2022-1학기 스터디 #10

나정휘

https://justiceHui.github.io/

목차

Shortest Path Algorithm

Shortest Path Algorithm

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

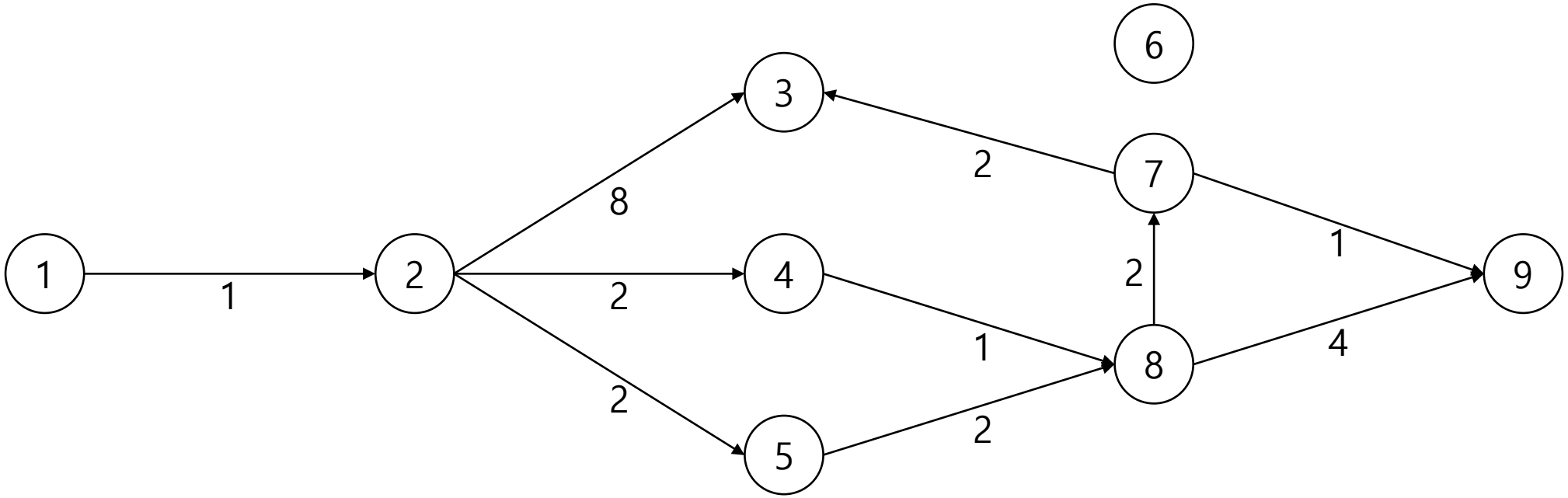
Dijkstra’s Algorithm

Dijkstra's Algorithm

- Dijkstra's Algorithm
  - 가중치가 0 이상인 그래프에서 SSSP를 푸는 알고리즘
  - 시간 복잡도 : $O(V^2)$ / $O(E \log V)$

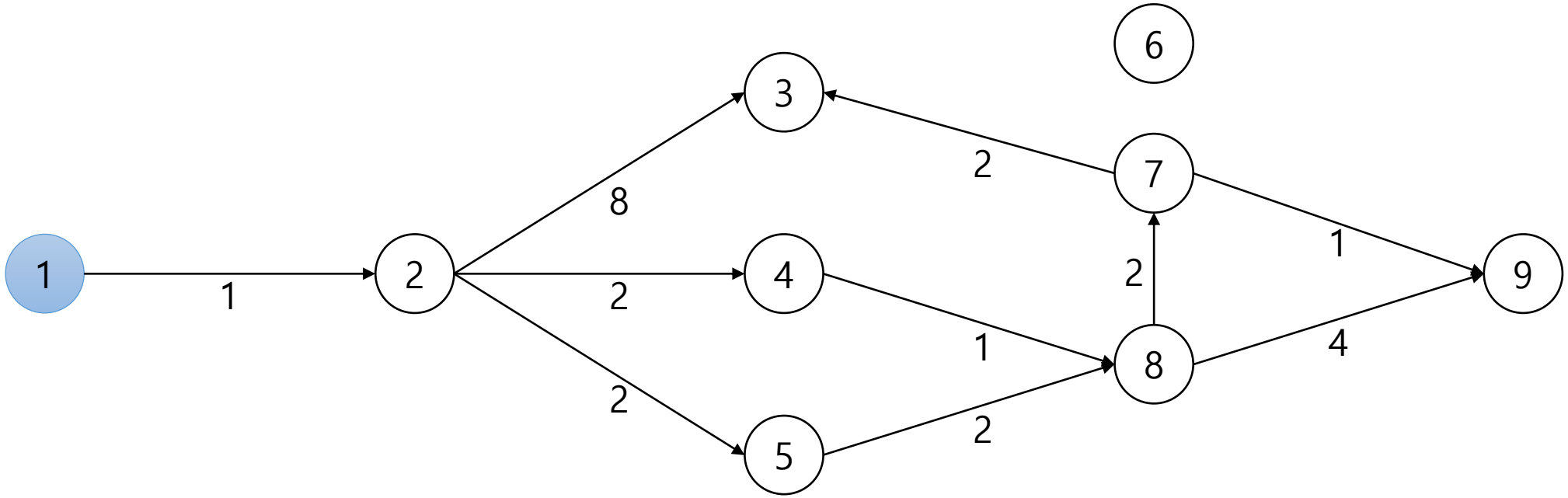
  - 그리디 기반 알고리즘
    1. 시작점(S)까지의 거리는 0, 다른 모든 정점까지의 거리는 INF로 초기화
    2. 아직 거리가 "확정"되지 않은 정점 중 거리가 가장 짧은 정점(v) 선택 (처음에는 S를 선택함)
    3. v의 거리를 "확정"시킴
    4. v와 인접하면서 아직 거리가 "확정"되지 않은 정점들의 거리를 갱신(D[i] ← D[v] + weight)
    5. 모든 정점의 거리가 "확정"되었다면 종료 / 그렇지 않으면 2번으로 돌아감

Dijkstra's Algorithm



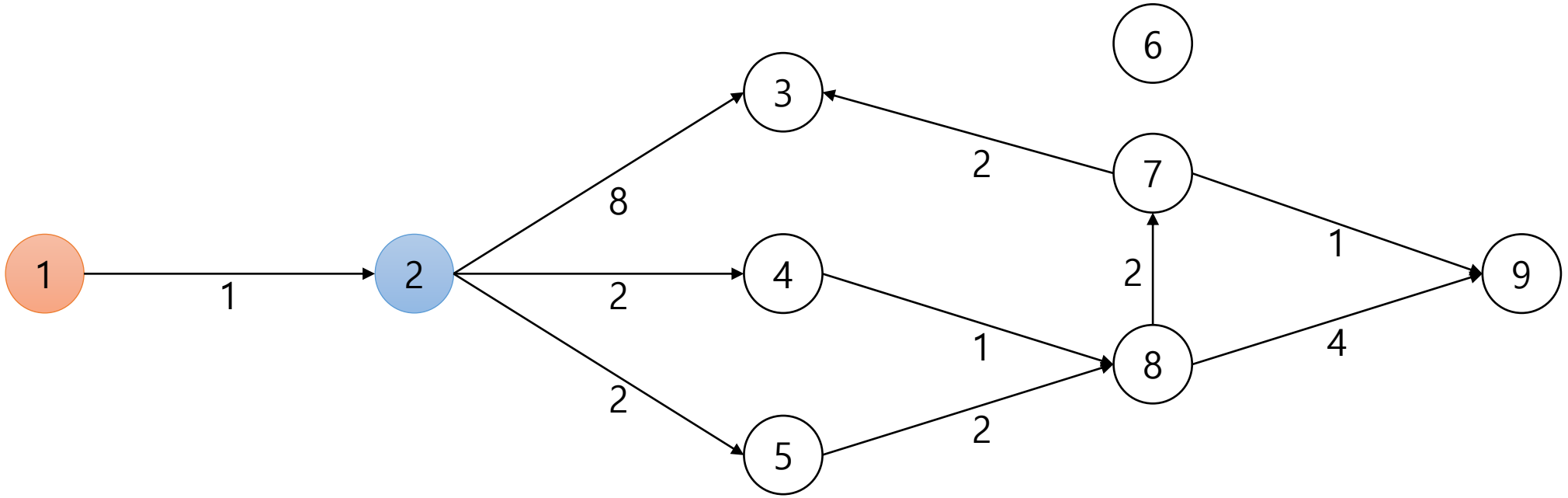
1	2	3	4	5	6	7	8	9
0	inf	inf	inf	inf	inf	inf	inf	inf

Dijkstra's Algorithm



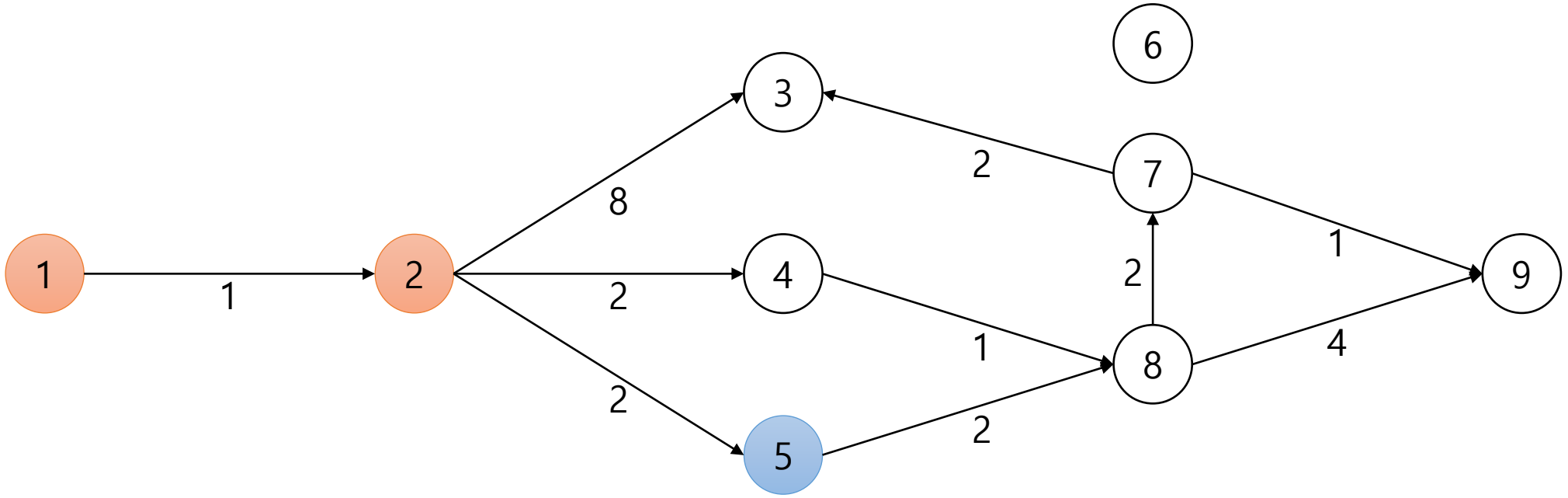
1	2	3	4	5	6	7	8	9
0	1	inf	inf	inf	inf	inf	inf	inf

Dijkstra's Algorithm



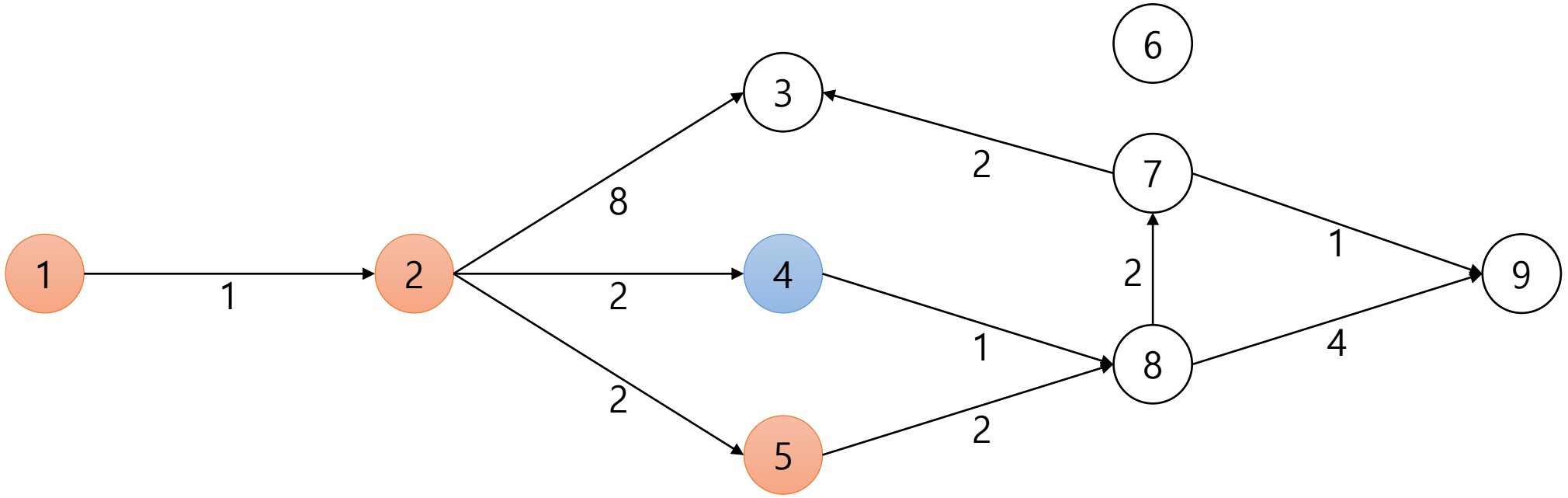
1	2	3	4	5	6	7	8	9
0	1	9	3	3	inf	inf	inf	inf

Dijkstra's Algorithm



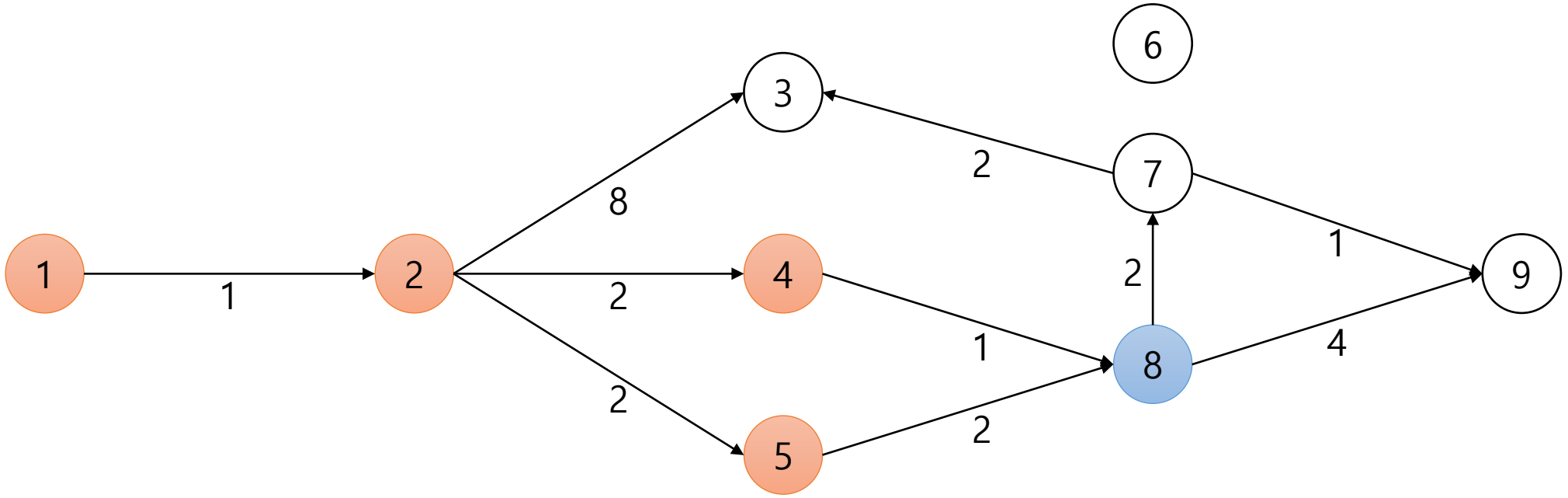
1	2	3	4	5	6	7	8	9
0	1	9	3	3	inf	inf	5	inf

