

#4. 최단 경로 알고리즘

나정휘

<https://justicehui.github.io/>

최단 경로 알고리즘

최단 경로 알고리즘

- 최단 경로(또는 거리)를 찾는 알고리즘
- 문제 상황에 따라 여러 알고리즘을 사용할 수 있음
 - 구해야 하는 값 - Single Source Shortest Path(SSSP), All Pair Shortest Path(APSP)
 - 그래프의 형태 - 간선 방향 유무, DAG, 트리, 선인장, ...
 - 가중치의 범위 - 양수, 실수, 0/1, ...
- 가장 범용적인 알고리즘 3가지를 다룸

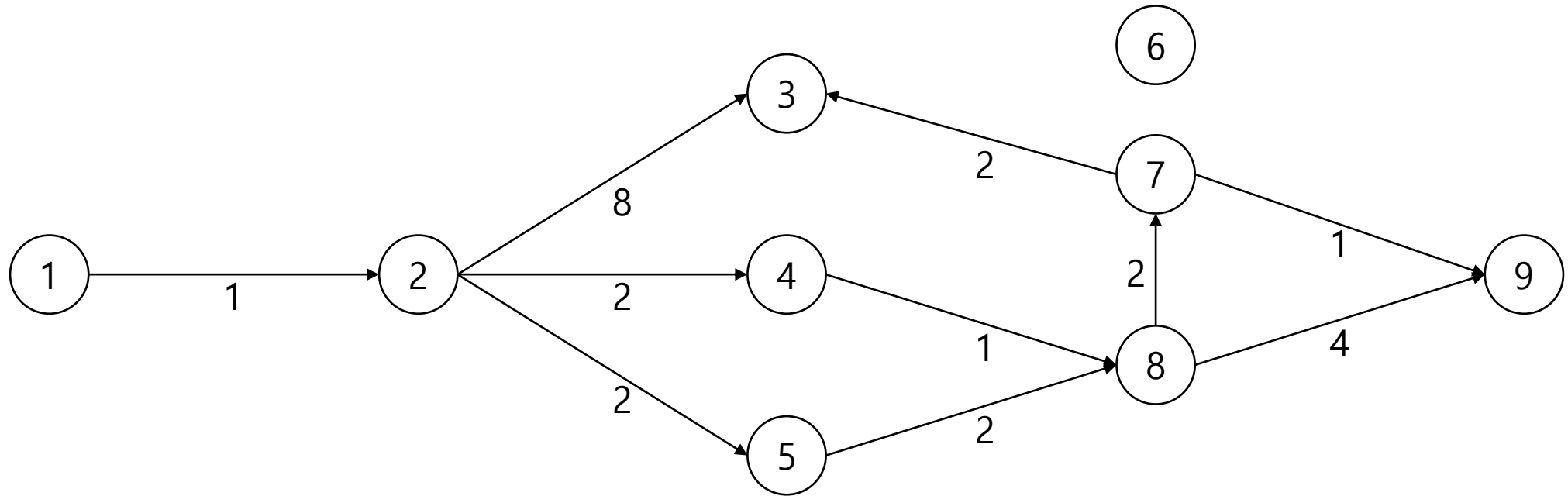
Dijkstra's Algorithm

다익스트라 알고리즘

Dijkstra's Algorithm

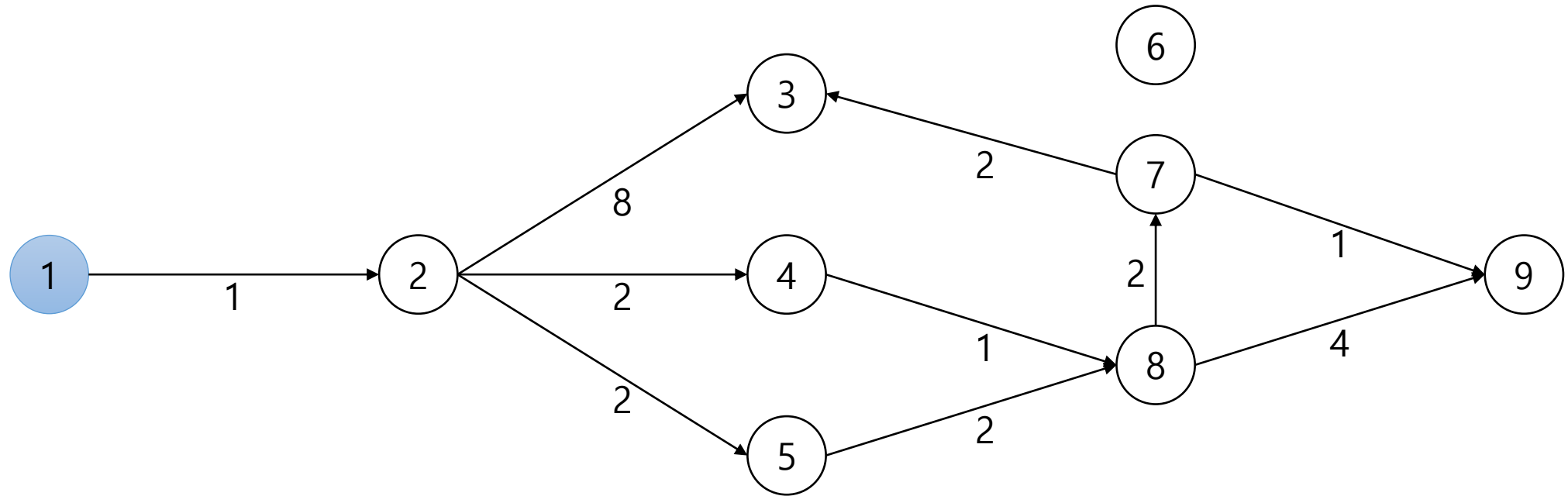
- 가중치가 0 이상인 그래프에서 SSSP를 푸는 알고리즘
- 시간 복잡도 : $O(V^2)$ / $O(E \log E)$ / $O(E + V \log V)$
- 그리디 기반 알고리즘
 1. 시작점(S)까지의 거리는 0, 다른 모든 정점까지의 거리는 INF로 초기화
 2. 아직 거리가 "확정"되지 않은 정점 중 거리가 가장 짧은 정점(v) 선택 (처음에는 S를 선택함)
 3. v의 거리를 "확정"시킴
 4. v와 인접하면서 아직 거리가 "확정"되지 않은 정점들의 거리를 갱신($D[i] \leftarrow D[v] + \text{weight}$)
 5. 모든 정점의 거리가 "확정"되었다면 종료 / 그렇지 않으면 2번으로 돌아감

다익스트라 알고리즘



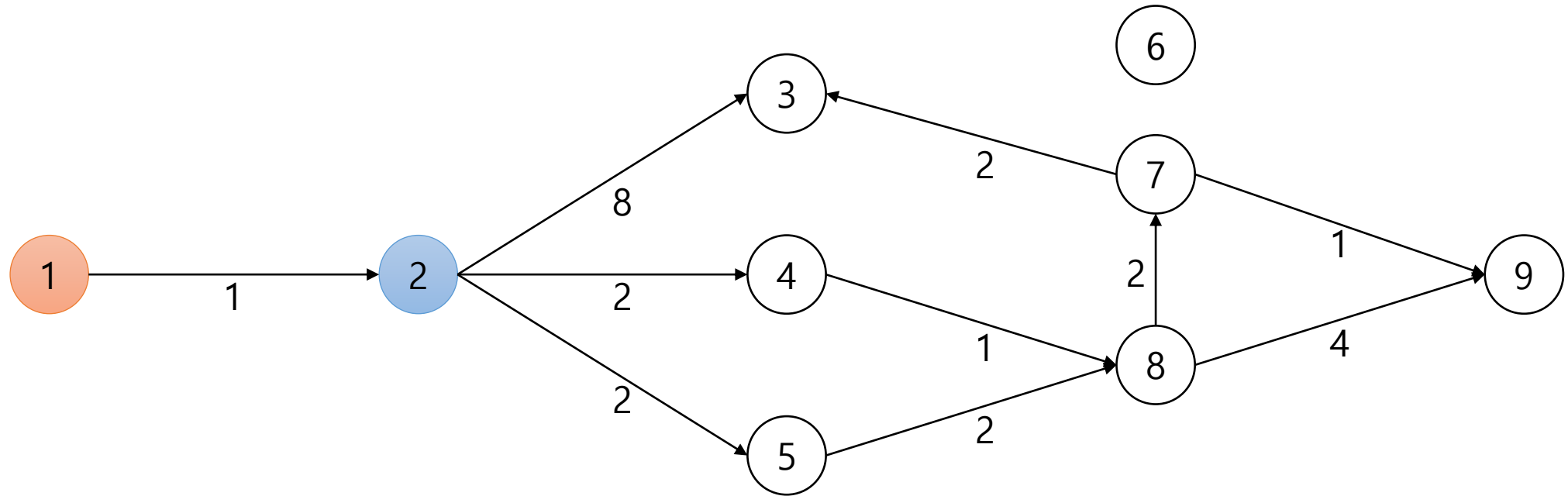
1	2	3	4	5	6	7	8	9
0	inf	inf	inf	inf	inf	inf	inf	inf

