) 일치하는 경우 우선순위 큐에서 꺼낸 값의 도착 노드 (이 경우 2번 노드)에서 갈 수 있는 노드의 최소 비용을 계산하고 필요에 따라 최소 비용 테이블을 갱신하고 우선순위 큐에 넣는다.

도착 노드	1	2	3	4	5	6
최소 비용	0	3	2	4	11	INF
PQ	{4, 4}	{5, 4}	{11, 5}			

PQ에서 꺼낸 값: {3, 2}

2번 노드에서 갈 수 있는 노드는 3번, 5번인데 3번 노드의 최소 비용은 갱신되지 않는다. 따라서 5번 노드의 최소 비용만 갱신하고 우선순위 큐에 넣는다.

Step b3. 우선순위 큐에서 값을 꺼낸다. 꺼낸 값은 도착 노드:4, 최소 비용: 4인 페어다. 이 정보가 최소 비용 테이블과 일치하는지 확인한다. 일치하지 않으면 이번 b3의 남은 과정은 스킵하고 다음 b3 과정으로 진행한다. (하지만 이번에는 우선순위 큐에서 꺼낸 값이 최소 비용 테이블과 일치한다.) 일치하는 경우 우선순위 큐에서 꺼낸 값의 도착 노드 (이 경우 4번 노드)에서 갈 수 있는 노드의 최소 비용을 계산하고 필요에 따라 최소 비용 테이블을 갱신하고 우선순위 큐에 넣는다.

도착 노드	1	2	3	4	5	6
최소 비용	0	3	2	4	10	INF
PQ	{5, 4}	{10, 5}	{11, 5}			

PQ에서 꺼낸 값: {4, 4}

4번 노드에서 갈 수 있는 노드는 5번이고, 4번을 경유해서 갈 때 비용인 10은 기존의 5번 노드의 최소 비용 11 보다 효율적이다. 따라서 이번 값은 최소 비용 테이블을 갱신하게 되고 우선순위 큐에도 들어가게 된다.

Step b3. 우선순위 큐에서 값을 꺼낸다. 꺼낸 값은 도착 노드: 4, 최소 비용: 5인 페어다. 이 정보가 최소 비용 테이블과 일치하는지 확인한다. 일치하지 않으면 이번 b3의 남은 과정은 스킵하고 다음 b3 과정으로 진행한다. (이번에는 우선순위 큐에서 꺼낸 값이 최소 비용 테이블과 불일치한다.)

도착 노드	1	2	3	4	5	6
최소 비용	0	3	2	4	10	INF
PQ	{10, 5}	{11, 5}				

PQ에서 꺼낸 값: {5, 4}

Step b3. 우선순위 큐에서 값을 꺼낸다. 꺼낸 값은 도착 노드: 5, 최소 비용: 10인 페어다. 이 정보가 최소 비용 테이블과 일치하는지 확인한다. 일치하지 않으면 이번 b3의 남은 과정은 스킵하고 다음 b3 과정으로 진행한다. (하지만 이번에는 우선순위 큐에서 꺼낸 값이 최소 비용 테이블과 일치한다.) 일치하는 경우 우선순위 큐에서 꺼낸 값의 도착 노드 (이 경우 5번 노드)에서 갈 수 있는 노드의 최소 비용을 계산하고 필요에 따라 최소 비용 테이블을 갱신하고 우선순위 큐에 넣는다.

도착 노드	1	2	3	4	5	6
최소 비용	0	3	2	4	10	11
PQ	{11, 5}	{11, 6}				

PQ에서 꺼낸 값: {10, 5}

Step b3. 우선순위 큐에서 값을 꺼낸다. 꺼낸 값은 도착 노드: 5, 최소 비용: 11인 페어다. 이 정보가 최소 비용 테이블과 일치하는지 확인한다. 일치하지 않으면 이번 b3의 남은 과정은 스킵하고 다음 b3 과정으로 진행한다. (이번에는 우선순위 큐에서 꺼낸 값이 최소 비용 테이블과 불일치한다.)

도착 노드	1	2	3	4	5	6
최소 비용	0	3	2	4	10	11
PQ	{11, 6}					

PQ에서 꺼낸 값: {11, 5}

Step b3. 우선순위 큐에서 값을 꺼낸다. 꺼낸 값은 도착 노드: 6, 최소 비용: 11인 페어다. 이 정보가 최소 비용 테이블과 일치하는지 확인한다. 일치하지 않으면 이번 b3의 남은 과정은 스킵하고 다음 b3 과정으로 진행한다. (하지만 이번에는 우선순위 큐에서 꺼낸 값이 최소 비용 테이블과 일치한다.) 일치하는 경우 우선순위 큐에서 꺼낸 값의 도착 노드 (이 경우 6번 노드)에서 갈 수 있는 노드의 최소 비용을 계산하고 필요에 따라 최소 비용 테이블을 갱신하고 우선순위 큐에 넣는다.

도착 노드	1	2	3	4	5	6
최소 비용	0	3	2	4	10	11
PQ						

PQ에서 꺼낸 값: {11, 6}

Step b4. 우선순위 큐가 비게 되면 해당 알고리즘이 종료된다.

Note 3. 최소 비용 테이블에서 아직 고정되지 않은 값 중 최소치를 찾는 과정을 우선순위 큐를 사용하여 깔끔하게 해결할 수 있다. 또한 적용할 그래프에 노드가 많을 수록 우선순위 큐를 도입할 때 얻는 이득이 더욱 커진다.
$O(V^2+E)$ -> $O(ElogV)$

Note 4. Dijkstra 알고리즘을 사용하여 시작 노드에서 특정 노드에 도달하기까지 경로를 얻기 위해서는 “직전 테이블”을 만든다. 직전 테이블[i]가 j일 경우 노드 i에 도달하기 위해 j -> i 간선을 사용해야 함을 의미한다.