Dijkstra's Algorithm



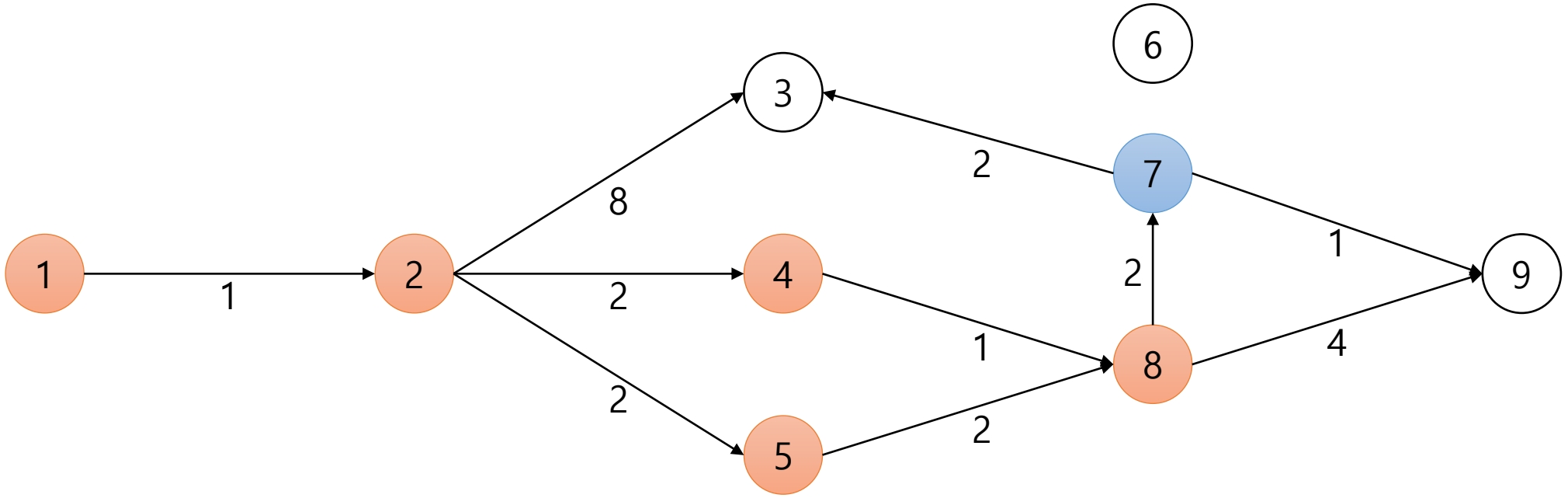
1	2	3	4	5	6	7	8	9
0	1	9	3	3	inf	inf	4	inf

Dijkstra's Algorithm



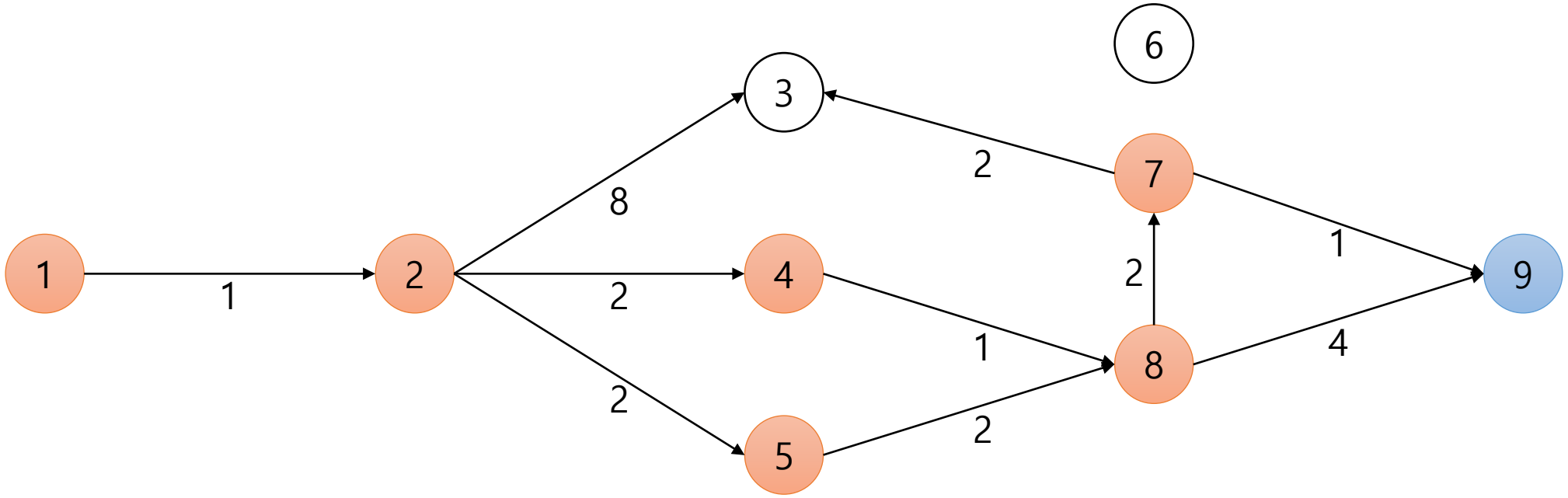
1	2	3	4	5	6	7	8	9
0	1	9	3	3	inf	6	4	8

Dijkstra's Algorithm



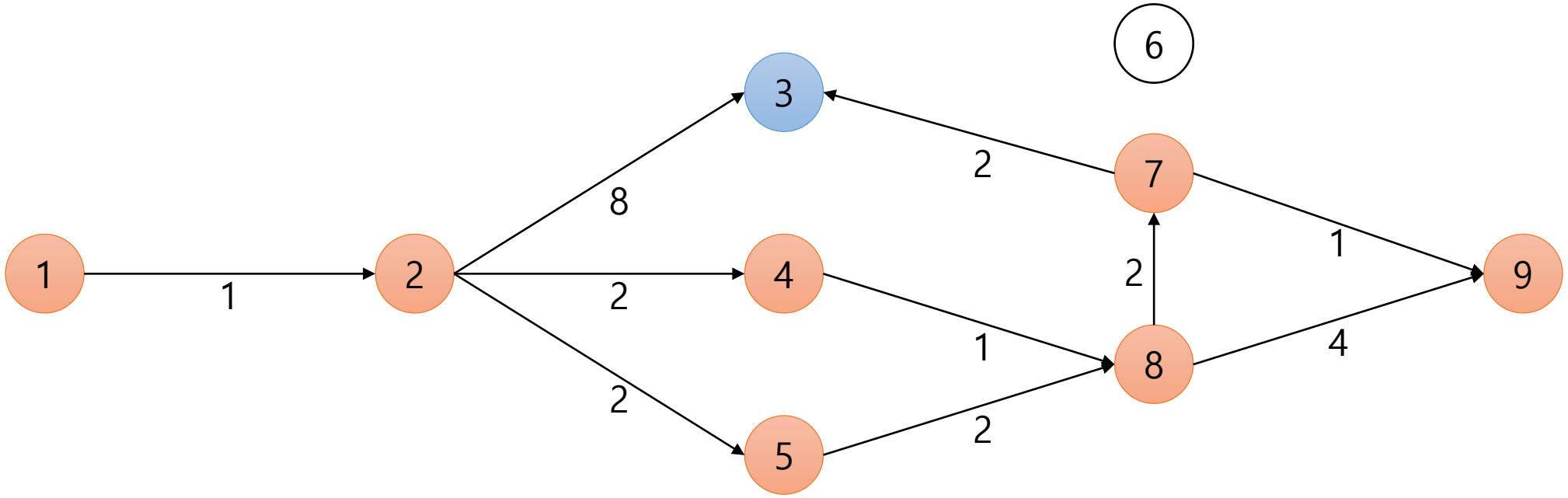
1	2	3	4	5	6	7	8	9
0	1	8	3	3	inf	6	4	7

Dijkstra's Algorithm



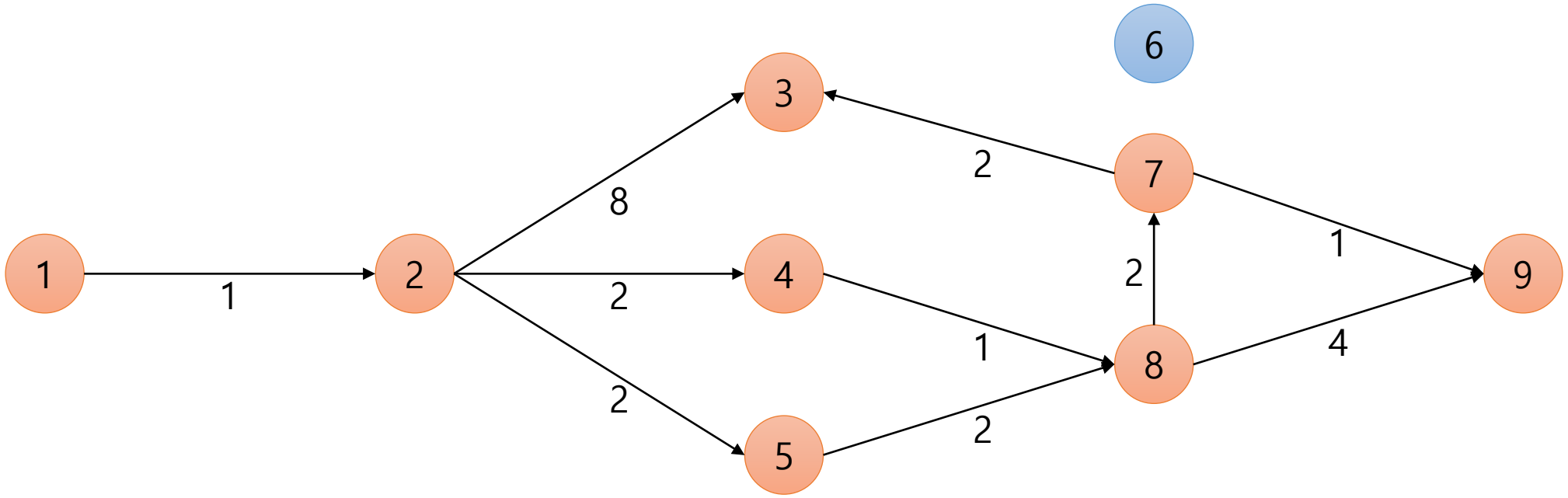
1	2	3	4	5	6	7	8	9
0	1	8	3	3	inf	6	4	7

Dijkstra's Algorithm



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

Dijkstra's Algorithm



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

Dijkstra's Algorithm

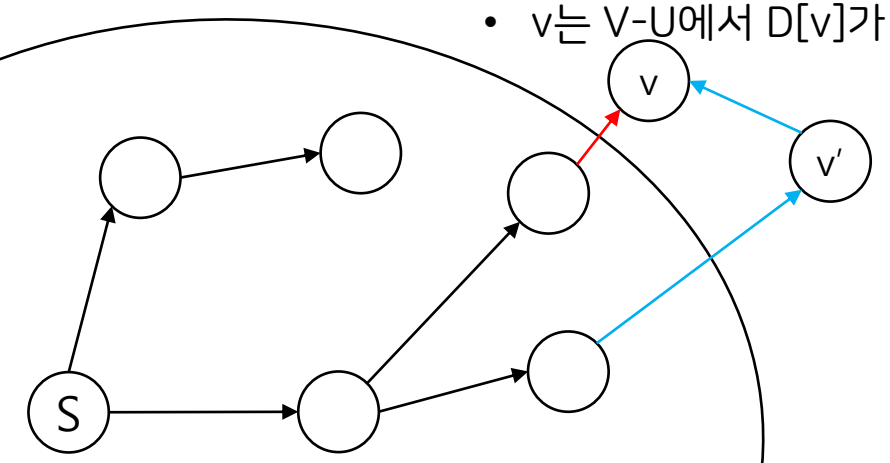
```cpp
using ll = long long;
using PLL = pair<ll, ll>;
constexpr int MAX_N = 5050;

int N, M;
vector<PLL> G[MAX_N]; // 인접 리스트
ll D[MAX_N]; // 거리
bool C[MAX_N]; // 거리 확정 여부

void Dijkstra(int S){
    memset(D, 0x3f, sizeof D);                                          // inf로 초기화
    D[S] = 0;                                                           // 시작점의 거리는 0으로 초기화
    for(int iter=1; iter<=N; iter++){
        int v = -1;
        for(int i=1; i<=N; i++){
            if(C[i]) continue;                                          // 이미 거리가 확정된 정점은 넘어감
            if(v == -1 || D[v] > D[i]) v = i;                           // 거리가 가장 작은 정점 선택
        }
        C[v] = 1;                                                       // 거리 확정
        for(auto [i,w] : G[v]) if(!C[i]) D[i] = min(D[i], D[v] + w); // 거리 갱신
    }
}
```

Dijkstra's Algorithm

- 정당성 증명
  - 수학적 귀납법을 사용해 증명
    - 거리가 확정된 정점 집합을 U라고 하자.
    - U에 정점 v를 추가할 때, D[v]가 v까지의 실제 최단 거리 sp[v]와 같음을 증명
      - v는 V-U에서 D[v]가 최소인 정점
    - 귀류법을 사용한다.
      - D[v] > sp[V]라고 가정하자.
      - s에서 v로 가는 경로에서 v를 방문하기 직전에 v' ∈ V-U를 방문해야 함
      - sp[v] = sp[v'] + w(v', v)라는 의미이고, w(v', v)는 양수이므로 sp[v'] < sp[v]가 되어야 함
      - v는 V-U에서 D[v]가 최소인 정점이 아니므로 모순



질문?

Dijkstra's Algorithm

- 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 뻗어 나가는 간선은 모두 봐야 하므로 O(deg(v))가 하한임
  - 거리가 최소인 정점을 빠르게 찾을 수 있을까?
    - Min Heap

Dijkstra's Algorithm

```
void Dijkstra(int S){
    memset(D, 0x3f, sizeof D);                          // inf로 초기화
    priority_queue<PLL, vector<PLL>, greater<>> pq; // {거리, 정점} pair를 저장하는 min heap
    D[S] = 0; pq.emplace(0, S);                         // 시작 정점 거리 초기화, {0, S}를 heap에 삽입
    while(!pq.empty()){
        // structured binding declaration 문법
        auto [cst,now] = pq.top(); pq.pop();            // cst: now까지의 거리
        if(C[now]) continue;                            // 이미 거리가 확정되었으면 넘어감
        C[now] = 1;                                     // now까지의 거리 확정
        for(auto [nxt,len] : G[now]){                   // {다음 정점, 간선 길이}
            if(D[nxt] > D[now] + len){                  // 거리 갱신
                D[nxt] = D[now] + len;
                pq.emplace(D[nxt], nxt);
            }
        }
    }
}
```

Dijkstra's Algorithm

- 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 * (간선 가중치 최댓값)으로 초기화 : 모든 경로는 최대 V-1개의 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)

Dijkstra's Algorithm

- 응용
  - 정점에 가중치가 있는 경우
    - 정점을 2개로 분할 : in(v), out(v)
    - u에서 v로 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
  - $\{S_1, S_2, \dots, S_k\}$에서 다른 모든 정점으로 가는 최단 거리
    - Multi Source Shortest Path
    - 새로운 정점 S_0에서 $S_1, S_2, \dots, S_k$로 가는 가중치 0 간선 만들면
    - S_0에서 다른 모든 정점으로 가는 최단 거리 문제로 바뀜

질문?