다익스트라 알고리즘



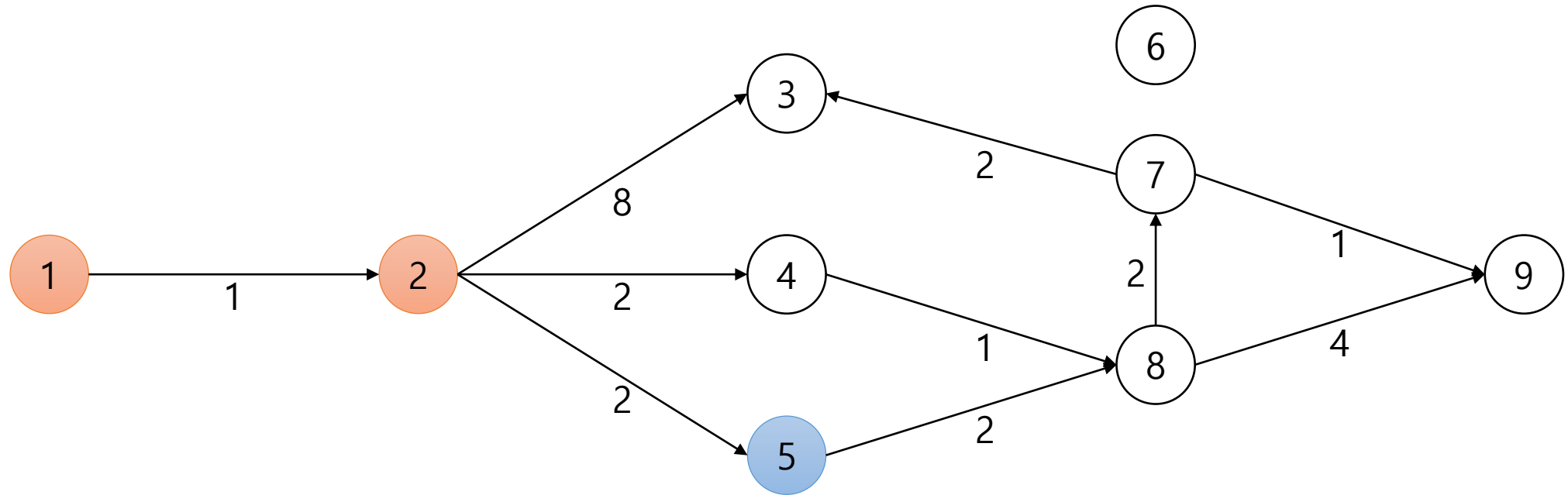
1	2	3	4	5	6	7	8	9
0	1	inf	inf	inf	inf	inf	inf	inf

다익스트라 알고리즘



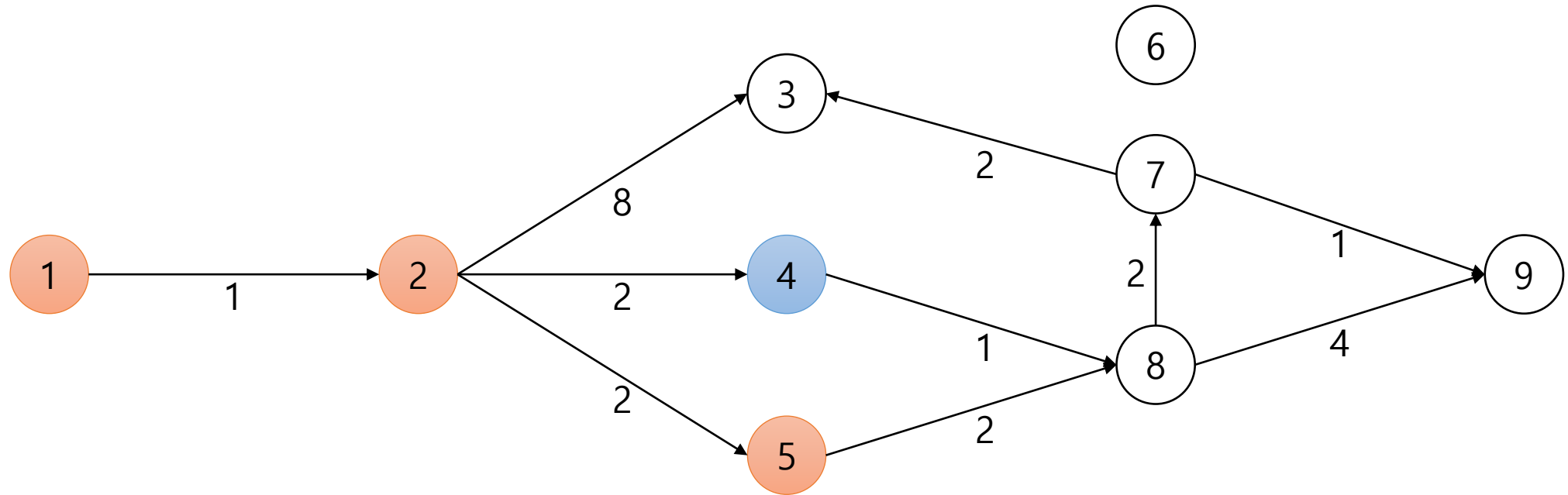
1	2	3	4	5	6	7	8	9
0	1	9	3	3	inf	inf	inf	inf

다익스트라 알고리즘



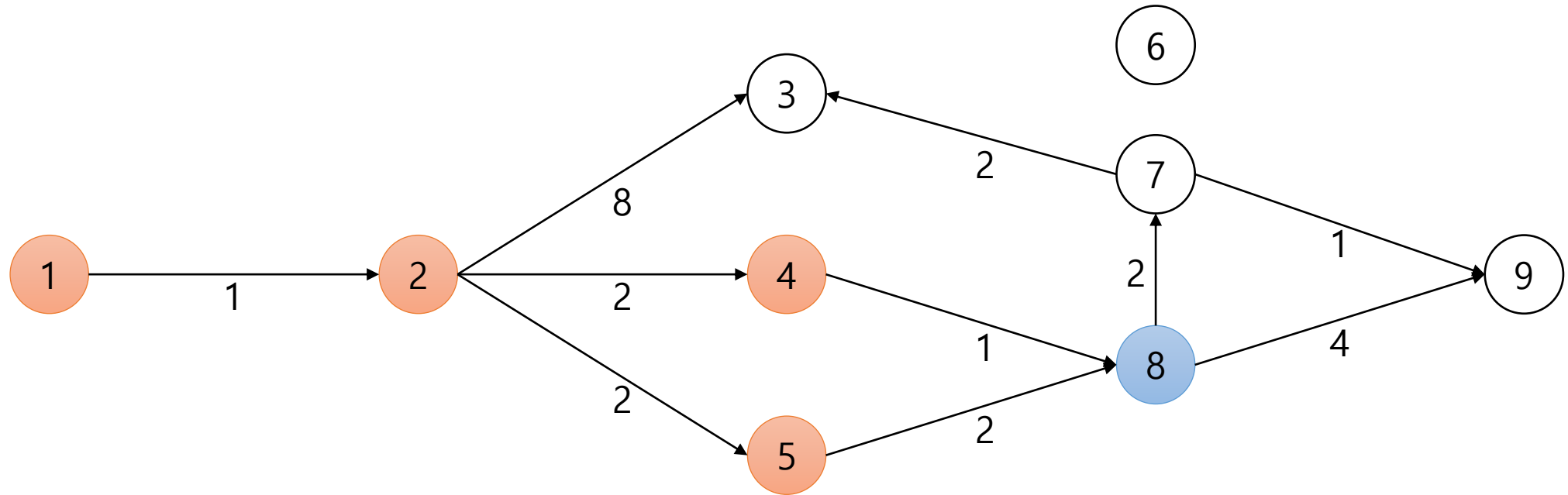
1	2	3	4	5	6	7	8	9
0	1	9	3	3	inf	inf	5	inf

다익스트라 알고리즘



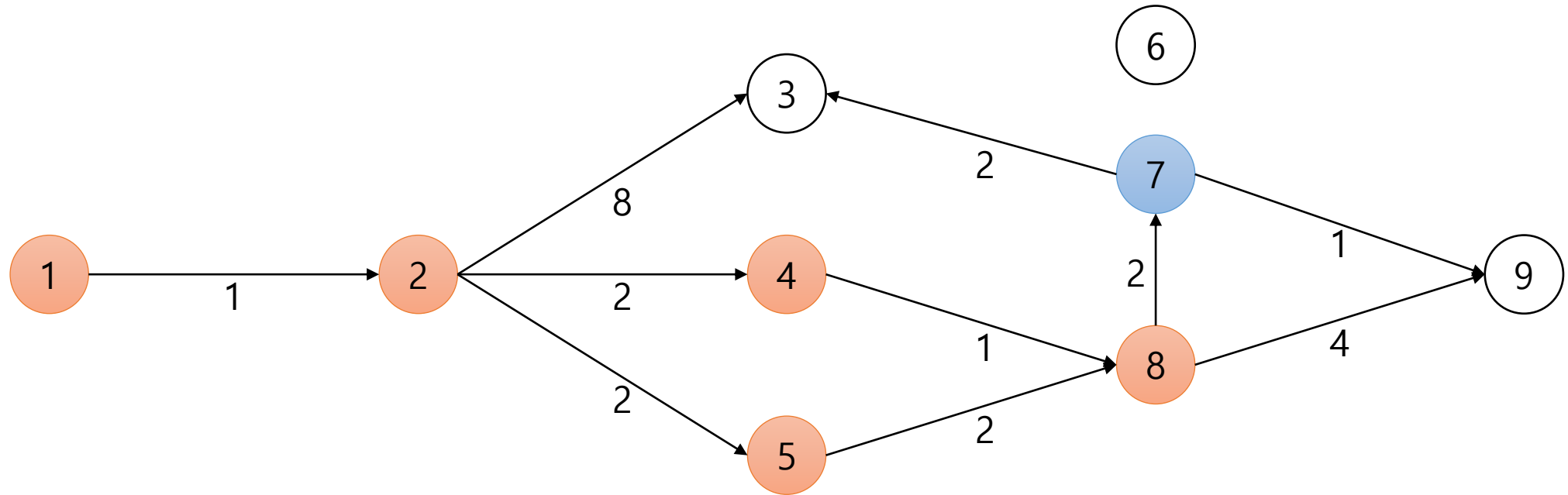
1	2	3	4	5	6	7	8	9
0	1	9	3	3	inf	inf	4	inf