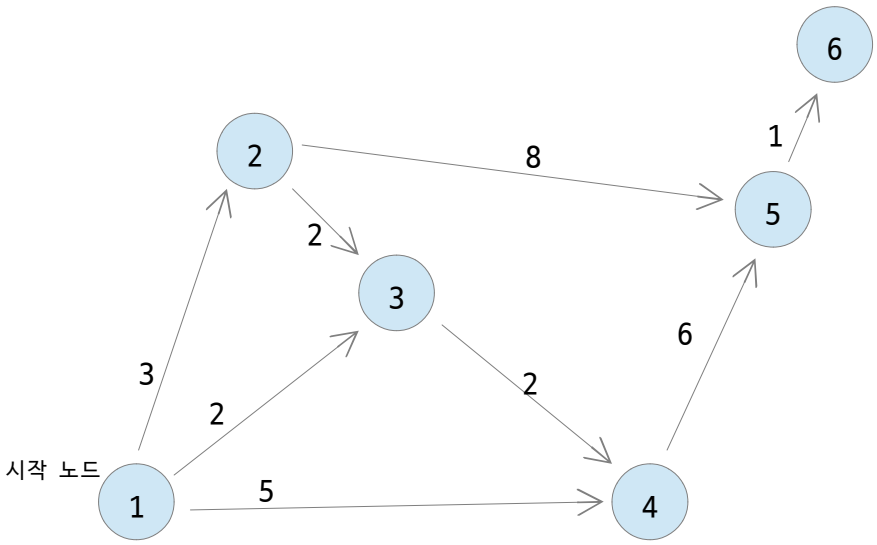
```
vector<pair<int, int>> adjList[1001]; //인접 리스트
int distTable[1001]; //거리 테이블 (비용 테이블)
int prevTable[1001]; //직전 테이블
int startingNode, endingNode; //시작 노드, 도착 노드

priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq; //{비용, 도착노드}로 구성된 pair를 사용
pq.push({0, startingNode});
while (!pq.empty()) {
    int currentDist = pq.top().first;
    int currentNode = pq.top().second;
    pq.pop();
    if (currentDist!=distTable[currentNode]) continue; //이 로직이 들어가지 않으면 시간 초과가 뜬다...

    for (pair<int, int> p:adjList[currentNode]) {
        int nextDist = currentDist+p.first;
        int nextNode = p.second;

        if (distTable[nextNode]>nextDist) { //nextNode에 도달하는데 더 효율적인 방법을 찾은 경우-
            distTable[nextNode] = nextDist;
            prevTable[nextNode] = currentNode; //nextNode에 도달하기 위해서는 currentNode -> nextNode 간선을 사용해야 한다.
            pq.push({nextDist, nextNode});
        }
    } //p loop
```

```
} //while loop
```



예를 들어 이 그래프에 대해 직전 테이블을 구성하면 다음과 같이 된다.

노드	1	2	3	4	5	6
직전 노드	-1	1	1	3	4	5

(직전 테이블[4]가 3이라는 것은, *비록 1에서 4로 바로 갈 수 있는 노드가 있음에도 불구하고,* 3 -> 4 간선을 사용해야 4번 노드에 최소의 거리로 갈 수 있음을 의미한다.)

이 테이블을 사용하여 시작 노드인 1 번 노드에서 도착 노드 6 번 노드에 도달하기까지 경로를 알아낼 수 있다. 이 경로 복원은 도착 지점에서 거슬러 올라가는 방식을 사용한다.
6 <- 5 (6 번 노드에 도달하기 위해서는 직전에 5 번 노드에 도달해야 한다.)
5 <- 4 (5 번 노드에 도달하기 위해서는 직전에 4 번 노드에 도달해야 한다.)
4 <- 3 (4 번 노드에 도달하기 위해서는 직전에 3 번 노드에 도달해야 한다.)
3 <- 1 (3 번 노드에 도달하기 위해서는 직전에 1 번 노드에 도달해야 한다. 1 번 노드는 시작 노드이므로 이상으로 반복을 종료한다.)
이상의 과정을 하나로 묶으면 다음과 같은 경로를 얻어낼 수 있다.

6 <- 5 <- 4 <- 3 <- 1

```
stack<int> nodesInPath; //시작 노드에서 출발하여 도착 노드에 도달하기까지 거쳐야 하는 노드를 역순으로 작성한 stack
//시작 노드에서 출발하여 nextNode에 도달하기 위해 거쳐야 하는 경로를 찾아 nodesInPath를 구성한다.
//이 함수를 호출하기 전 prevTable이 구성되어야 한다.
void setPath() {
    if (startingNode == endingNode) {
        nodesInPath.push(startingNode);
        return;
    }
    int currentNode = endingNode;
    do {
        nodesInPath.push(currentNode);
        currentNode = prevTable[currentNode];
    } while (currentNode!=startingNode);
    nodesInPath.push(startingNode);
}


//경로 출력 (스택에 원소를 넣고 꺼내면 그 순서가 뒤집어지므로 별도의 처리 없이 경로를 출력할 수 있다.)
while (!nodesInPath.empty()) {
    printf("%d ", nodesInPath.top());
    nodesInPath.pop();
}
```