Floyd-Warshall Algorithm

Floyd-Warshall Algorithm

- 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] : u에서 출발해서 1..k번 정점을 경유한 뒤 v번 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] ← D[k-1][u][k] + D[k-1][k][v]
    - 배열 D의 크기는 V^3

Floyd-Warshall Algorithm

- 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^2로 줄일 수 있음
    - 최솟값을 구하는 문제인데 값이 매번 감소하기 때문에 그냥 덮어써도 됨
    - 시간 복잡도는 여전히 $O(V^3)$

Floyd-Warshall Algorithm

```cpp
int N, M, D[555][555];

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N >> M;
    memset(D, 0x3f, sizeof D);           // inf로 초기화
    for(int i=1; i<=N; i++) D[i][i] = 0; // i에서 i로 가는 비용은 0
    for(int i=1; i<=M; i++){
        int st, ed, cst;
        cin >> st >> ed >> cst;
        D[st][ed] = min(D[st][ed], cst);
    }
    // 여기까지 오면, D[i][j]는 정점을 경유하지 않고 i에서 j로 직접 가는 최단 거리임

    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
    }
}
```

질문?

Bellman-Ford Algorithm

Bellman-Ford Algorithm

- 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-1개의 간선을 지남

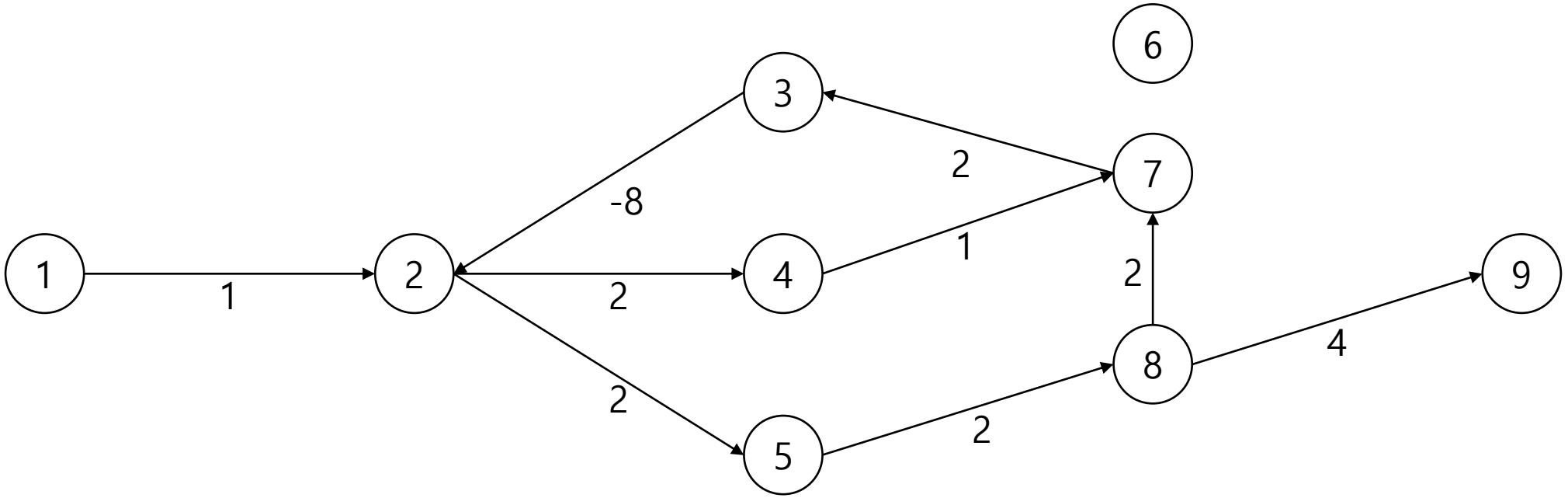
  - Dijkstra's Algorithm처럼 relaxation을 통해 최단 거리를 구함
    - 시작 정점 S의 거리는 0, 다른 모든 정점까지의 거리는 INF로 초기화
    - 모든 간선 (s, e, w)에 대해 D[e] = min(D[e], D[s] + w)를 수행하는 것을 V-1번 반복
      - i번째 iteration이 끝나면 간선 i개를 거쳐서 가는 최단 거리를 정확하게 구할 수 있음
      - 귀납법으로 증명 가능

Bellman-Ford Algorithm

- Bellman-Ford Algorithm
  - 음수 사이클이 존재하는지 판별하는 것도 가능

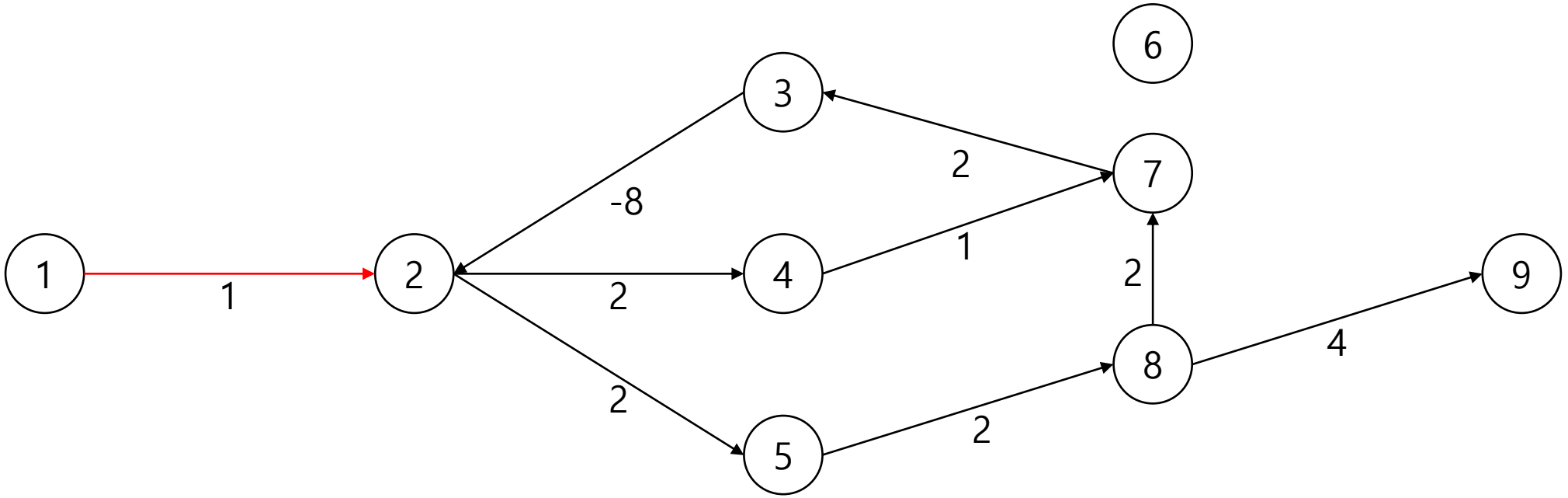
  - 음수 사이클이 없다면 V-1번의 iteration으로 항상 최단 거리를 구할 수 있음
  - 만약 V번째 iteration에서 relaxation이 발생하면? 음수 사이클 존재

Bellman-Ford Algorithm – 0



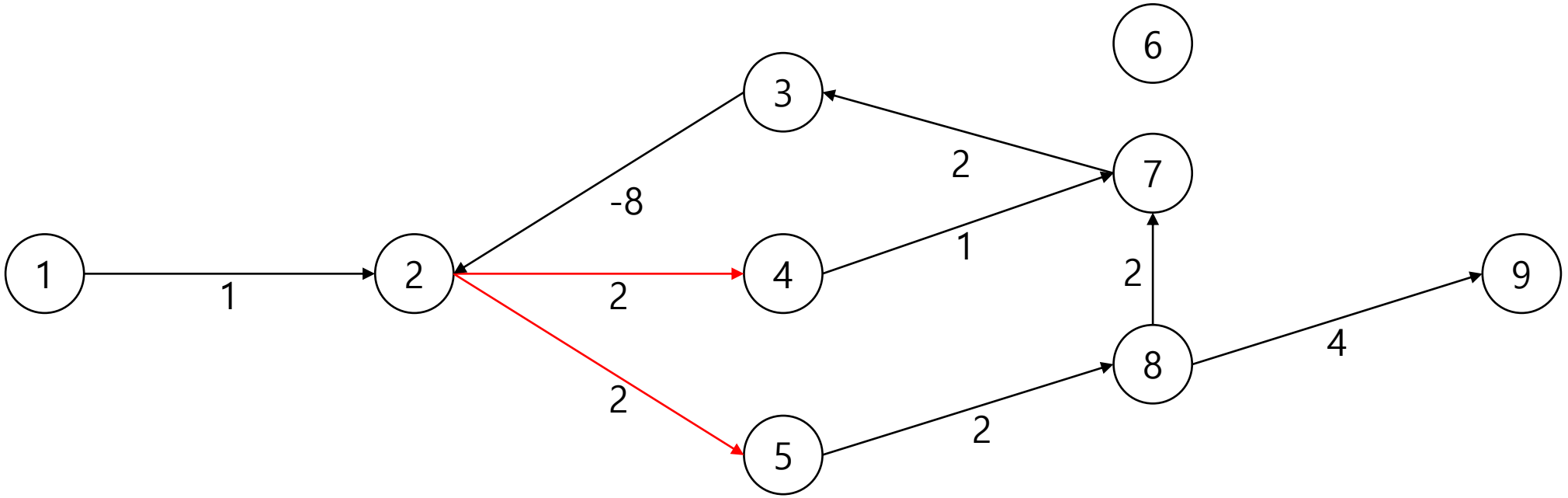
1	2	3	4	5	6	7	8	9
0	inf	inf	inf	inf	inf	inf	inf	inf

Bellman-Ford Algorithm – 1



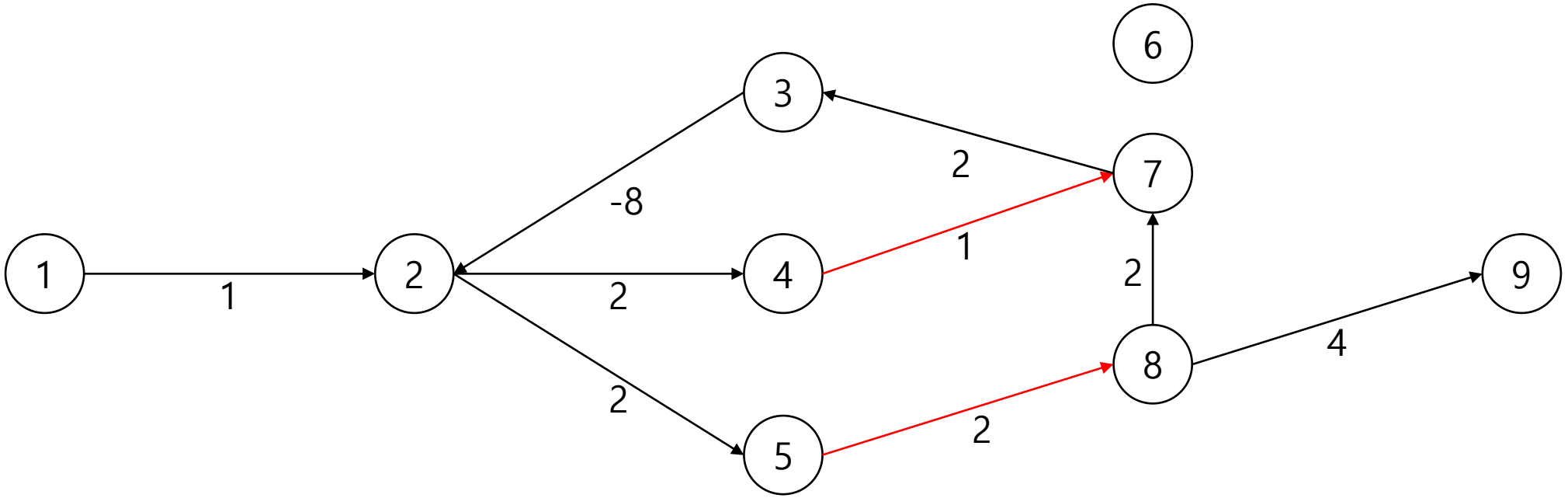
1	2	3	4	5	6	7	8	9
0	1	inf	inf	inf	inf	inf	inf	inf

Bellman-Ford Algorithm – 2



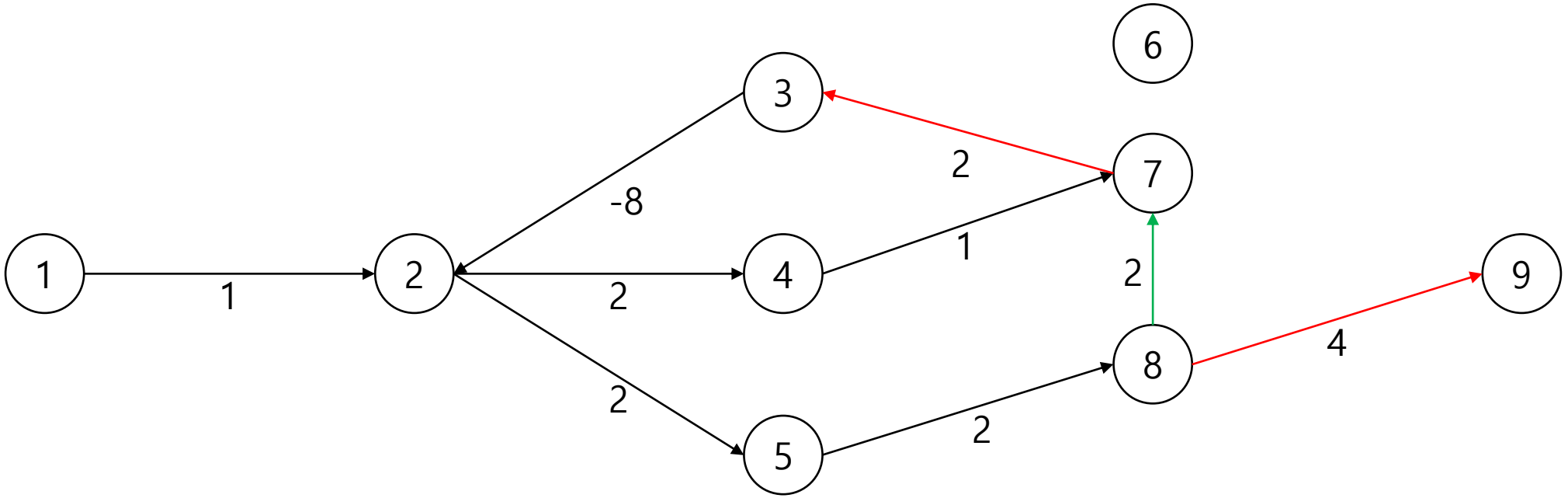
1	2	3	4	5	6	7	8	9
0	1	inf	3	3	inf	inf	inf	inf