다익스트라 알고리즘



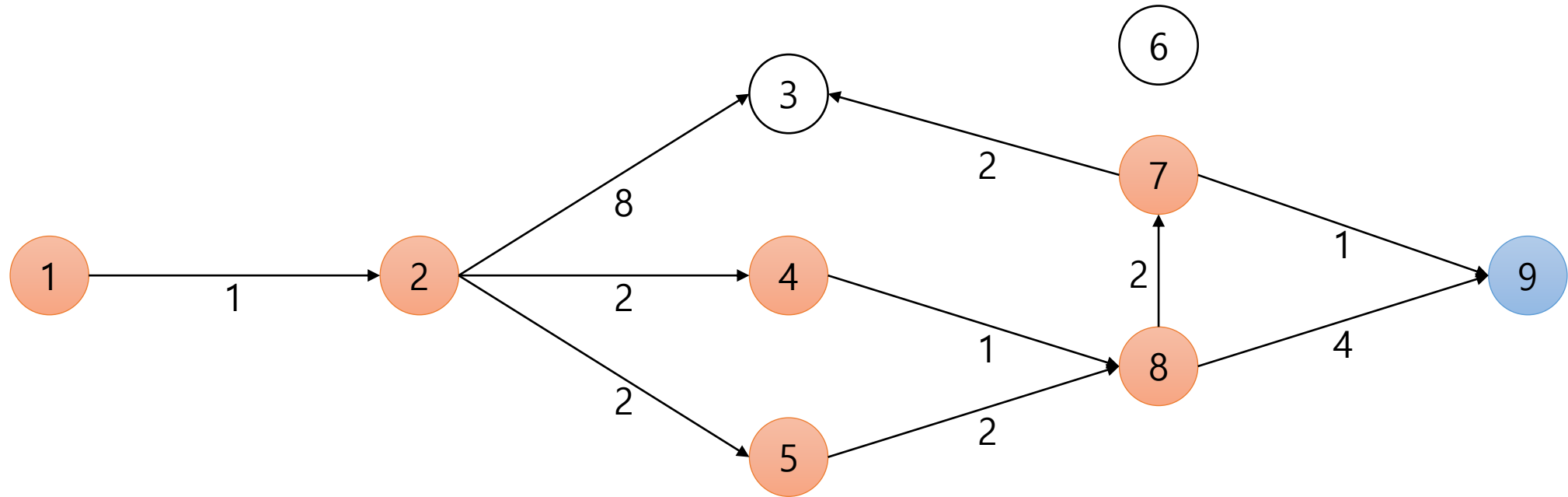
1	2	3	4	5	6	7	8	9
0	1	9	3	3	inf	6	4	8

다익스트라 알고리즘



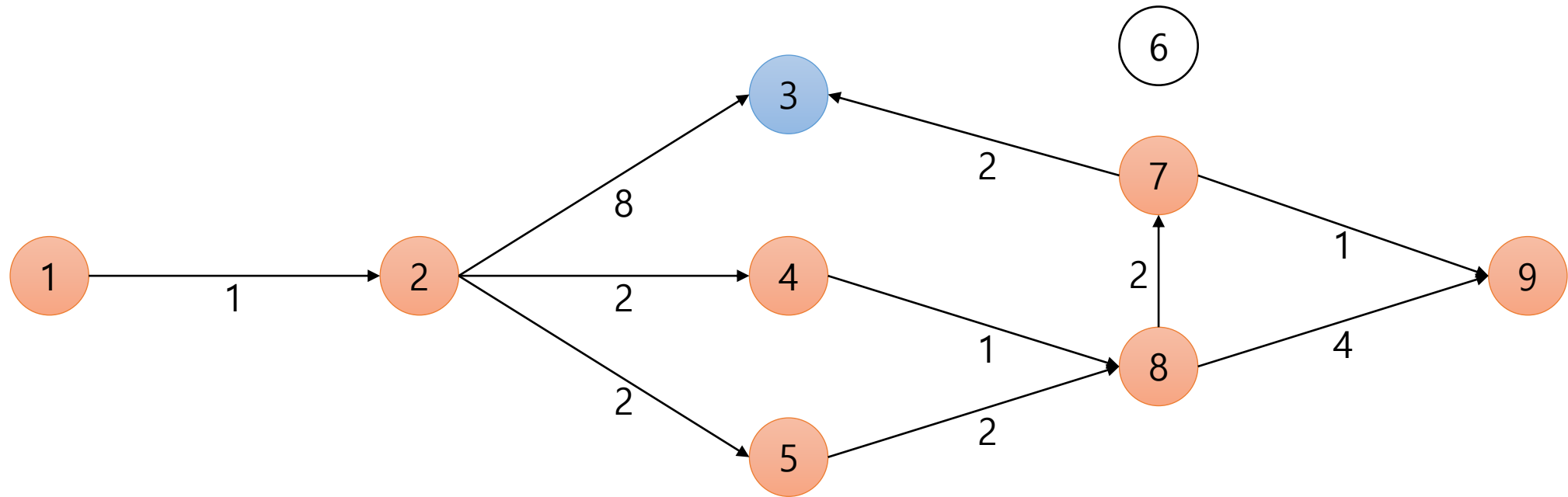
1	2	3	4	5	6	7	8	9
0	1	8	3	3	inf	6	4	7

다익스트라 알고리즘



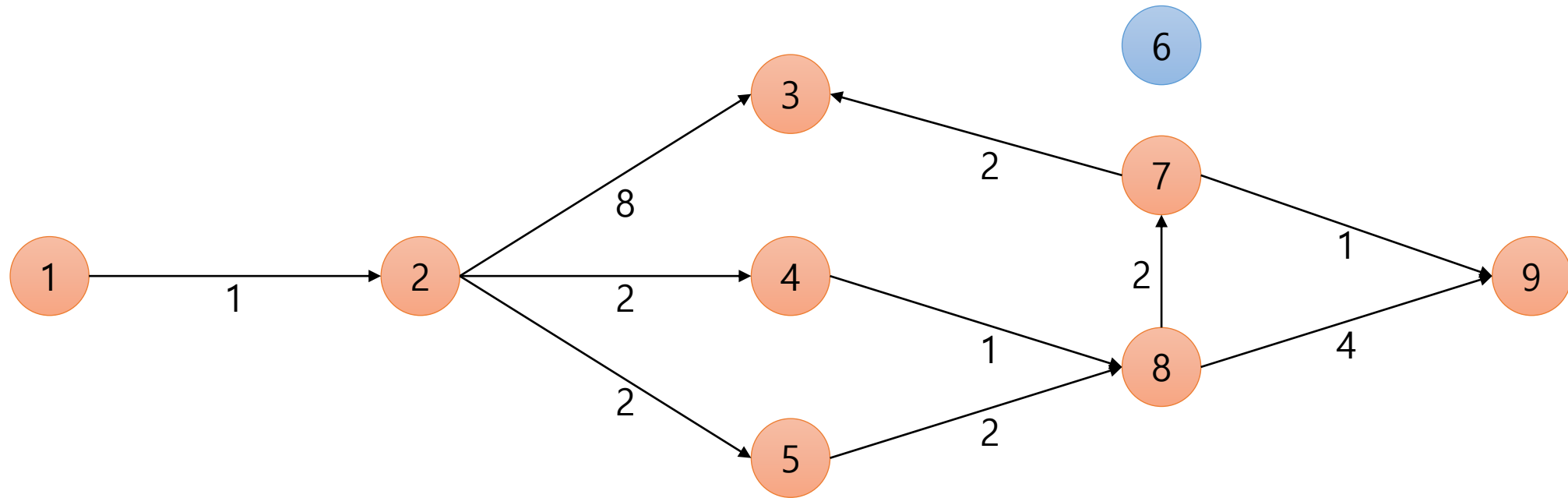
1	2	3	4	5	6	7	8	9
0	1	8	3	3	inf	6	4	7

다익스트라 알고리즘



1	2	3	4	5	6	7	8	9
0	1	8	3	3	inf	6	4	7

다익스트라 알고리즘



1	2	3	4	5	6	7	8	9
0	1	8	3	3	inf	6	4	7

다익스트라 알고리즘



```
int N, M, C[20202], D[20202], S, T;  
vector<pair<int,int>> G[20202];
```

```
void Dijkstra(){  
    for(int i=1; i<=N; i++) D[i] = 1e9;  
    D[S] = 0;  
    for(int iter=1; iter<=N; iter++){  
        int v = -1;  
        for(int i=1; i<=N; i++){  
            if(C[i]) continue;  
            if(v == -1 || D[v] > D[i]) v = i;  
        }  
        C[v] = 1;  
        for(auto [i,w] : G[v]) if(!C[i]) D[i] = min(D[i], D[v] + w);  
    }  
}
```

// 시작점까지의 거리는 0

// 이미 거리가 확정된 정점은 스킵

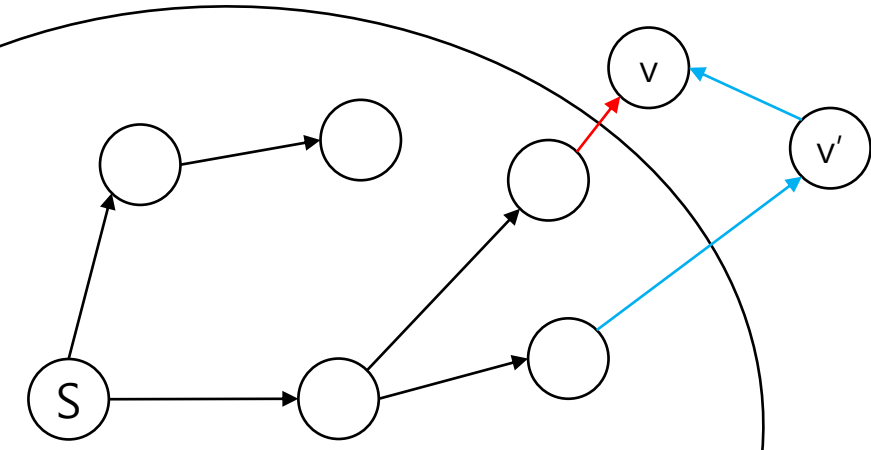
// 거리 확정

// 거리 갱신

다익스트라 알고리즘

정당성 증명

- 수학적 귀납법을 사용해 증명
 - 거리가 확정된 정점 집합을 U 라고 하자.
 - U 에 정점 v 를 추가할 때, $D[v]$ 가 v 까지의 실제 최단 거리 $sp[v]$ 와 같음을 증명
 - v 는 $V-U$ 에서 $D[v]$ 가 최소인 정점
 - 귀류법을 사용한다.
 - $D[v] > sp[v]$ 라고 가정하자.
 - s 에서 v 로 가는 경로에서 v 를 방문하기 직전에 $v' \in V-U$ 를 방문해야 함
 - $sp[v] = sp[v'] + w(v', v)$ 라는 의미이고, $w(v', v)$ 는 양수이므로 $sp[v'] < sp[v]$ 가 되어야 함
 - v 는 $V-U$ 에서 $D[v]$ 가 최소인 정점이 아니므로 모순