Bellman-Ford Algorithm – 3



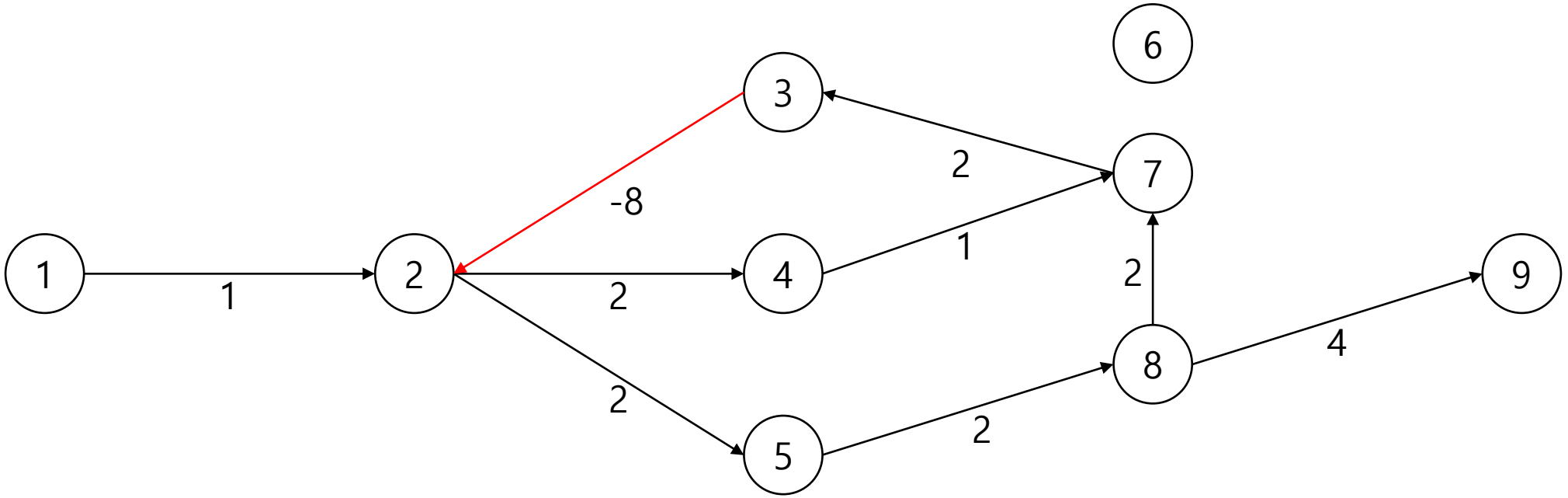
1	2	3	4	5	6	7	8	9
0	1	inf	3	3	inf	4	5	inf

Bellman-Ford Algorithm – 4



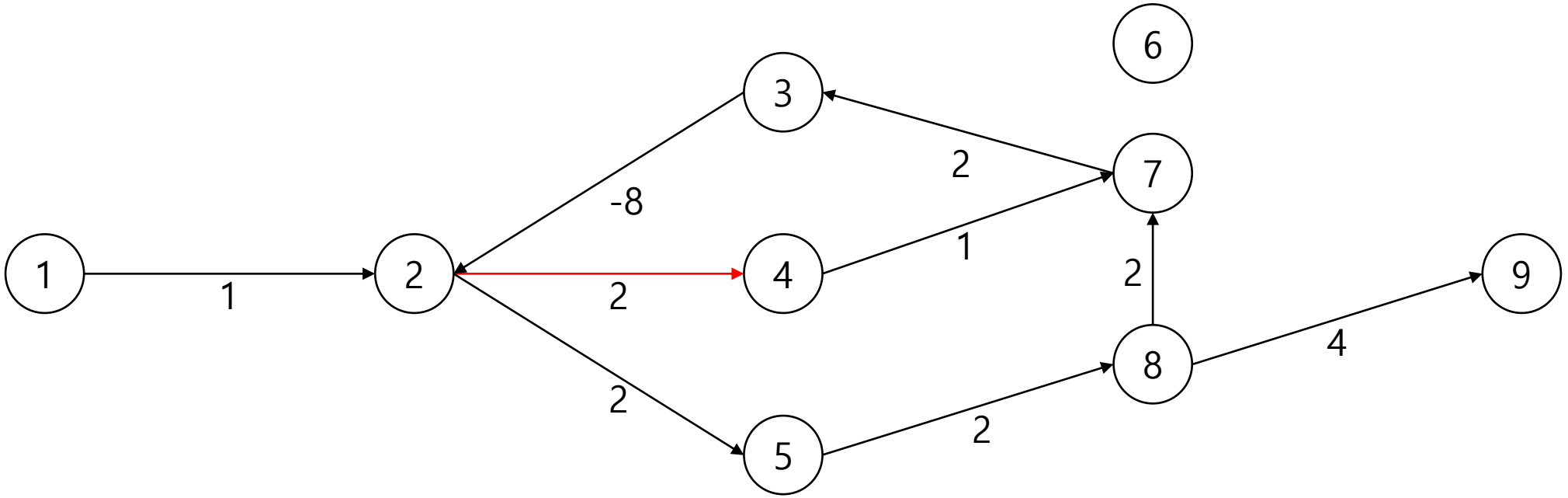
1	2	3	4	5	6	7	8	9
0	1	6	3	3	inf	4	5	9

Bellman-Ford Algorithm – 5



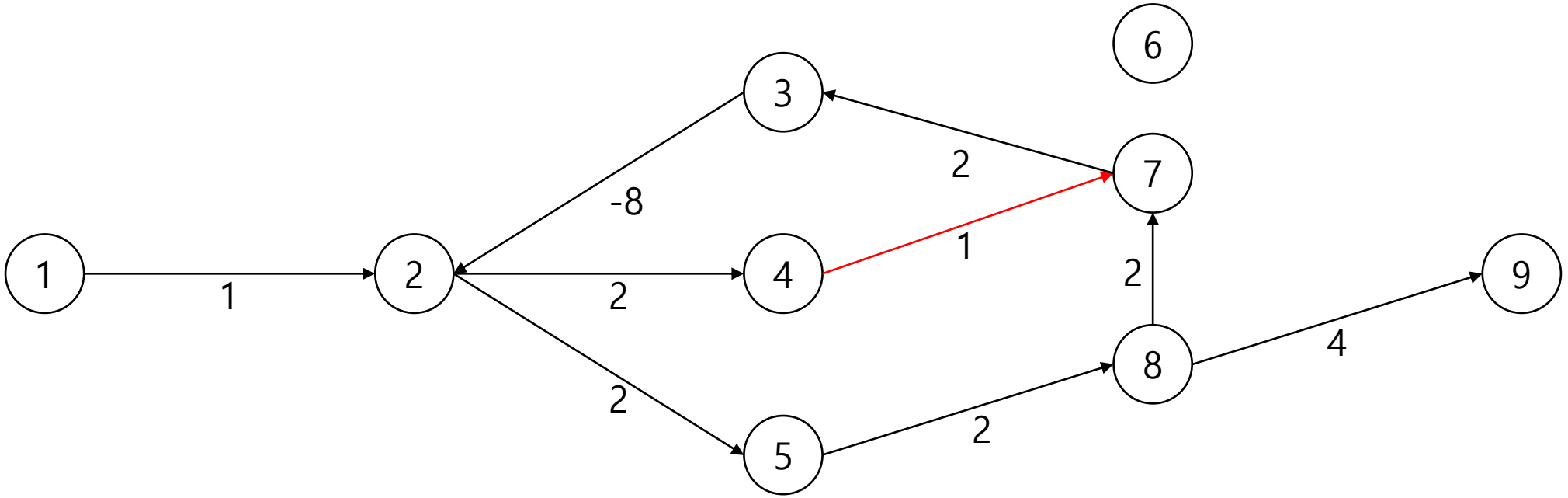
1	2	3	4	5	6	7	8	9
0	-2	6	3	3	inf	4	5	9

Bellman-Ford Algorithm – 6



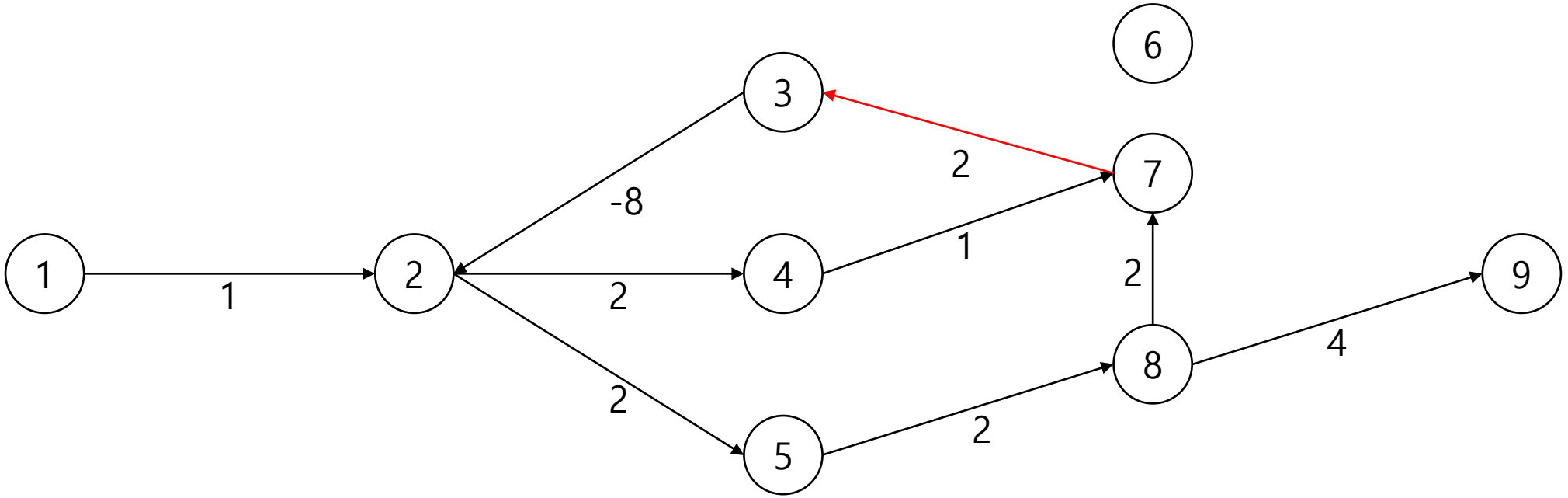
1	2	3	4	5	6	7	8	9
0	-2	6	0	3	inf	4	5	9

Bellman-Ford Algorithm – 7



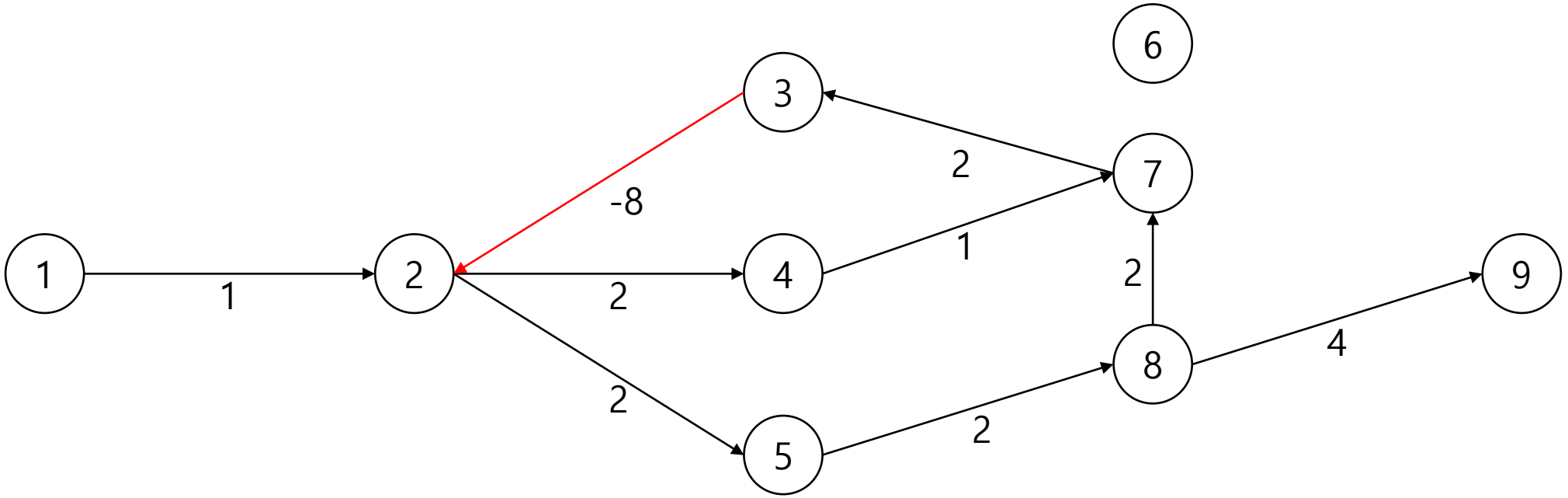
1	2	3	4	5	6	7	8	9
0	-2	6	0	3	inf	1	5	9

Bellman-Ford Algorithm – 8



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

Bellman-Ford Algorithm – 9



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

Bellman-Ford Algorithm

```cpp
using ll = long long;
constexpr ll INF = 0x3f3f3f3f3f3f3f3f;

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];

bool BellmanFord(int S){
    memset(D, 0x3f, sizeof D);
    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 INF면 넘어감
            if(D[e] > D[s] + w){
                D[e] = D[s] + w;
                isChanged = true;
            }
        }
        // N번째 iteration에서 relaxation이 발생하면 음수 사이클 존재
        if(isChanged && iter == N) return false;
    }
    return true;
}
```

Bellman-Ford Algorithm

- 시간 복잡도
  - V번의 iteration * E개의 간선 탐색 : O(VE)

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

Bellman-Ford Algorithm

- 응용
  - $X_b - X_a \leq w(a, b)$ 꼴의 부등식이 여러 개 주어졌을 때 X_i의 값을 구하는 문제
    - 가상의 정점 S에서 1, 2, … , N으로 가는 가중치 0 간선 만들고
    - a에서 b로 가는 가중치 w(a, b) 간선 만들면
    - S에서 시작하는 SSSP를 이용해 부등식을 만족하는 X_i를 구할 수 있음
  - Minimum Cost Flow
    - 생략

질문?

Shortest Path Faster Algorithm (SPFA)

SPFA

- SPFA
  - SSSP를 푸는 알고리즘 : Bellman-Ford의 연산량을 줄인 알고리즘
  - 시간 복잡도 : O(VE)

  - Bellman-Ford Algorithm이 O(VE)인 이유 : 매번 모든 간선을 다 보기 때문
    - 유효한 간선만 보는 방식으로 연산량을 줄일 수 있지 않을까?
    - 직전 iteration에서 거리가 갱신된 정점에 달려있는 간선만 고려해도 됨

    - 최악의 경우에는 O(VE)로 동일하지만, 보통 Bellman-Ford보다 빠름
    - 출제자가 데이터를 대충 만들었을 경우(랜덤 데이터) 평균적으로 O(V+E)
    - 그래프의 간선 구성이 매번 달라지는 경우(ex. MCMF) 매번 최악의 경우가 되지 않으므로 매우 빠름

SPFA

```cpp
using ll = long long;
using PLL = pair<ll, ll>;

int N, M;
vector<PLL> G[555];
ll D[555];
bool InQ[555];

bool SPFA(int S){
    queue<int> Q;
    memset(D, 0x3f, sizeof D);
    memset(InQ, false, sizeof InQ);

    Q.push(S); D[S] = 0; InQ[S] = true;
    for(int iter=1; !Q.empty(); iter++){
        if(iter > N) return false; // iteration이 N을 넘어갔으므로 음수 사이클 존재
        int sz = Q.size();         // 현재 iteration에서 봐야 하는 정점의 개수
        while(sz--){
            int v = Q.front(); Q.pop(); InQ[v] = false;
            for(auto [i,c] : G[v]){
                if(D[i] > D[v] + c){
                    D[i] = D[v] + c;
                    if(!InQ[i]) Q.push(i), InQ[i] = true; // 만약 i가 큐에 없다면 push
                }
            }
        }
    }
    return true;
}
```

SPFA

- 시간 복잡도
  - 최악의 경우
    - 최대 O(V)번의 iteration
    - 매번 모든 정점이 relaxation 된다면 모든 간선을 다 고려해야 하므로 O(VE)

  - 랜덤 데이터에서 O(V+E)라고 하던데 증명은 모름

질문?

Minimum Spanning Tree

Minimum Spanning Tree

- 용어 정의
  - 부분 그래프(Subgraph) : 그래프의 정점과 간선의 일부를 선택해서 만든 그래프
  - 신장 부분 그래프(Spanning ~) : 그래프의 모든 정점을 포함하는 부분 그래프
  - 신장 포레스트(Spanning Forest) : 사이클이 없는 신장 부분 그래프
  - 신장 트리(Spanning Tree) : 모든 정점이 연결된 신장 포레스트
  - 최소 비용 신장 트리(Minimum ~) : 간선의 가중치의 합이 최소인 신장 트리

Minimum Spanning Tree

- Minimum Spanning Tree
  - 그래프가 주어지면 최소 신장 트리를 구하는 알고리즘
  - 여러 가지 알고리즘이 있다.
    - Prim's Algorithm (V^2, E log V, E + V log V 등등)
    - Kruskal's Algorithm (E log E)
    - Boruvka's Algorithm (E log V, 이건 설명 안 함)
      - Sollin's Algorithm이라고 부르기도 함

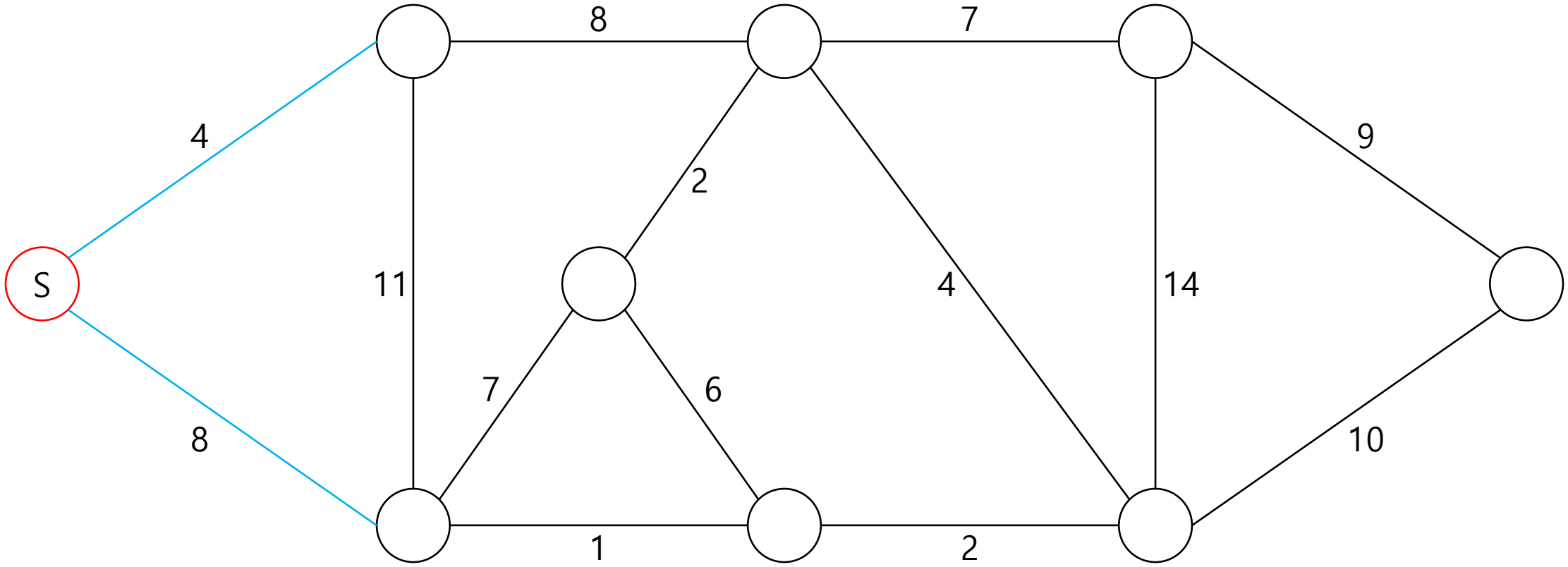
Prim's Algorithm

- Prim's Algorithm
  - Dijkstra's Algorithm과 매우 유사함
  - 시간 복잡도: $O(V^2)$ / O(E log E)

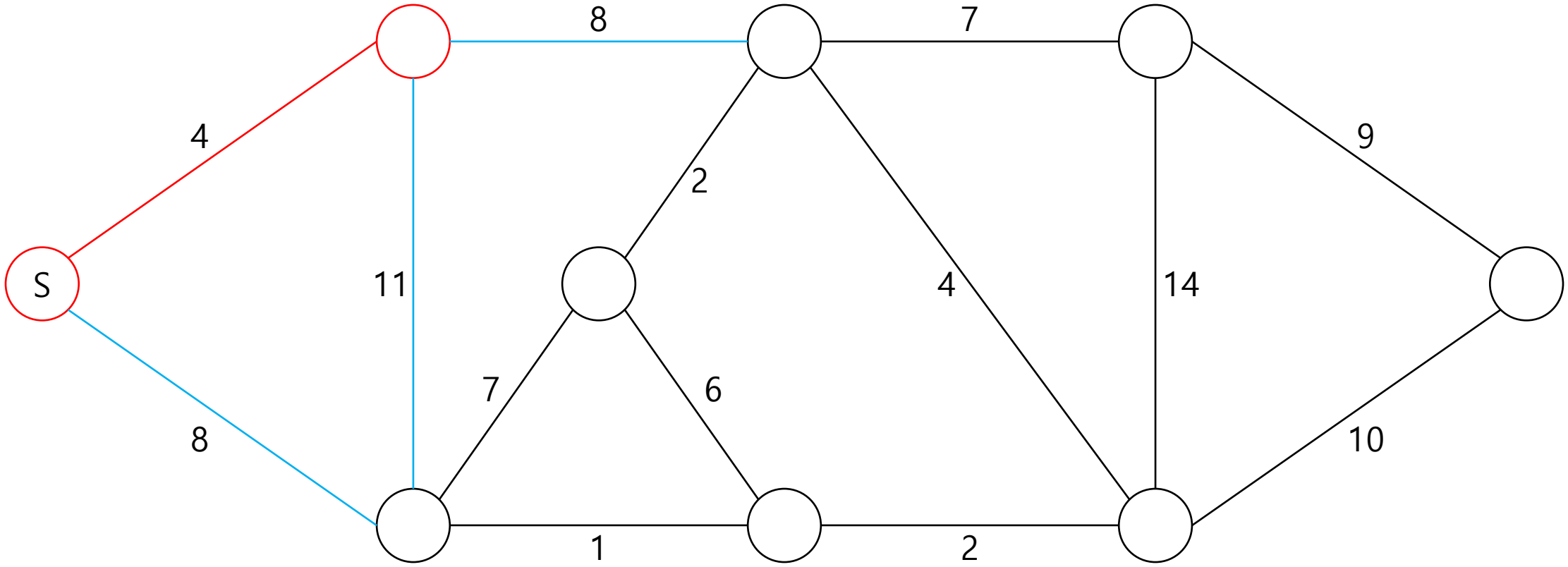
  - Spanning Tree에 정점을 하나씩 포함시키면서 확장하는 방식으로 진행

  - 그리디 기반 알고리즘
    1. 시작점(S)을 MST에 넣음
    2. 현재 MST에 있는 정점에서 뻗어 나가는 간선 중 가중치가 가장 작은 간선(e) 선택
    3. 만약 MST에 e를 추가할 수 있다면(사이클이 생기지 않는다면) e를 MST에 추가
       - 사이클 판별은 e = (u, v)에서 u와 v가 이미 MST에 포함되었는지 확인하면 됨