다익스트라 알고리즘

시간 복잡도

- V번의 iteration
 - 거리가 확정되지 않은 정점 중 가장 가까운 정점 찾기 : $O(V)$
 - v에서 갈 수 있는 정점들의 거리 갱신 : $O(\deg(V))$
- $O(\sum(V + \deg(i))) = O(V^2 + E) = O(V^2)$
 - Handshaking Lemma : $\sum(\deg(i)) = 2E$
- v에서 뺀어 나가는 간선은 모두 봐야 하므로 $O(\deg(v))$ 가 하한임
- 거리가 최소인 정점을 빠르게 찾을 수 있을까?
 - Min Heap

다익스트라 알고리즘



```
int N, M, C[20202], D[20202], S, T;
vector<pair<int,int>> G[20202];

void Dijkstra(){
    for(int i=1; i<=N; i++) D[i] = 1e9;
    priority_queue<pair<int,int>, vector<pair<int,int>>, greater<>> Q;
    D[S] = 0; Q.emplace(0, S);           // 시작점까지의 거리 0으로 초기화, {0, S}를 힙에 추가
    while(!Q.empty()){
        auto [c,v] = Q.top(); Q.pop();
        if(C[v]) continue; C[v] = 1;    // 아직 거리가 확정되지 않았다면 거리 확정
        for(auto [i,w] : G[v]){          // {다음 정점, 간선 가중치}
            if(D[i] > D[v] + w){          // 거리 갱신
                D[i] = D[v] + w;
                Q.emplace(D[i], i);
            }
        }
    }
}
```

다익스트라 알고리즘

시간 복잡도

- 각 간선을 한 번씩 보기 때문에 거리 갱신은 최대 $O(E)$ 번 발생
- Heap에 원소 $O(E)$ 번 삽입
- Heap의 크기는 최대 $O(E)$ 이므로 시간 복잡도는 $O(E \log E)$

참고

- 거리 배열은 $V * (\text{간선 가중치 최대값})$ 으로 초기화 : 모든 경로는 최대 $V-1$ 개의 간선으로 구성
- 각 정점마다 Heap에 원소가 최대 한 개 존재하도록 구현하면 $O(E \log V)$
- Heap의 decrease key 연산을 $O(1)$ 에 구현하면 $O(E + V \log V)$ 도 가능 (Fibonacci Heap/Thin Heap)
 - GCC 확장의 `__gnu_pbds::priority_queue`는 `pq.modify(iterator, value)`를 이용해 decrease key 가능
 - C++ 표준의 `std::priority_queue`는 decrease key 연산을 지원하지 않음

다익스트라 알고리즘 - $O(E + V \log V)$



```
void Dijkstra(int S){
    thin_heap pq;
    for(int i=1; i<=N; i++) D[i] = 1e9;
    D[S] = 0;

    // iter[i] = {i까지의 거리, i}를 저장하고 있는 노드에 이터레이터
    vector<heap_iter> iter(N+1);
    for(int i=1; i<=N; i++) iter[i] = pq.push({D[i], i});

    while(!pq.empty()){
        auto [c,v] = pq.top(); pq.pop();
        for(auto [i,w] : G[v]){
            // 거리 갱신 -> iter[i]가 가리키는 노드 수정
            // decrease key 연산은 amortized O(1)에 동작함
            if(D[i] > D[v] + w){
                D[i] = D[v] + w;
                pq.modify(iter[i], {D[i], i});
            }
        }
    }
}
```

다익스트라 알고리즘

응용

- 정점에 가중치가 있는 경우
 - 정점을 2개로 분할 : $in(v), out(v)$
 - u 에서 v 로 가는 간선 : $out(u)$ 에서 $in(v)$ 로 가는 간선
 - 정점 가중치 $w(v)$: $in(v)$ 에서 $out(v)$ 로 가는 가중치 $w(v)$ 간선
- $\{S_1, S_2, \dots, S_k\}$ 중 원하는 곳에서 시작해서 다른 모든 정점으로 가는 최단 거리
 - Multi Source Shortest Path
 - 새로운 정점 S_0 에서 $S_1, S_2, \dots, S_k$ 로 가는 가중치 0 간선 만들면
 - S_0 에서 다른 모든 정점으로 가는 최단 거리 문제로 바뀜

다익스트라 알고리즘

BOJ 11779 최소비용 구하기 2

- S에서 T로 가는 최소 비용과 그 경로를 구하는 문제
- $P[i]$ = S에서 i로 가는 최단 경로에서 i 바로 직전에 방문하는 정점



```
void Dijkstra(){
    for(int i=1; i<=N; i++) D[i] = 1e9;
    priority_queue<pair<int,int>, vector<pair<int,int>>, greater<>> Q;
    D[S] = 0; Q.emplace(0, S);
    while(!Q.empty()){
        auto [c,v] = Q.top(); Q.pop();
        if(C[v]) continue; C[v] = 1;
        for(auto [i,w] : G[v]){
            if(D[i] > D[v] + w){
                D[i] = D[v] + w; P[i] = v;
                Q.emplace(D[i], i);
            }
        }
    }
    vector<int> V;
    for(int i=T; i; i=P[i]) V.push_back(i);
    reverse(V.begin(), V.end());
    cout << D[T] << "\n" << V.size() << "\n";
    for(auto i : V) cout << i << " ";
}
```

다익스트라 알고리즘 - 예시

BOJ 16118 달빛 여우

- N개의 정점과 M개의 간선으로 구성된 무향 가중치 그래프
- 여우는 간선을 따라 이동할 때마다 간선의 가중치 만큼의 비용이 필요함
- 늑대는 홀수 번째로 방문하는 간선은 절반, 짝수 번째로 방문하는 간선은 2배 만큼의 비용이 필요함
- 여우와 늑대가 1번 정점에서 출발할 때, 여우가 늑대보다 먼저 도착할 수 있는 정점의 개수를 구하는 문제

- 여우의 이동 거리는 다익스트라 알고리즘을 이용해 구할 수 있음
- 늑대의 이동 거리는?

다익스트라 알고리즘 - 예시

BOJ 16118 달빛 여우

- 여우의 이동 거리는 다익스트라 알고리즘을 이용해 구할 수 있음
- 늑대의 이동 거리는?
- 홀수 번째 이동과 짝수 번째 이동을 구분할 수 있도록 각 정점을 2개로 분할
 - i 번 정점: 지금까지 간선 짝수 개 방문함
 - $N+i$ 번 정점: 지금까지 간선 홀수 개 방문함
- u 에서 v 로 가는 가중치 w 간선
 - u 에서 $N+v$ 로 가는 가중치 $w/2$ 간선
 - $N+u$ 에서 v 로 가는 가중치 $w*2$ 간선
- 실수 범위는 다루기 귀찮으니까 w 입력받을 때 2 곱하는 거 추천

다익스트라 알고리즘 - 예시

BOJ 16118 달빛 여우



```
int N, M, D1[8080], D2[8080], R;
vector<pair<int,int>> G1[8080], G2[8080];

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N >> M;
    for(int i=1; i<=M; i++){
        int u, v, w; cin >> u >> v >> w;
        G1[u].emplace_back(v, w*2);
        G1[v].emplace_back(u, w*2);
        G2[u].emplace_back(N+v, w);
        G2[v].emplace_back(N+u, w);
        G2[N+u].emplace_back(v, w*4);
        G2[N+v].emplace_back(u, w*4);
    }
    Dijkstra(G1, D1);
    Dijkstra(G2, D2);
    for(int i=1; i<=N; i++) R += D1[i] < min(D2[i], D2[N+i]);
    cout << R;
}
```

Floyd-Warshall Algorithm

플로이드 와샬 알고리즘

Floyd-Warshall Algorithm

- APSP를 푸는 알고리즘
- 시간 복잡도 : $O(V^3)$
- 공간 복잡도 : $O(V^3) \rightarrow O(V^2)$
- DP 기반 알고리즘
 - $D[k][u][v]$: u 에서 출발해서 1.. k 번 정점을 경유한 뒤 v 번 정점으로 가는 최단 거리
 - $D[0][u][u]$: 0으로 초기화
 - $u \neq v$ 일 때 $D[0][u][v]$ 는 간선이 있으면 간선의 가중치, 없으면 INF로 초기화
 - $D[k][u][v] \leftarrow D[k-1][u][k] + D[k-1][k][v]$
 - 배열 D 의 크기는 V^3

플로이드 와샬 알고리즘

Floyd-Warshall Algorithm

- 배열 D의 크기를 $2V^2$ 로 줄일 수 있음
 - $D[k][u][v]$ 를 계산할 때 $D[k-1][*][*]$ 만 사용하므로
 - $D[2][V][V]$ 크기로 만들고 토글링하면 됨
- 배열 D의 크기를 V^2 로 줄일 수 있음
 - 최솟값을 구하는 문제인데 값이 매번 감소하기 때문에 그냥 덮어써도 됨
 - 시간 복잡도는 여전히 $O(V^3)$

플로이드 와샬 알고리즘



```
int N, M, D[111][111];

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N >> M;
    for(int i=1; i<=N; i++) for(int j=1; j<=N; j++) D[i][j] = 1e9; // inf로 초기화
    for(int i=1; i<=N; i++) D[i][i] = 0; // i -> i 비용은 0
    for(int i=1; i<=M; i++){
        int s, e, w; cin >> s >> e >> w;
        D[s][e] = min(D[s][e], w);
    }
    // 여기까지 오면, D[i][j]는 다른 정점을 경유하지 않고 i에서 j로 직접 가는 최단 거리
    for(int k=1; k<=N; k++){
        for(int i=1; i<=N; i++){
            for(int j=1; j<=N; j++){
                D[i][j] = min(D[i][j], D[i][k] + D[k][j]);
            }
        }
        // D[i][j]: 1..k번 정점을 경유해서 i에서 j로 가는 최단 거리
    }
    for(int i=1; i<=N; i++) for(int j=1; j<=N; j++) cout << D[i][j] << " \n"[j==N];
}
```