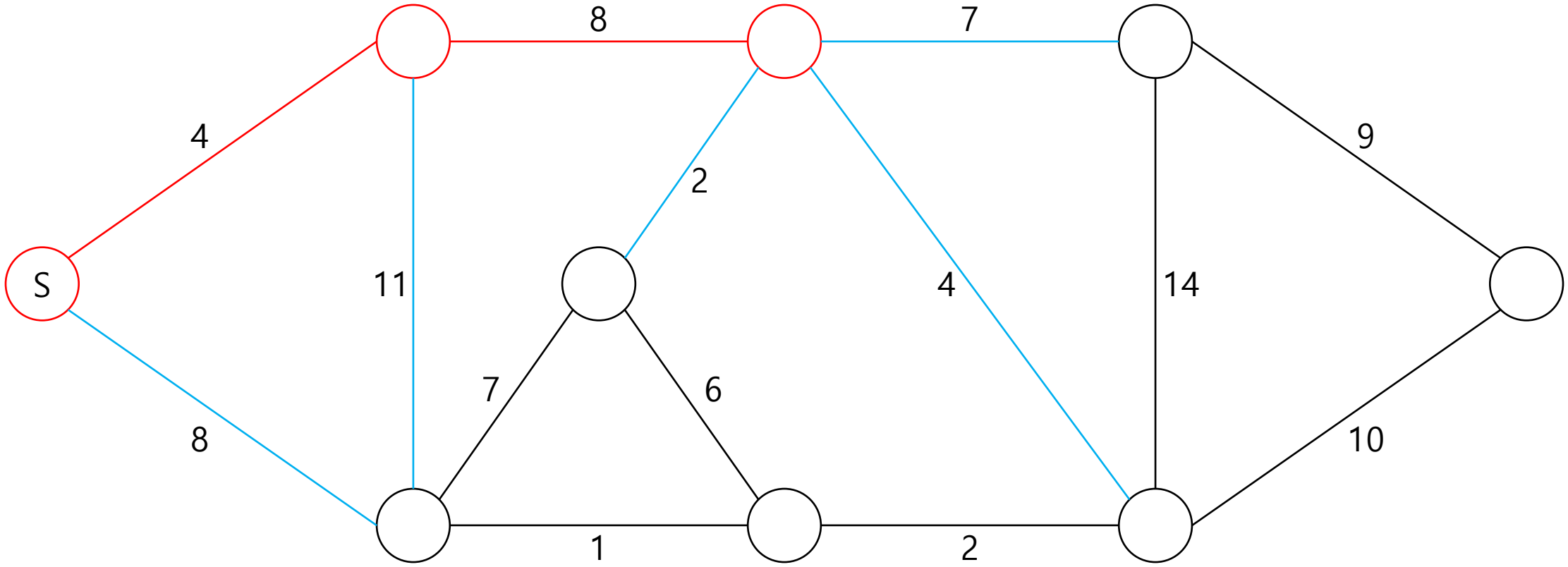
Prim's Algorithm



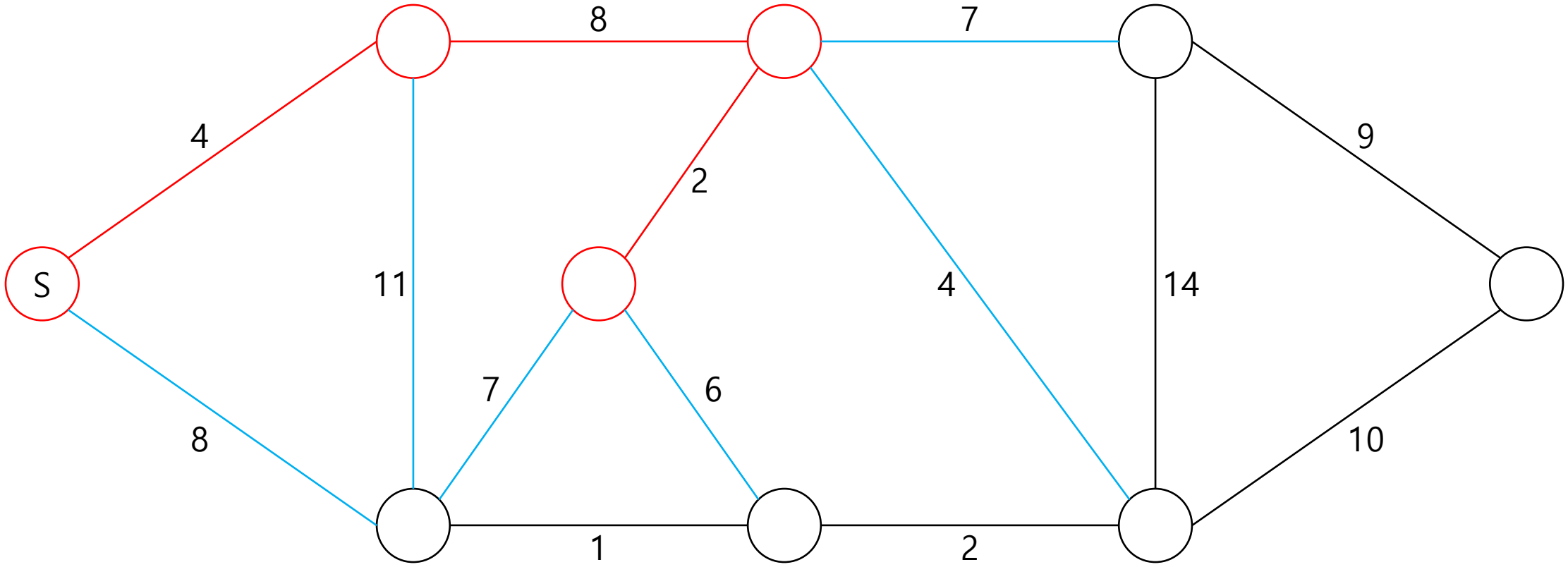
Prim's Algorithm



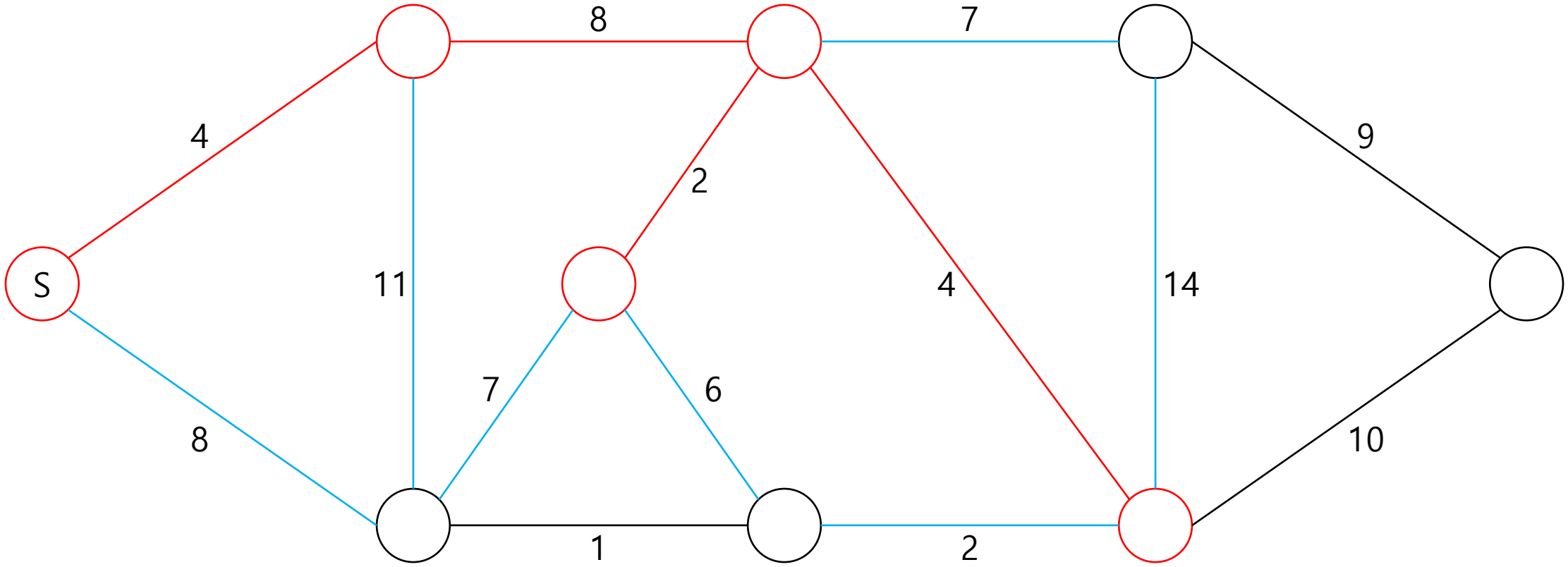
Prim's Algorithm



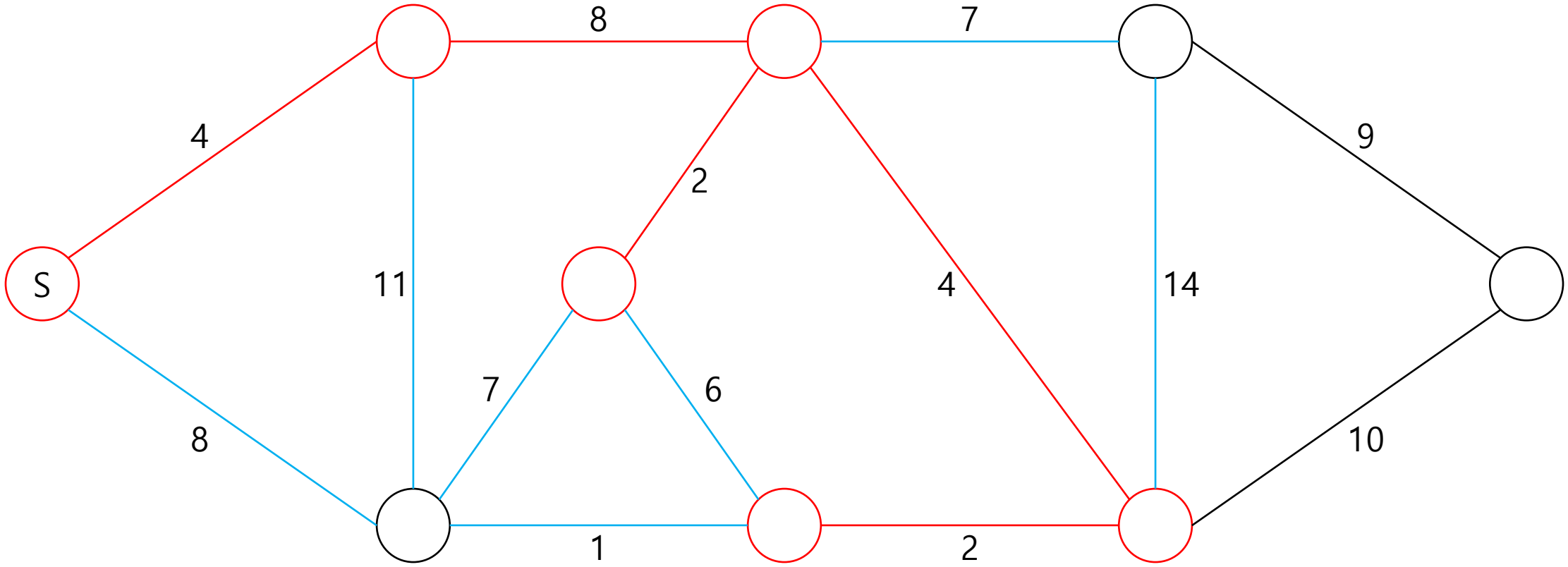
Prim's Algorithm



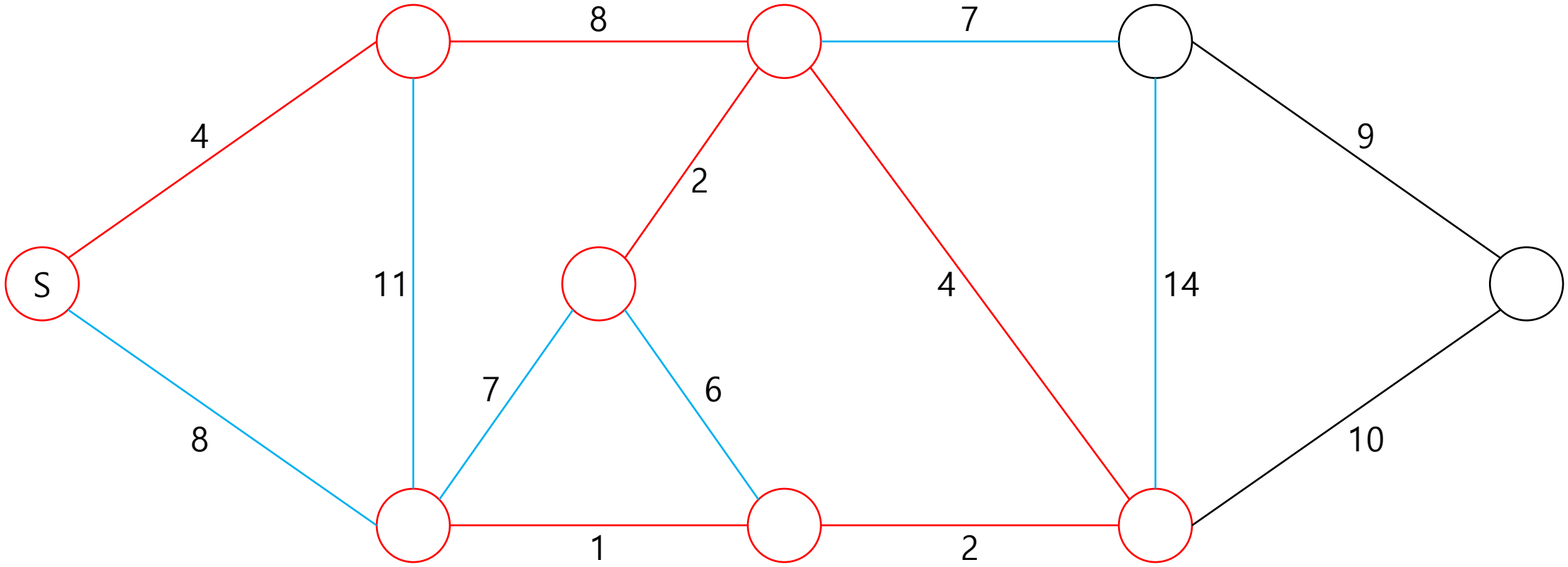
Prim's Algorithm



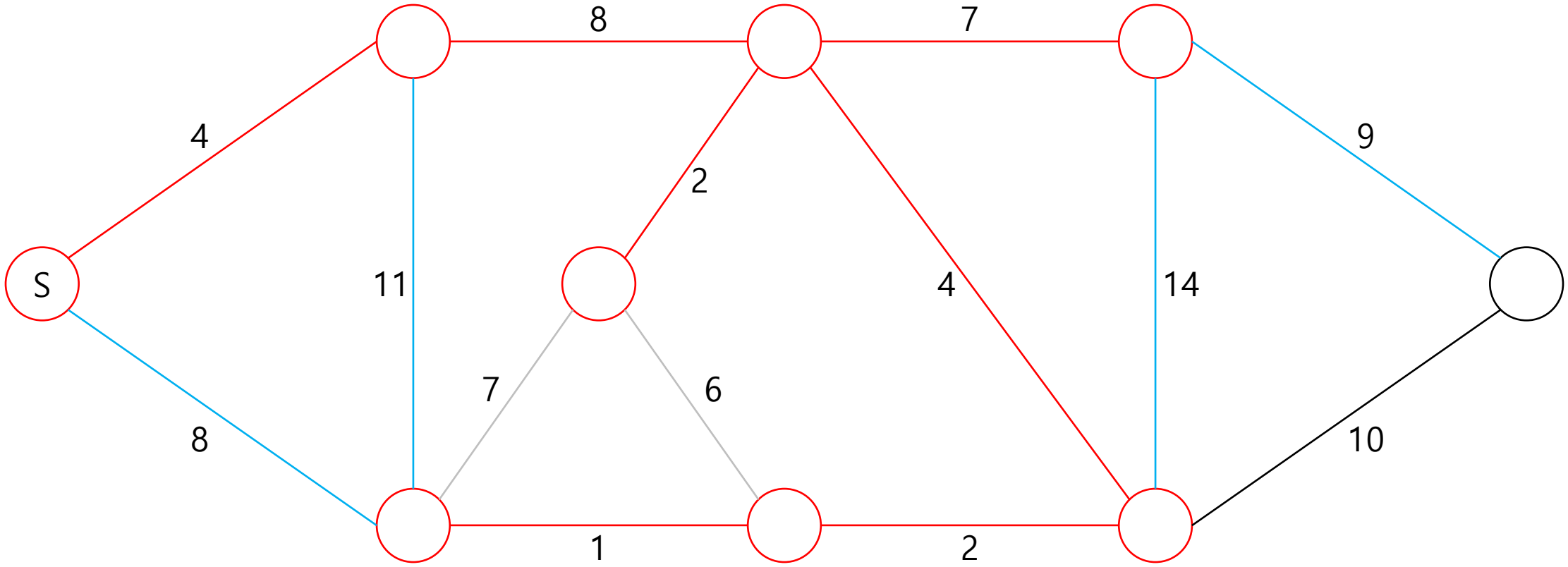
Prim's Algorithm



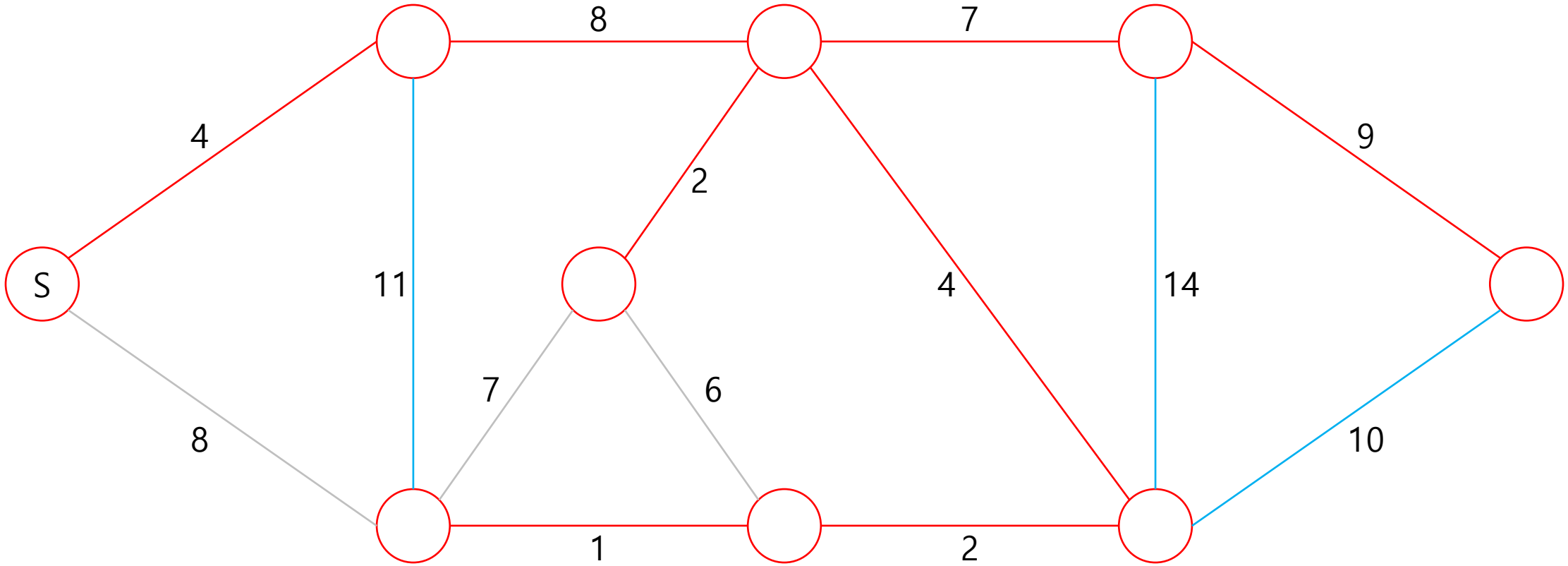
Prim's Algorithm



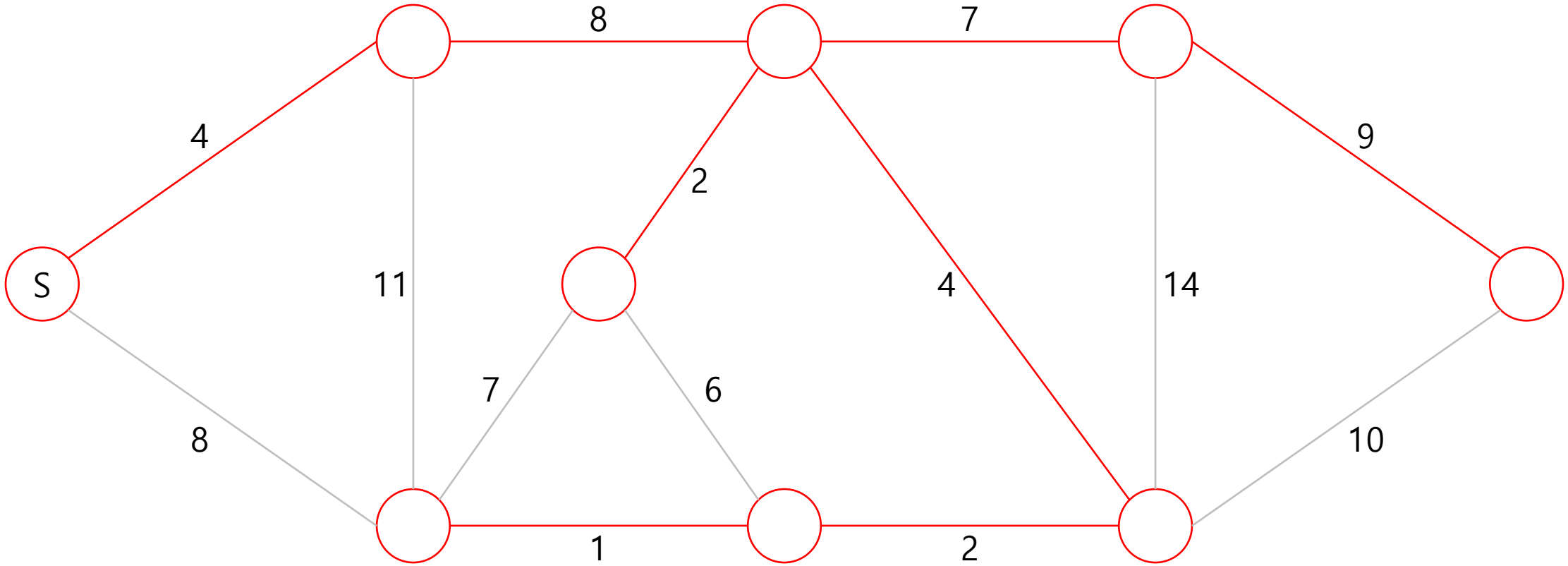
Prim's Algorithm



Prim's Algorithm



Prim's Algorithm



Prim's Algorithm

```cpp
using PII = pair<int, int>;
int N, M, C[5050], D[5050];
vector<PII> G[5050];

int Prim(){
    int ret = 0;
    memset(D, 0x3f, sizeof D);                    // D[i] : i번 정점을 MST에 추가하기 위해 필요한 비용(간선 가중치)
    D[1] = 0;
    for(int iter=1; iter<=N; iter++){
        int v = -1;                               // 아직 MST에 포함되자 않은 정점 중 비용이 최소인 정점 선택
        for(int i=1; i<=N; i++){
            if(C[i]) continue;                    // 이미 MST에 포함된 정점이라면 넘어감
            if(v == -1 || D[v] > D[i]) v = i;
        }
        C[v] = 1; ret += D[v];                    // MST에 넣음
        for(auto [i,w] : G[v]) D[i] = min(D[i], w); // v에서 뻗어나가는 간선 정보 반영
    }
    return ret;
}
```

Prim's Algorithm

- 정당성 증명
  - 수학적 귀납법을 사용해 증명
    - 현재까지 만든 포레스트 F를 포함하는 최소 스패닝 트리 T가 존재할 때
    - F와 V-F를 연결하는 최소 간선 e를 추가한 F+e를 포함하는 최소 스패닝 트리 T'이 존재함을 증명

    - 만약 e가 T에 포함되면 T' = T이다.
    - 그렇지 않은 경우, T+e는 e를 포함하는 단순 사이클 C를 갖는다.
    - 이때 C는 F+e에 속하지 않으면서 F와 V-F를 연결하는 간선 f를 갖는다.
    - e는 F와 V-F를 연결하는 최소 간선이므로 f의 가중치는 e보다 크거나 같아야 한다.
    - 단순 사이클에서 간선 하나를 끊어낸 T-f+e는 트리가 되고, 이것의 가중치는 T 이하이므로
    - T' = T-f+e는 F+e를 포함하는 최소 스패닝 트리이다.

질문?

Prim's Algorithm

- 시간 복잡도
  - V번의 iteration
    - MST에 포함되지 않은 정점 중 비용이 최소인 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 뻗어 나가는 간선은 모두 봐야 하므로 O(deg(v))가 하한임
  - 비용이 최소인 정점을 빠르게 찾을 수 있을까?
    - Min Heap!

Prim's Algorithm

```cpp
using PII = pair<int, int>;
int N, M, C[10101];
vector<PII> G[10101];

int Prim(){
    int ret = 0;
    priority_queue<PII, vector<PII>, greater<>> pq; // {거리, 정점} pair를 저장하는 min heap
    C[1] = 1;                                       // 시작점 S = 1은 MST에 포함
    for(auto [i,w] : G[1]) pq.emplace(w, i);        // S에서 뻗어 나가는 간선들 Heap에 삽입
    while(!pq.empty()){
        auto [c,v] = pq.top(); pq.pop();
        if(C[v]) continue;                          // heap에 같은 정점이 여러 번 들어갈 수 있으니 조심
        C[v] = 1; ret += c;                         // v를 MST에 삽입
        for(auto [i,w] : G[v]) pq.emplace(w, i);    // v에서 뻗어 나가는 간선 정보 반영
    }
    return ret;
}
```

Prim's Algorithm

- 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
  - 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)

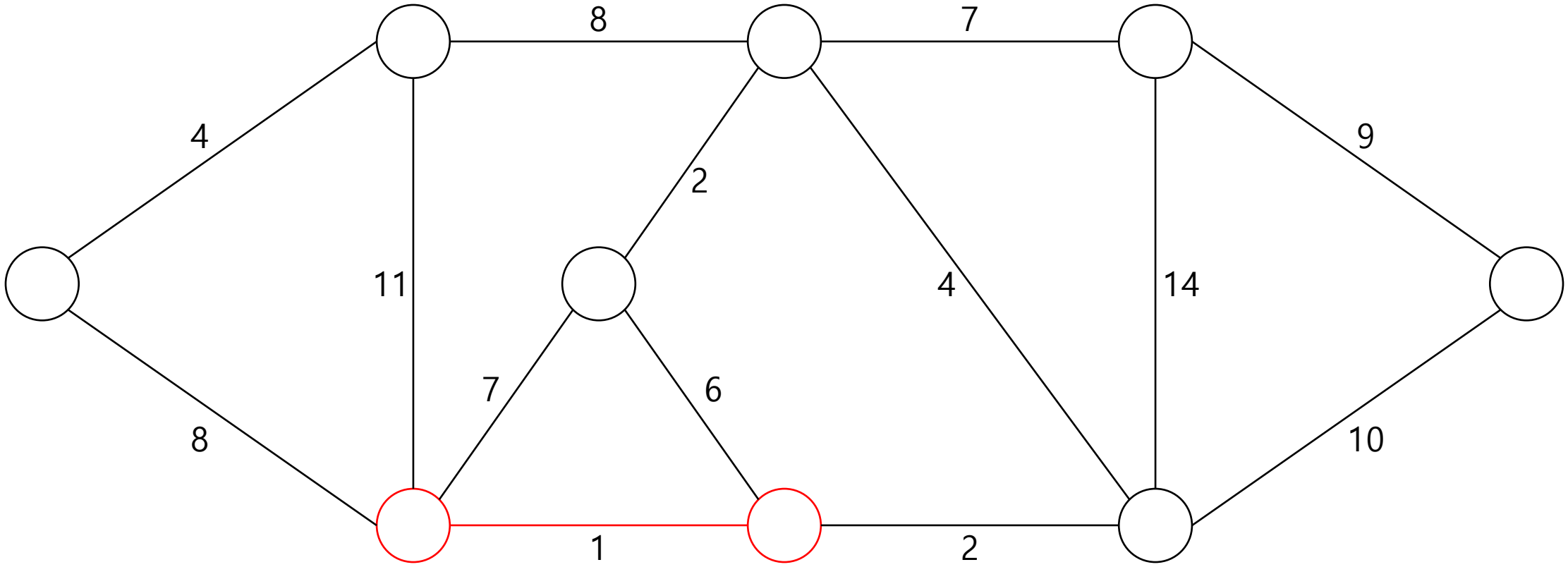
질문?

Kruskal's Algorithm

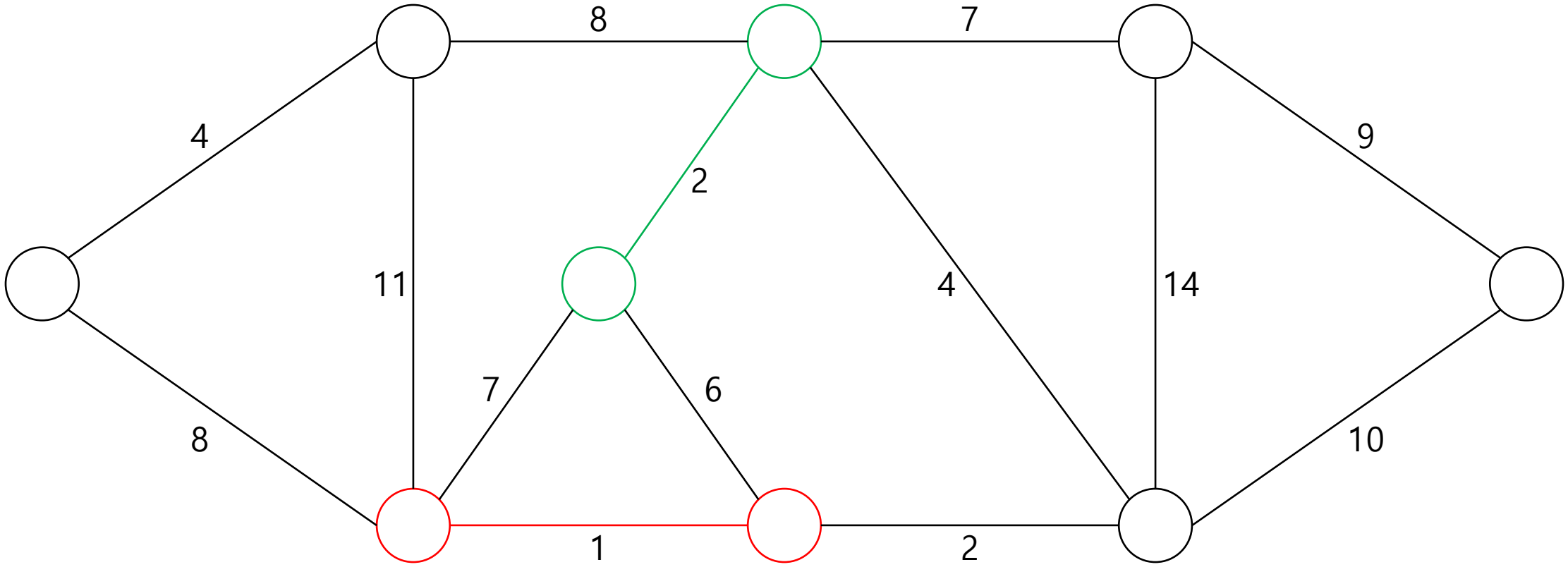
Kruskal's Algorithm

- Kruskal's Algorithm
  - 시간 복잡도 : O(E log E)
  - Spanning Tree에 간선을 하나씩 포함시키는 방식
  - 그리디 기반 알고리즘
    1. 모든 간선을 가중치 오름차순으로 정렬
    2. 가중치가 작은 간선부터 보면서, 사이클이 생기지 않는다면 해당 간선을 MST에 추가
       - Prim's Algorithm과 다르게 알고리즘 진행 도중 트리가 여러 개 있을 수 있음
       - e = (u, v)에서 u와 v가 같은 트리에 있는지 확인해야 함 -> Union-Find