플로이드 와샬 알고리즘

BOJ 11780 플로이드 2

- 경로가 존재하는 모든 (i, j) 에 대해, i 에서 j 로 가는 최단 거리와 그 경로를 구하는 문제
- $P[i][j]$ = i 에서 j 로 가는 최단 경로에서 경유하는 가장 큰 정점 번호
 - 만약 i 에서 바로 j 로 가는 경로가 최소 비용이라면 $P[i][j] = 0$
- $i \rightarrow j$ 최단 경로를 구하는 것은 $i \rightarrow P[i][j]$ 의 최단 경로, $P[i][j] \rightarrow j$ 의 최단 경로를 구하는 것으로 분할 가능
 - $\text{Recursive}(s, e)$ = s 에서 e 로 갈 때 경유하는 모든 정점을 구하는 함수

```
void Recursion(vector<int> &v, int s, int e){
    if(P[s][e] == 0) return;
    Recursion(v, s, P[s][e]);
    v.push_back(P[s][e]);
    Recursion(v, P[s][e], e);
}

vector<int> GetPath(int s, int e){
    vector<int> path;
    path.push_back(s);
    Recursion(path, s, e);
    path.push_back(e);
    return path;
}
```

플로이드 와샬 알고리즘 - 예시

BOJ 23258 밤편지

- N개의 정점으로 구성된 무향 가중치 그래프가 주어짐
- 출발 정점 S와 도착 정점 T가 주어지면, 반딧불은 S에서 T까지 최단 경로로 이동함
- x번 정점에는 2x 만큼의 이슬이 있고, 반딧불은 S와 T를 제외하고 방문하는 모든 정점에서 이슬을 마심
- 각 반딧불은 상수 C를 갖고 있어서, 이슬을 2C 방울 이상 마시면 더 이상 이동할 수 없음
- S, T, C가 주어지면 S에서 T로 가는 최소 시간을 구하는 쿼리를 Q번 처리하는 문제
- 사실 경유하는 정점 번호의 최댓값만 신경 써도 됨
 - $2^1 + 2^2 + \dots + 2^{x-1} < 2^x$ 이기 때문
 - $D[k][i][j]$ = k번 이하의 정점만 경유해서 i에서 j로 가는 최단 거리
- 플로이드 와샬 알고리즘을 이용해서 해결할 수 있음
 - 시간 복잡도, 공간 복잡도 모두 $O(N^3)$

플로이드 와샬 알고리즘 - 예시

BOJ 23258 밤편지

```
● ● ●

#include <bits/stdc++.h>
using namespace std;
constexpr int INF = 0x3f3f3f3f;

int N, Q, D[333][333][333];

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N >> Q;
    for(int i=1; i<=N; i++){
        for(int j=1; j<=N; j++){
            cin >> D[0][i][j];
            if(D[0][i][j] == 0 && i != j) D[0][i][j] = INF;
        }
    }
    for(int k=1; k<=N; k++){
        for(int i=1; i<=N; i++){
            for(int j=1; j<=N; j++){
                D[k][i][j] = min(D[k-1][i][j], D[k-1][i][k] + D[k-1][k][j]);
            }
        }
    }
    while(Q--){
        int c, s, e; cin >> c >> s >> e;
        if(D[c-1][s][e] == INF) cout << -1 << "\n";
        else cout << D[c-1][s][e] << "\n";
    }
}
```

Bellman-Ford Algorithm

벨만 포드 알고리즘

Bellman-Ford Algorithm

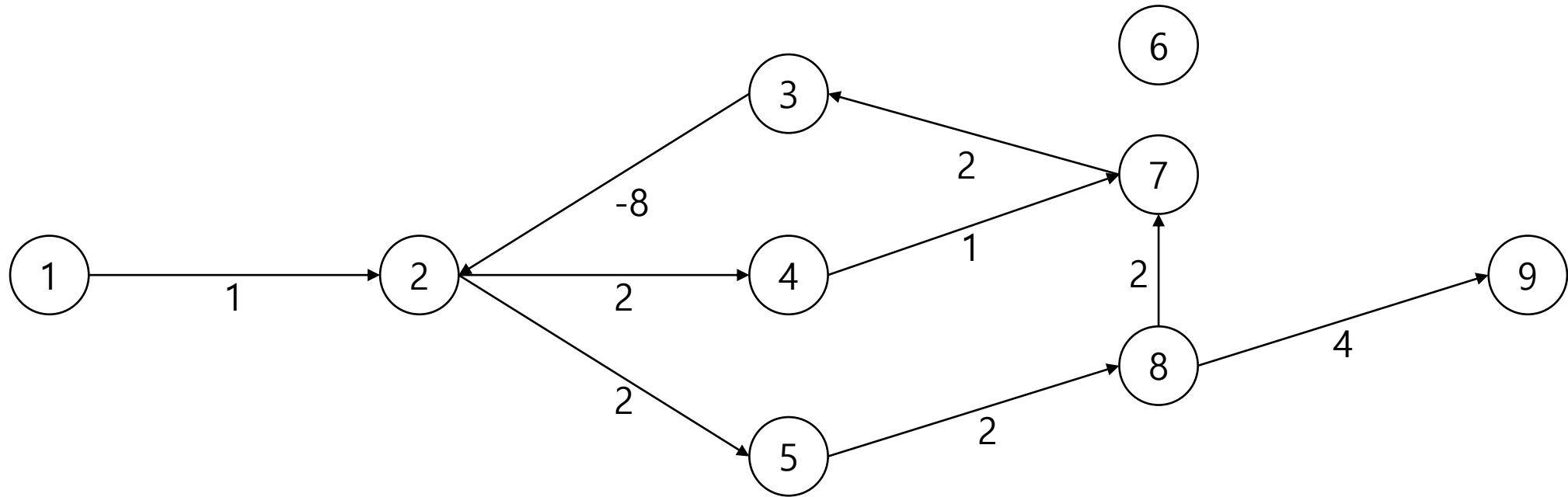
- SSSP를 푸는 알고리즘
- 시간 복잡도 : $O(VE)$
- 몇 가지 관찰을 해보자.
 - 관찰 1) 가중치의 합이 음수인 사이클이 있으면 최단 거리를 구할 수 없음 (음의 무한대로 발산)
 - 관찰 2) 음수 사이클을 지나지 않는다면 모든 최단 경로는 최대 $V-1$ 개의 간선을 지남
- Dijkstra's Algorithm처럼 relaxation을 통해 최단 거리를 구함
 - 시작 정점 S 의 거리는 0, 다른 모든 정점까지의 거리는 INF로 초기화
 - 모든 간선 (s, e, w) 에 대해 $D[e] = \min(D[e], D[s] + w)$ 를 수행하는 것을 $V-1$ 번 반복
 - i 번째 iteration이 끝나면 간선 i 개를 거쳐서 가는 최단 거리를 정확하게 구할 수 있음
 - 귀납법으로 증명 가능

벨만 포드 알고리즘

Bellman-Ford Algorithm

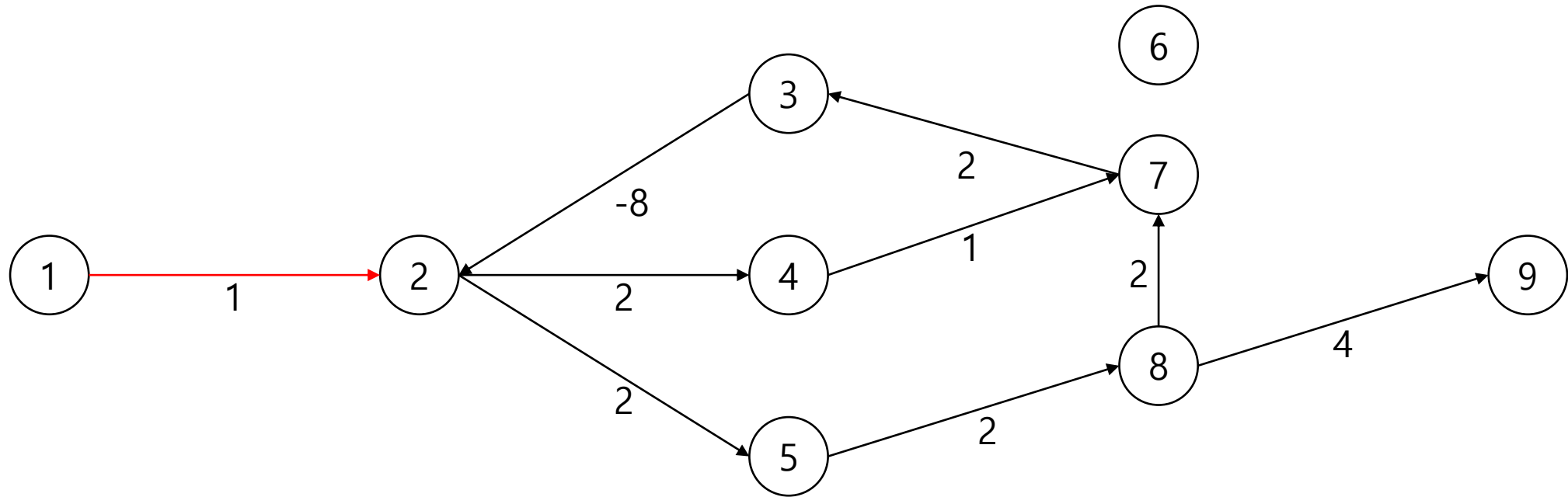
- 음수 사이클이 존재하는지 판별하는 것도 가능
- 음수 사이클이 없다면 $V-1$ 번의 iteration으로 항상 최단 거리를 구할 수 있음
- 만약 V 번째 iteration에서 relaxation이 발생하면? 음수 사이클 존재