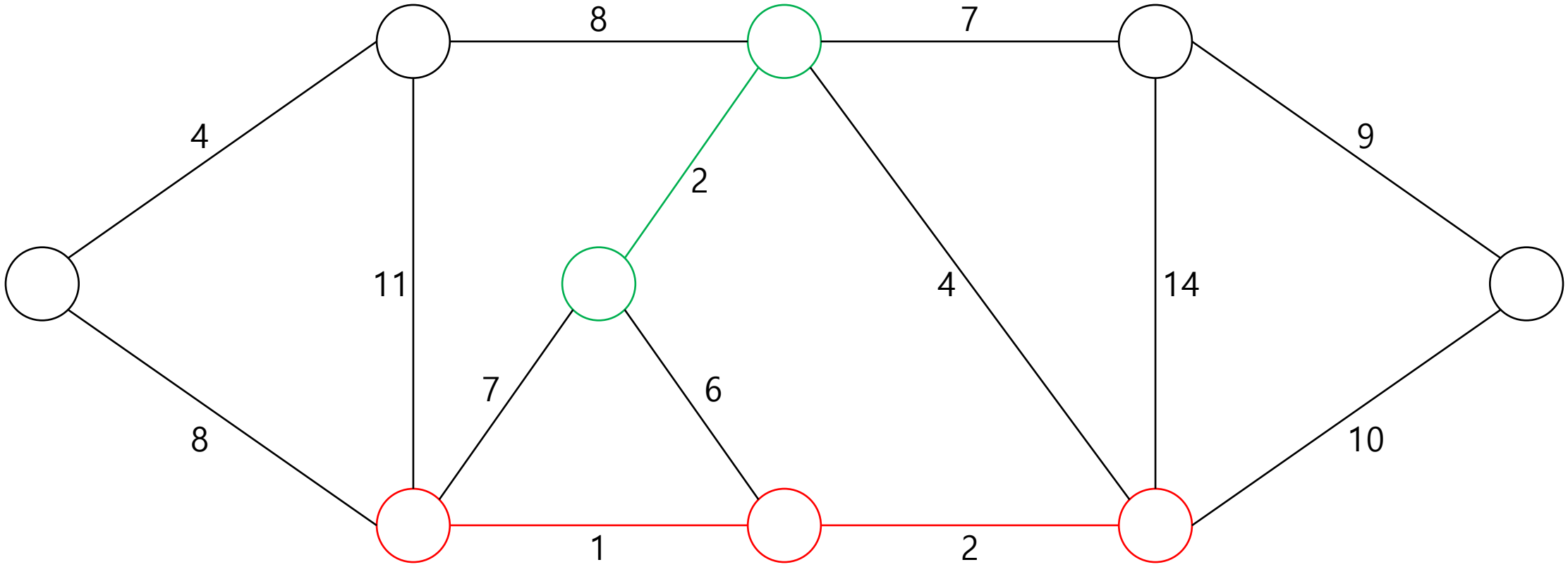
Kruskal's Algorithm



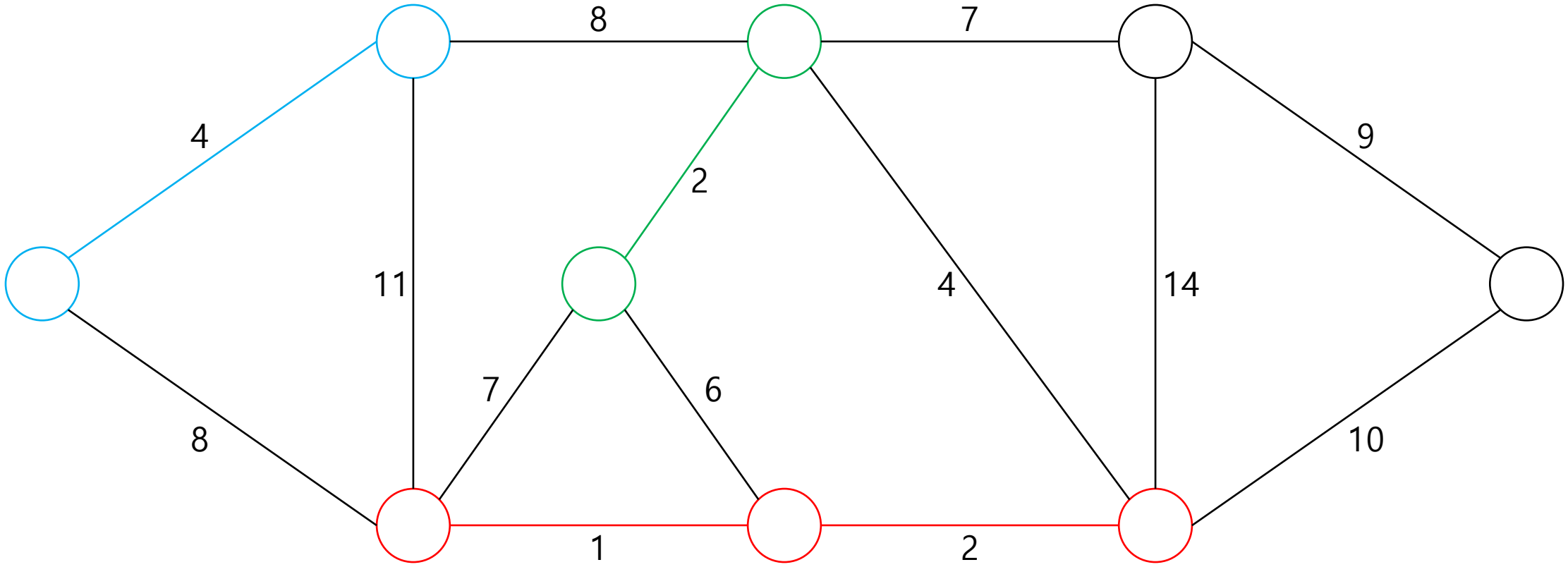
Kruskal's Algorithm



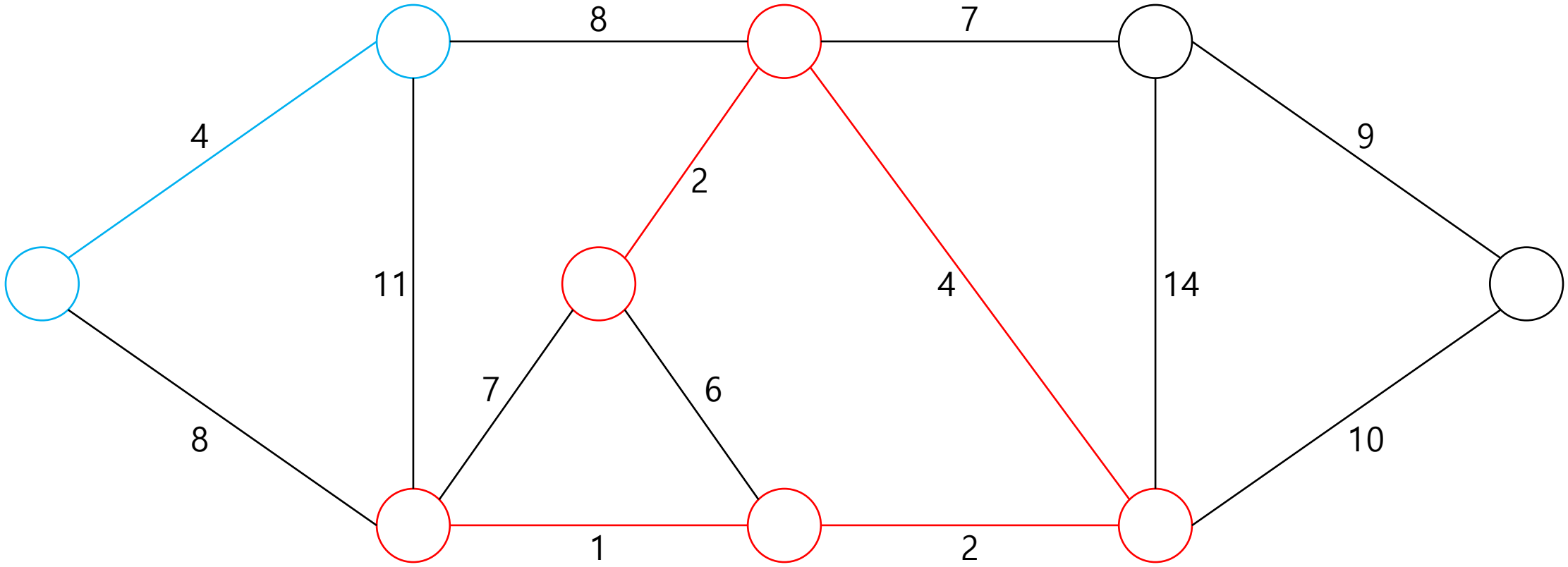
Kruskal's Algorithm



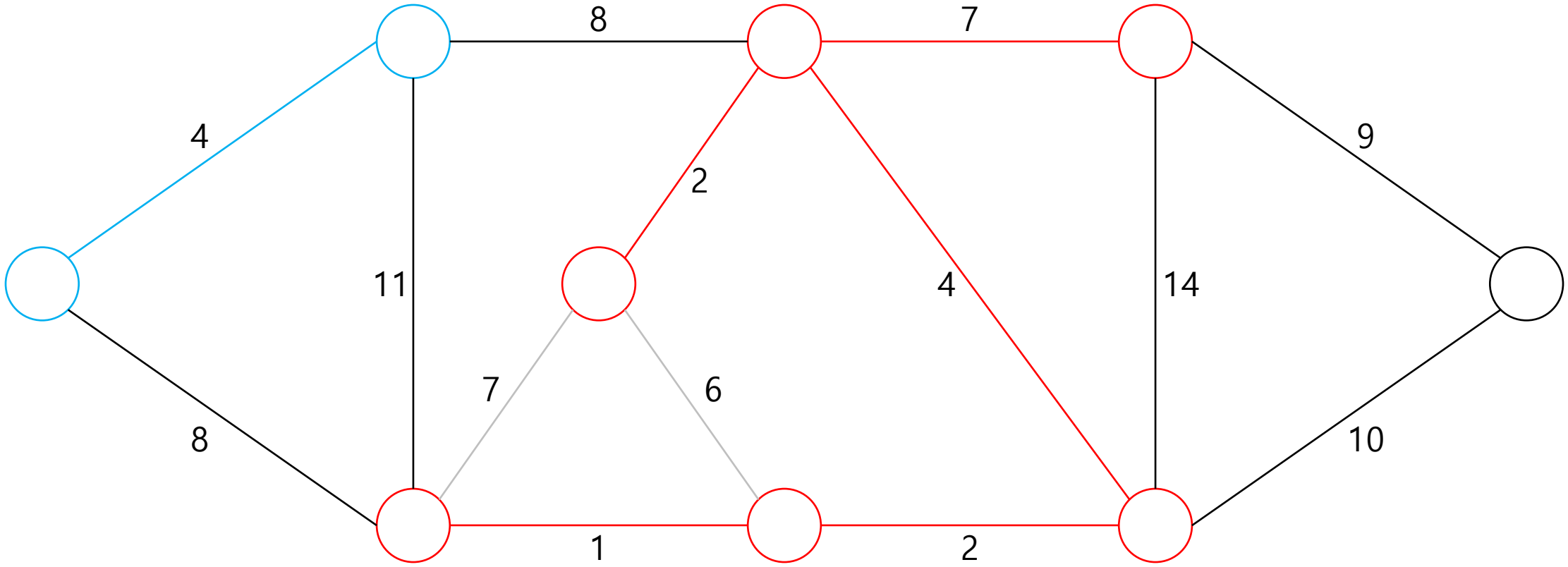
Kruskal's Algorithm



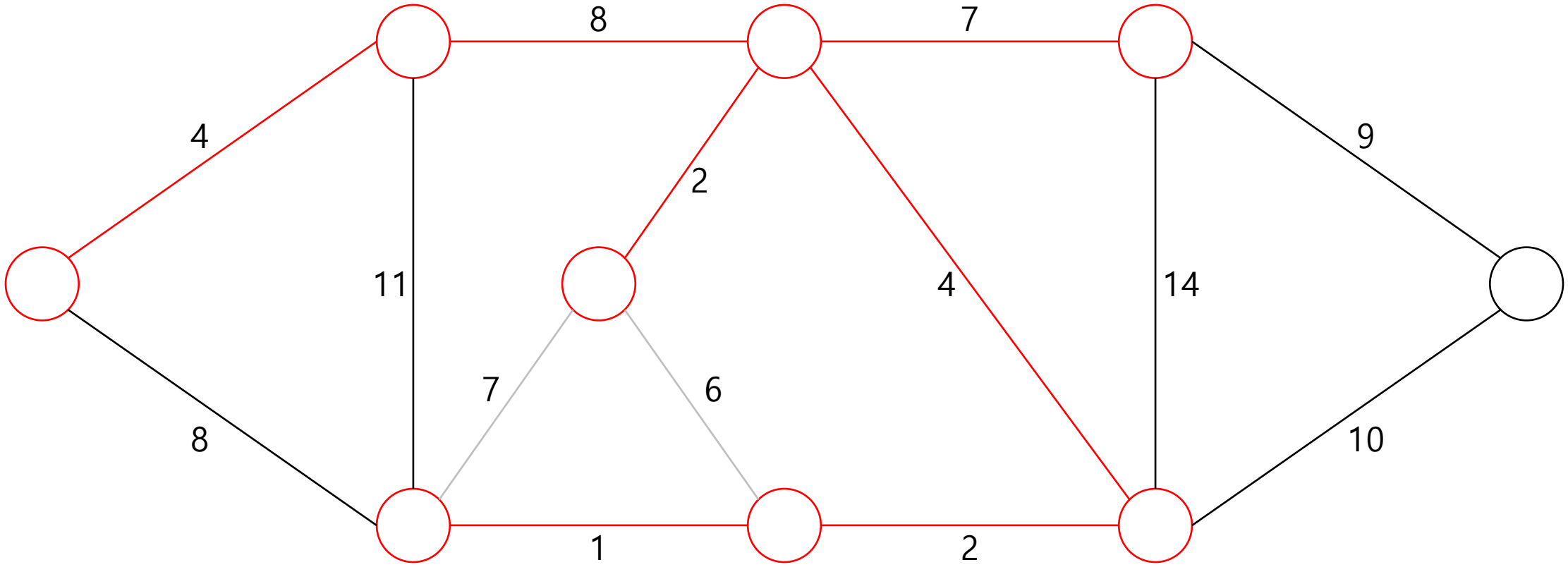
Kruskal's Algorithm



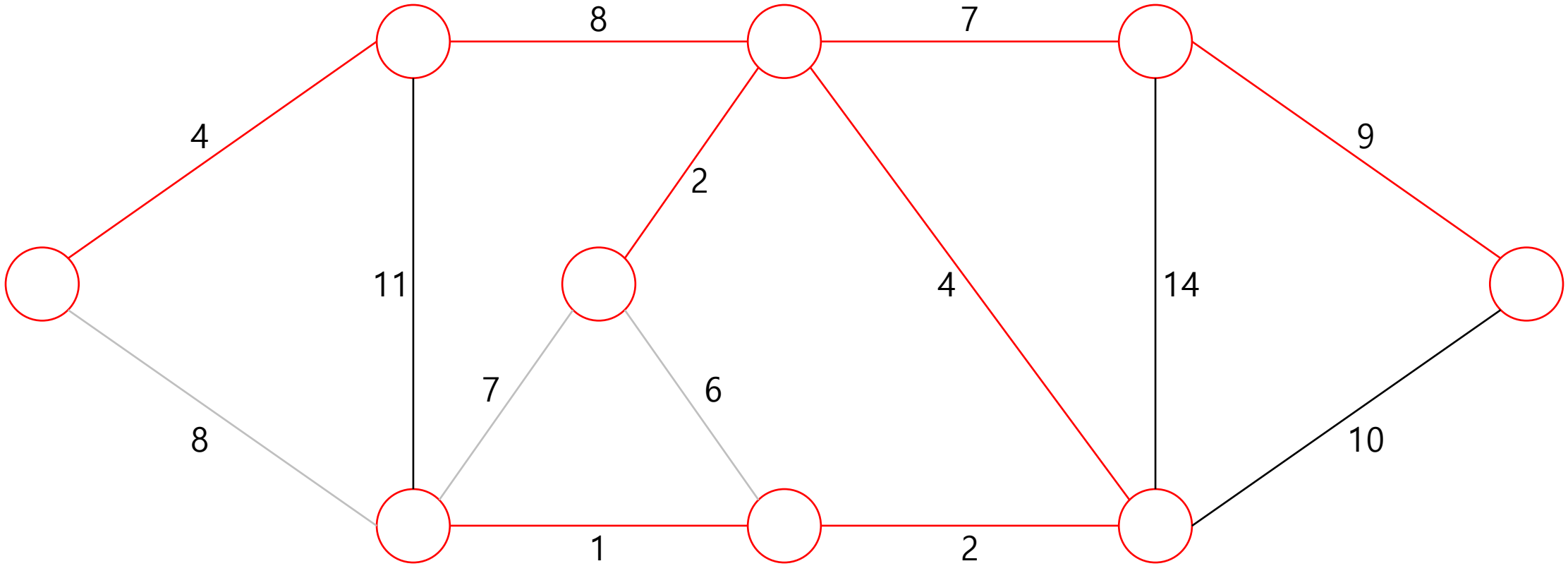
Kruskal's Algorithm



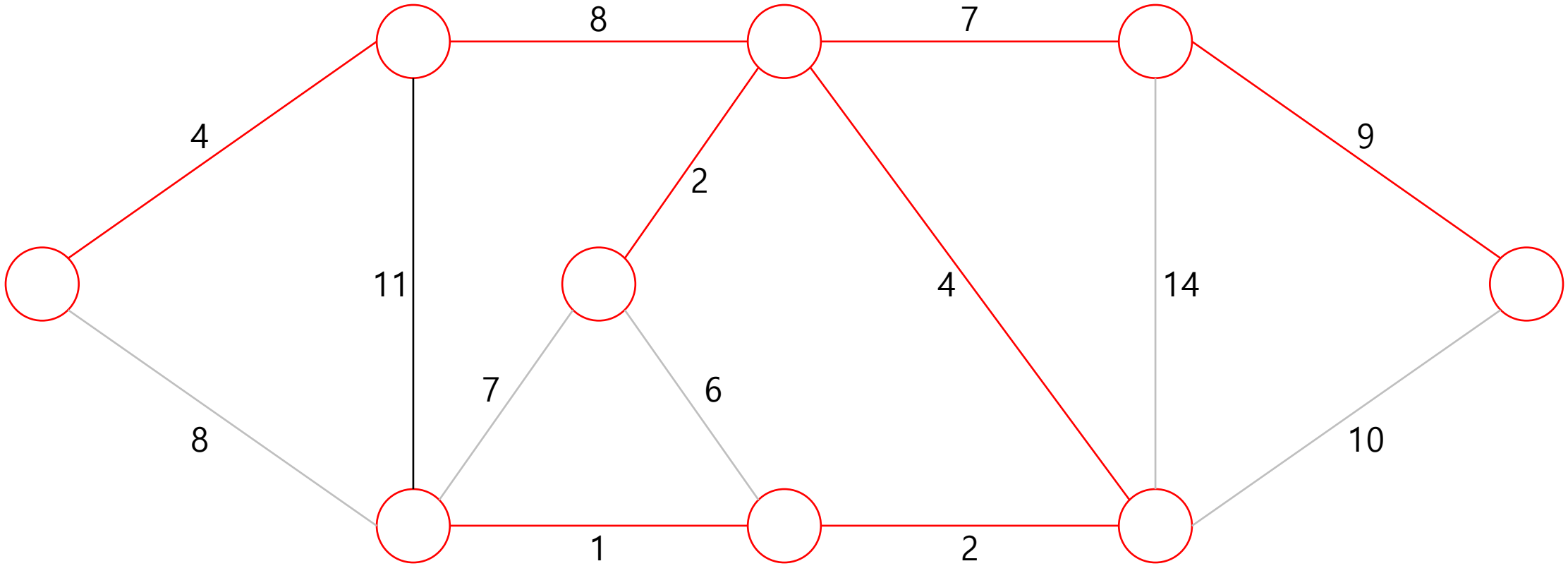
Kruskal's Algorithm



Kruskal's Algorithm



Kruskal's Algorithm



Kruskal's Algorithm

```cpp
struct Edge{
    int s, e, w;
    Edge() = default;
    Edge(int s, int e, int w) : s(s), e(e), w(w) {}
    bool operator < (const Edge &t) const { return w < t.w; }
};

int N, M, P[10101];
vector<Edge> E;

int Find(int v){ return v == P[v] ? v : P[v] = Find(P[v]); }
bool Union(int u, int v){ return Find(u) != Find(v) && (P[P[u]]=P[v], true); }

int Kruskal(){
    int ret = 0;
    for(int i=1; i<=N; i++) P[i] = i;
    sort(E.begin(), E.end());
    for(auto [s,e,w] : E) if(Union(s, e)) ret += w;
    return ret;
}
```

Kruskal's Algorithm

- 정당성 증명
  - 수학적 귀납법을 사용해 증명
    - 현재까지 만든 포레스트 F를 포함하는 최소 스패닝 트리 T가 존재할 때
    - 사이클을 만들지 않는 최소 간선 e를 추가한 F+e를 포함하는 최소 스패닝 트리 T'이 존재함을 증명

    - 만약 e가 T에 포함되면 T' = T이다.
    - 그렇지 않은 경우, T+e는 e를 포함하는 단순 사이클 C를 갖는다.
    - 이때 C는 F+e에 속하지 않으면서 F와 V-F를 연결하는 간선 f를 갖는다.
    - f는 아직 알고리즘 과정에서 고려되지 않았으므로 e보다 가중치가 크거나 같아야 한다.
    - 단순 사이클에서 간선 하나를 끊어낸 T-f+e는 트리가 되고, 이것의 가중치는 T 이하이므로
    - T' = T-f+e는 F+e를 포함하는 최소 스패닝 트리이다.

Kruskal's Algorithm

- 시간 복잡도
  - 간선 정렬 : O(E log E)
  - Union-Find 연산 : O(log V)짜리 연산을 O(E)번 수행
  - 시간 복잡도 : O(E log E)

질문?

Minimum Spanning Tree

- Prim vs Kruskal
  - 구현: Kruskal이 쉬움
  - 속도: Kruskal이 빠름
  - Prim: 왜 씀?

  - 간선의 가중치가 정점 번호에 대한 수식으로 표현되고, 메모리 제한이 작은 경우
    - $O(V^2)$짜리 Prim's Algorithm은 공간 복잡도 O(V)
    - Kruskal's Algorithm은 공간 복잡도 O(V+E)
      - 모든 간선을 다 구해서 정렬해야 함
    - BOJ 20390 완전그래프의 최소 스패닝 트리

Minimum Spanning Tree

- 응용
  - 최대 신장 트리
    - 가중치에 -1 곱하고 Kruskal's Algorithm

  - u에서 v로 가는 경로 상의 가중치 최댓값을 최소화
    - MST에서 경로 최댓값 쿼리
    - LCA(Sparse Table)이나 HLD 같은 걸로 처리하면 됨

질문?