벨만 포드 알고리즘



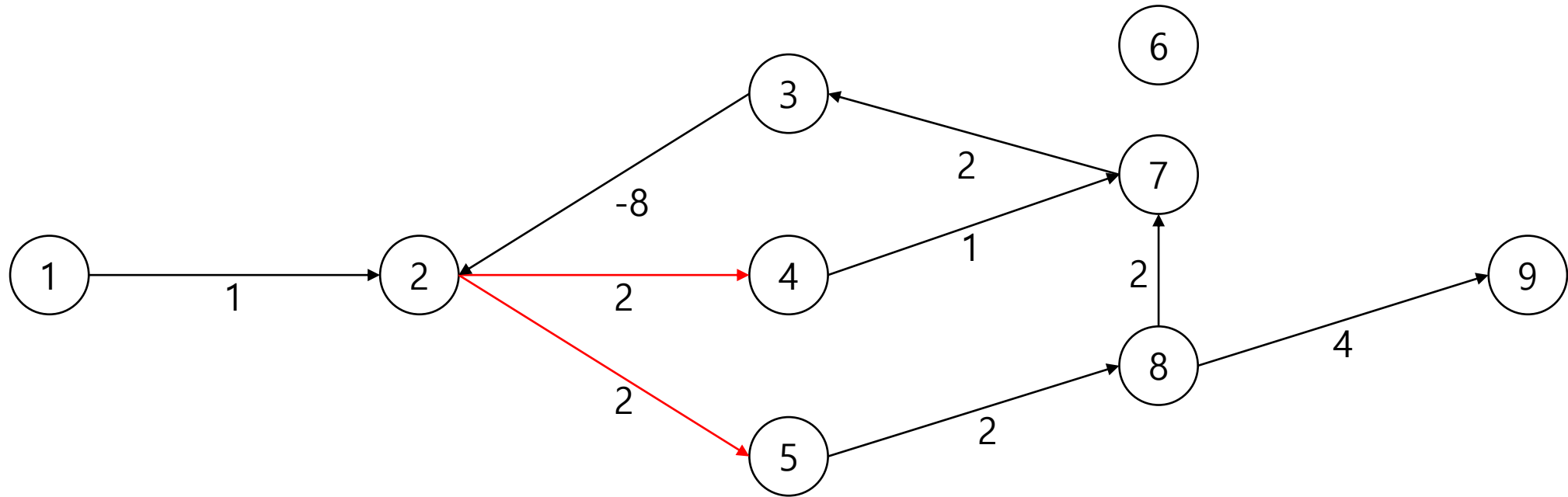
1	2	3	4	5	6	7	8	9
0	inf	inf	inf	inf	inf	inf	inf	inf

벨만 포드 알고리즘 - 1



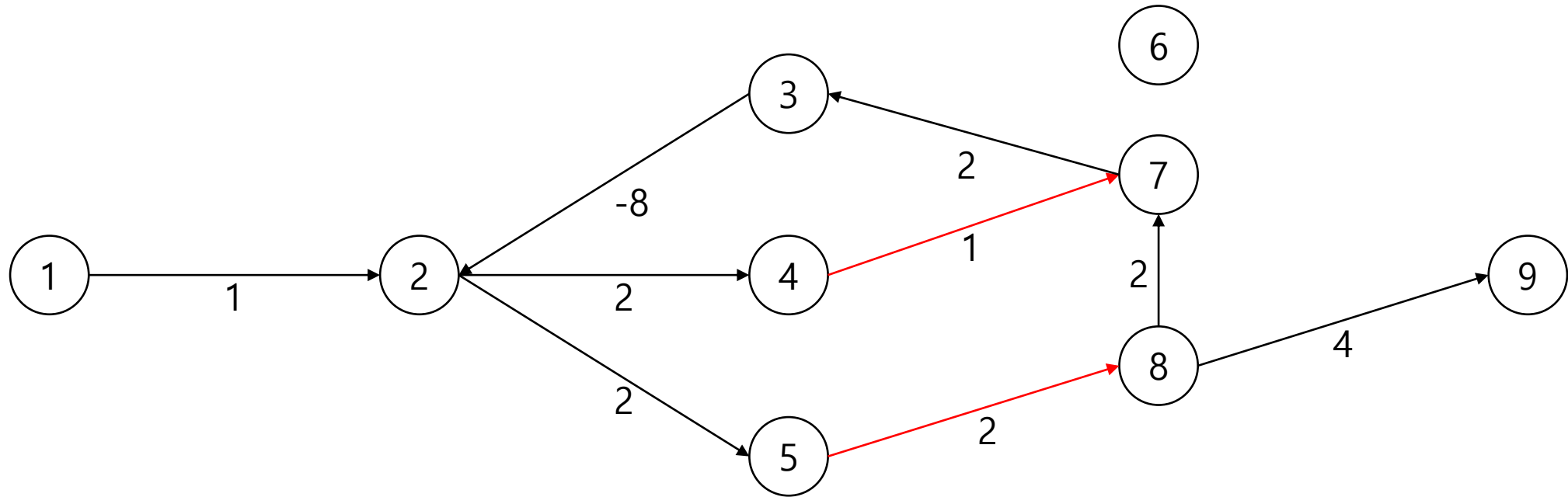
1	2	3	4	5	6	7	8	9
0	1	inf	inf	inf	inf	inf	inf	inf

벨만 포드 알고리즘 - 2



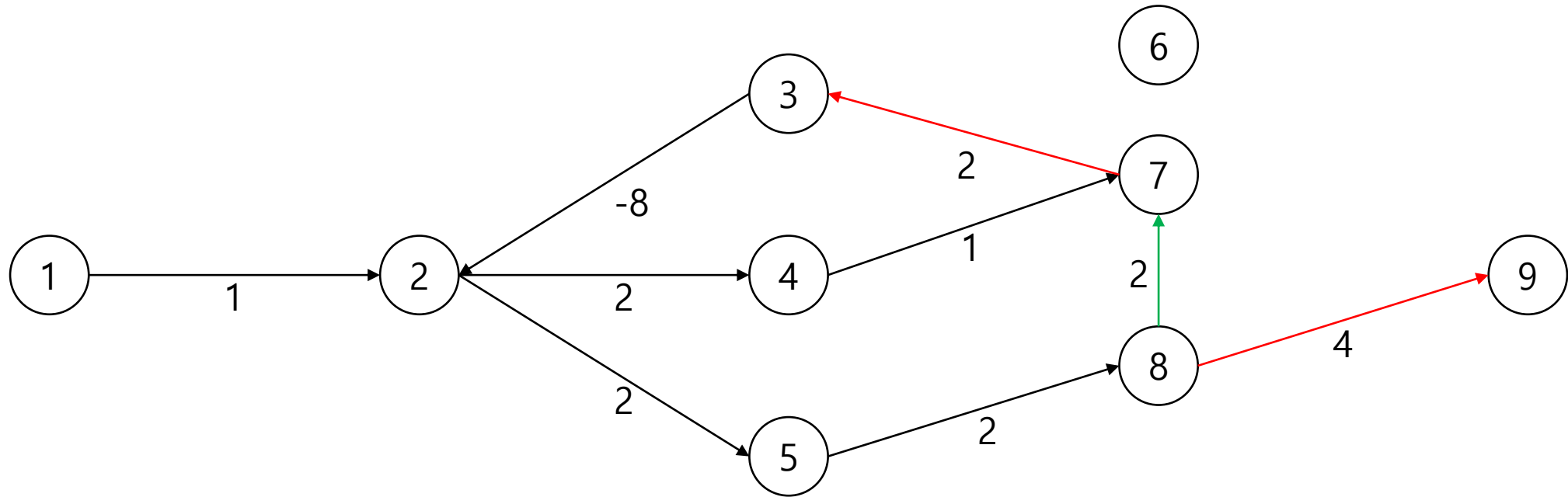
1	2	3	4	5	6	7	8	9
0	1	inf	3	3	inf	inf	inf	inf

벨만 포드 알고리즘 - 3



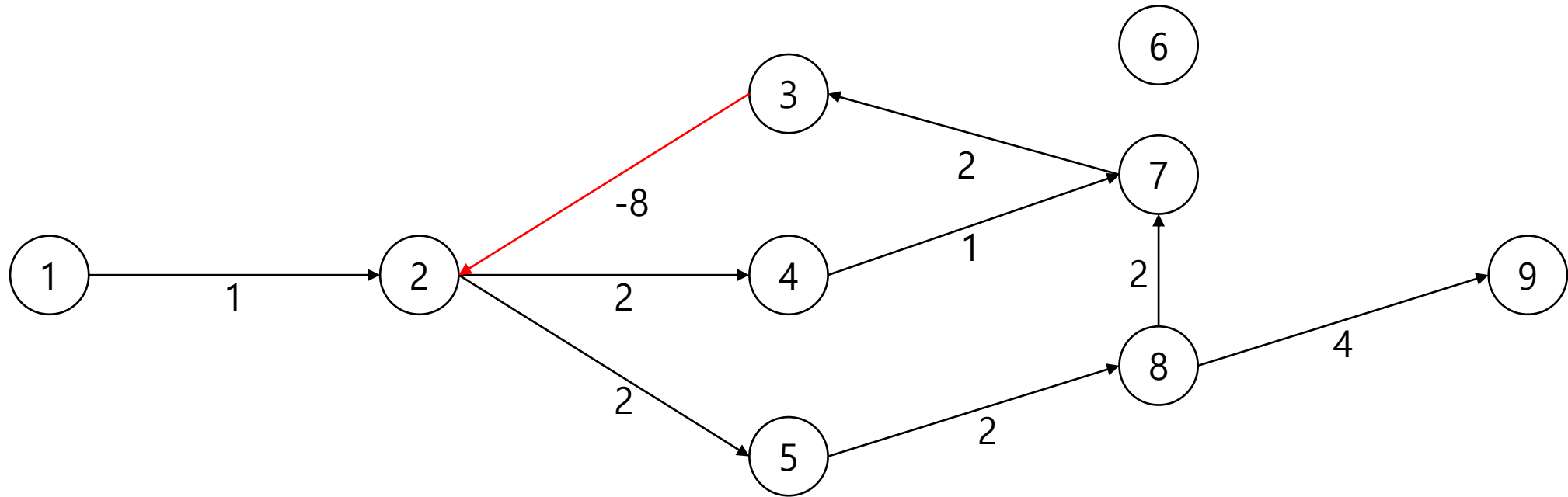
1	2	3	4	5	6	7	8	9
0	1	inf	3	3	inf	4	5	inf

벨만 포드 알고리즘 - 4



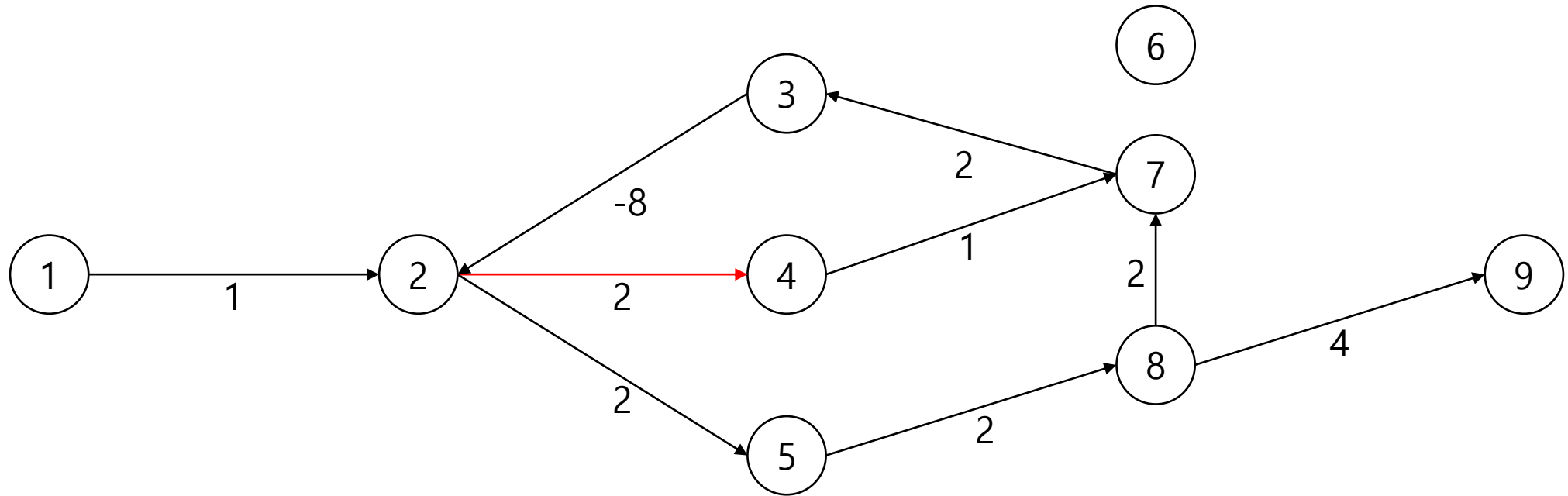
1	2	3	4	5	6	7	8	9
0	1	6	3	3	inf	4	5	9

벨만 포드 알고리즘 - 5



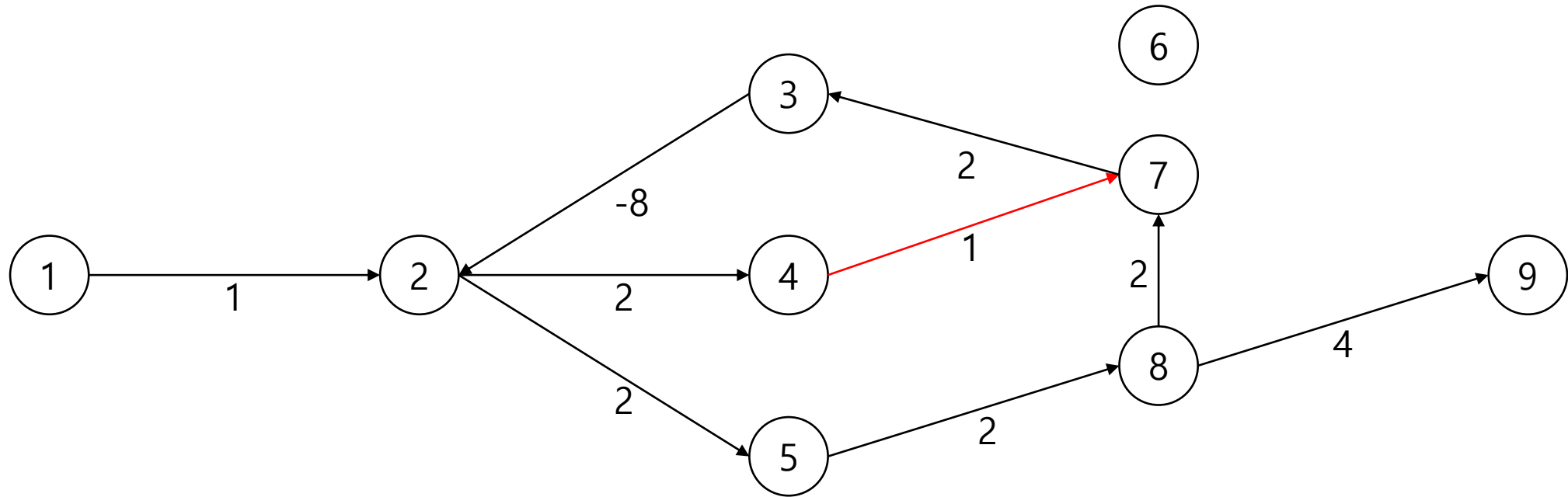
1	2	3	4	5	6	7	8	9
0	-2	6	3	3	inf	4	5	9

벨만 포드 알고리즘 - 6



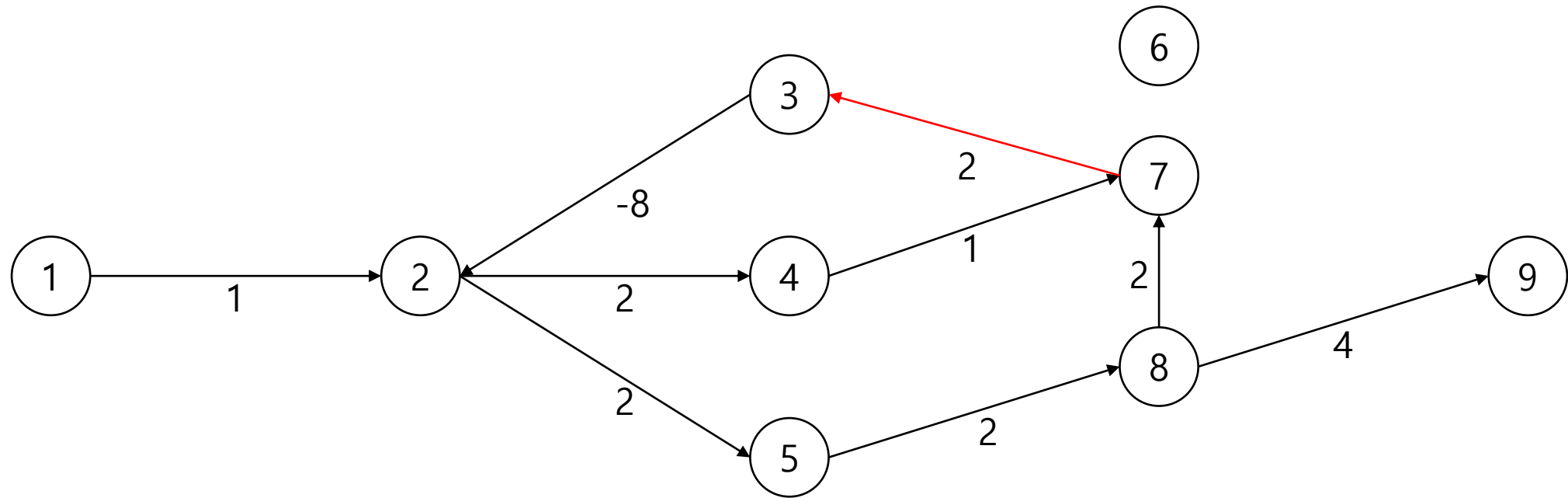
1	2	3	4	5	6	7	8	9
0	-2	6	0	3	inf	4	5	9

벨만 포드 알고리즘 - 7



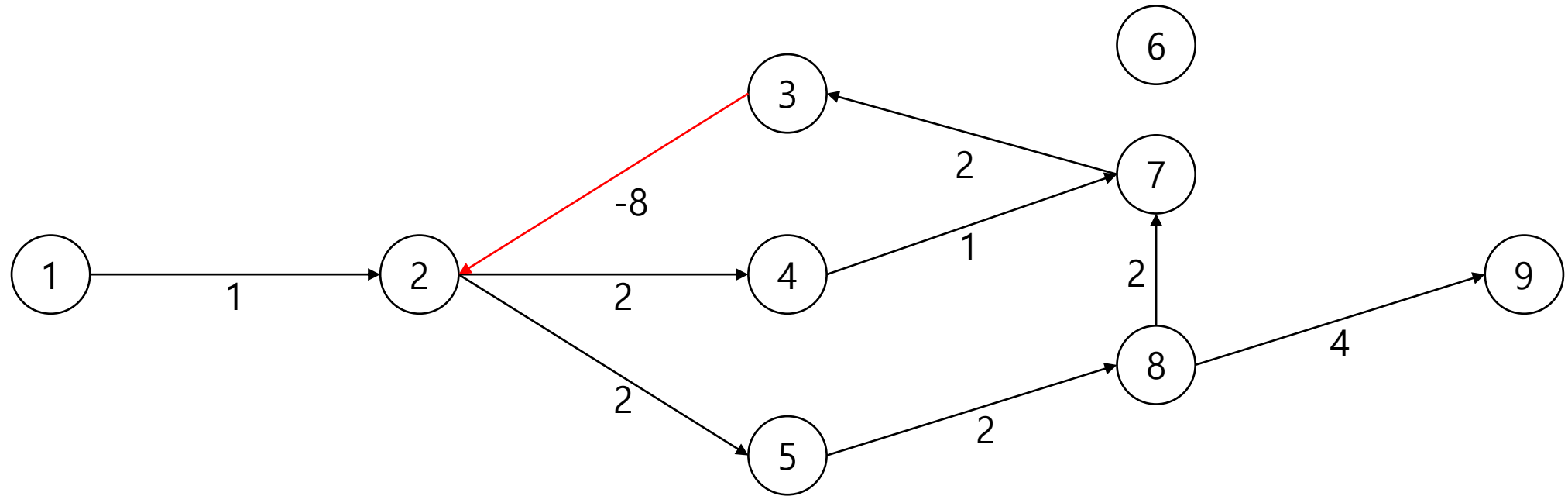
1	2	3	4	5	6	7	8	9
0	-2	6	0	3	inf	1	5	9

벨만 포드 알고리즘 - 8



1	2	3	4	5	6	7	8	9
0	-2	3	0	3	inf	1	5	9

벨만 포드 알고리즘 - 9



1	2	3	4	5	6	7	8	9
0	-5	3	0	3	inf	1	5	9

벨만 포드 알고리즘



```
struct Edge{
    int s, e, w;
    Edge() = default;
    Edge(int s, int e, int w) : s(s), e(e), w(w) {}
};

int N, M;
vector<Edge> E;
ll D[555], INF;

bool BellmanFord(int S){
    INF = N * 10000LL;
    fill(D, D+555, INF);
    D[S] = 0;
    for(int iter=1; iter<=N; iter++){
        bool isChanged = false;
        for(const auto &[s,e,w] : E){
            if(D[s] == INF) continue; // 간선의 출발점이 INF라면 넘어감
            if(D[e] > D[s] + w){      // relaxation
                D[e] = D[s] + w;
                isChanged = true;
            }
        }
        // N번째 iteration에서 relaxation이 발생하면 음수 사이클 존재
        if(isChanged && iter == N) return false;
    }
    return true;
}
```

벨만 포드 알고리즘

시간 복잡도

- V 번의 iteration * E 개의 간선 탐색 : $O(VE)$

주의 사항

- 거리 배열은 $V * (\text{간선 가중치 최댓값})$ 으로 초기화 : 모든 경로는 최대 $V-1$ 개의 간선으로 구성
- 실행 도중에 값이 약 $V * E * (\text{간선 가중치 최솟값})$ 까지 내려갈 수 있음 : 오버 플로우 주의
 - 간선이 $(1,2,-x), (2,1,-x), (1,2,-x), (2,1,-x), \dots$, 순으로 주어진다고 하자.
 - 한 번의 iteration에서
 - $D[2] = -x, D[1] = -2x, D[2] = -3x, \dots$
 - iteration 종료되면 $D[1] = D[2] \approx -xE/2$
 - V 번 반복하므로 $D[1] = D[2] \approx -xVE/2$

벨만 포드 알고리즘 - 예시

BOJ 1219 오민식의 고민

- 정점과 간선에 가중치가 있는 유향 그래프가 주어짐
- 간선을 따라 이동할 때마다 가중치 만큼 돈을 지불해야 하고, 정점을 방문할 때마다 가중치 만큼 돈을 얻음
- S번 정점에서 T번 정점으로 갈 때 얻을 수 있는 최대 이익을 구하는 문제
- 같은 정점을 여러 번 방문할 수 있고, 방문할 때마다 돈을 얻음
- 도착이 불가능하면 "gg" 출력, 돈을 무한히 많이 얻을 수 있다면 "Gee" 출력
- 돈을 $A[x]$ 만큼 얻는 것을 $-A[x]$ 만큼 잃는 것이라고 생각하자.
- S에서 출발해서 음수 사이클을 지난 다음 T로 갈 수 있으면 "Gee"를 출력하는 문제
 - 정점 가중치는 $x*2$ 번 정점에서 $x*2+1$ 번 정점으로 가는 가중치 $-A[x]$ 간선으로 생각할 수 있음
- 음수 사이클을 지난 다음 도착점으로 이동
 - 음수 사이클을 판별할 때 S에서 도달 가능한 사이클인지, T까지 도달할 수 있는 사이클인지 확인해야 함
 - S에서 도달할 수 없거나, T로 도달할 수 없는 정점을 모두 제거한 다음 벨만 포드 알고리즘 수행
- 실수할 여지가 많은 문제

```

#include <bits/stdc++.h>
using namespace std;
using ll = long long;
constexpr ll INF = 0x3f3f3f3f3f3f3f;

ll N, M, S, T, D[111], U[111];
vector<pair<ll,ll>> G[111], R[111];
void AddEdge(int s, int e, int w){
    G[s].emplace_back(e, w);
    R[e].emplace_back(s, w);
}

void DFS(vector<pair<ll,ll>> *gph, int v, int flag){
    U[v] |= flag;
    for(auto [i,w] : gph[v]) if(~U[i] & flag) DFS(gph, i, flag);
}

bool BellmanFord(){
    memset(D, 0x3f, sizeof D); D[S*2] = 0;
    for(int iter=1; iter<=N+N; iter++){
        bool flag = false;
        for(int i=0; i<N+N; i++){
            if(U[i] != 3 || D[i] == INF) continue;
            for(auto [j,w] : G[i]){
                if(U[j] == 3 && D[j] > D[i] + w){
                    D[j] = D[i] + w; flag = true;
                }
            }
        }
        if(flag && iter == N+N) return false;
    }
    return true;
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N >> S >> T >> M;
    for(int i=1,s,e,w; i<=M; i++) cin >> s >> e >> w, AddEdge(s*2+1, e*2, w);
    for(int i=0,t; i<N; i++) cin >> t, AddEdge(i*2, i*2+1, -t);
    DFS(G, S*2, 1); DFS(R, T*2+1, 2);
    if(U[T*2+1] != 3) cout << "gg";
    else if(!BellmanFord()) cout << "Gee";
    else cout << -D[T*2+1];
}

```

벨만 포드 알고리즘 - 예시

BOJ 7577 탐사

- 길이가 N인 배열이 있고, 각 칸의 값은 0 또는 1임
- 각 칸의 정확한 값은 모르지만, x부터 y까지의 합이 r이라는 정보는 알고 있음
- 가능한 배열의 값 중 하나를 구하는 문제
- 누적 합 배열을 생각하면, 주어지는 정보는 $S[y] - S[x-1] = r$ 이라는 것을 의미함
- 누적 합 배열에서 성립해야 하는 조건을 생각해 보자.
 - x부터 y까지의 합이 r이다: $S[y] - S[x-1] = r$
 - 각 칸의 값은 0 또는 1이다: $S[i-1] \leq S[i] \leq S[i-1] + 1$
 - 누적 합 배열의 모든 칸은 0 이상이다: $S[i] \geq 0$

벨만 포드 알고리즘 - 예시

Difference Constraints System

- $X_j - X_i \leq A$ 또는 $X_j - X_i \geq B$ 꼴의 부등식이 여러 개 주어지면, 이를 만족하는 $X_1, X_2, \dots, X_N$ 찾는 문제
- 다익스트라 / 벨만 포드에서 사용했던 거리 완화(relaxation) 방식 생각해 보면...
 - *if $D_i > D_v + w$ then $D_i \leftarrow D_v + w$*
 - 즉, 이 if문을 거치고 나면 $D_i \leq D_v + w$ 가 성립함
- $X_j - X_i \leq A$ 는 $X_j \leq X_i + A$ 와 같음 $\rightarrow i$ 에서 j 로 가는 가중치가 A 인 간선
- $X_j - X_i \geq B$ 는 $X_j \leq X_i + (-B)$ 와 같음 $\rightarrow i$ 에서 j 로 가는 가중치가 $-B$ 인 간선
- 해결 방법
 - 벨만 포드 알고리즘을 사용하면 가능한 해를 하나 구할 수 있음
 - 음수 사이클이 존재함과 부등식을 만족하는 해가 없음은 동치

벨만 포드 알고리즘 - 예시

BOJ 7577 탐사

- 누적 합 배열에서 성립해야 하는 조건을 생각해 보자.
 - x부터 y까지의 합이 r이다: $S[y] - S[x-1] = r$
 - $S[y] \leq S[x-1] + r$ x-1 에서 y 로 가는 가중치 r 간선
 - $S[x-1] \leq S[y] - r$ y 에서 x-1 로 가는 가중치 -r 간선
 - 각 칸의 값은 0 또는 1이다: $S[i-1] \leq S[i] \leq S[i-1] + 1$
 - $S[i-1] \leq S[i]$ i 에서 i-1 로 가는 가중치 0 간선
 - $S[i] \leq S[i-1] + 1$ i-1 에서 i 로 가는 가중치 1 간선
 - 누적 합 배열의 모든 칸은 0 이상이다: $S[i] \geq 0$
 - N+1(더미 정점)에서 다른 모든 정점으로 가는 가중치 0 간선
- N+1에서 시작하는 벨만 포드 알고리즘을 이용해 해결 가능
 - 음수 사이클이 발견되면 해가 존재하지 않음

벨만 포드 알고리즘 - 예시

BOJ 7577 탐사



```
int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N >> M;
    for(int i=1; i<=M; i++){
        int s, e, w; cin >> s >> e >> w;
        E.emplace_back(s-1, e, w);
        E.emplace_back(e, s-1, -w);
    }
    for(int i=0; i<=N; i++) E.emplace_back(N+1, i, 0);
    for(int i=1; i<=N; i++){
        E.emplace_back(i-1, i, 1);
        E.emplace_back(i, i-1, 0);
    }
    if(!BellmanFord(N+1)){ cout << "NONE"; return 0; }
    for(int i=1; i<=N; i++){
        if(D[i-1] + 1 == D[i]) cout << "#";
        else cout << "-";
    }
}
```