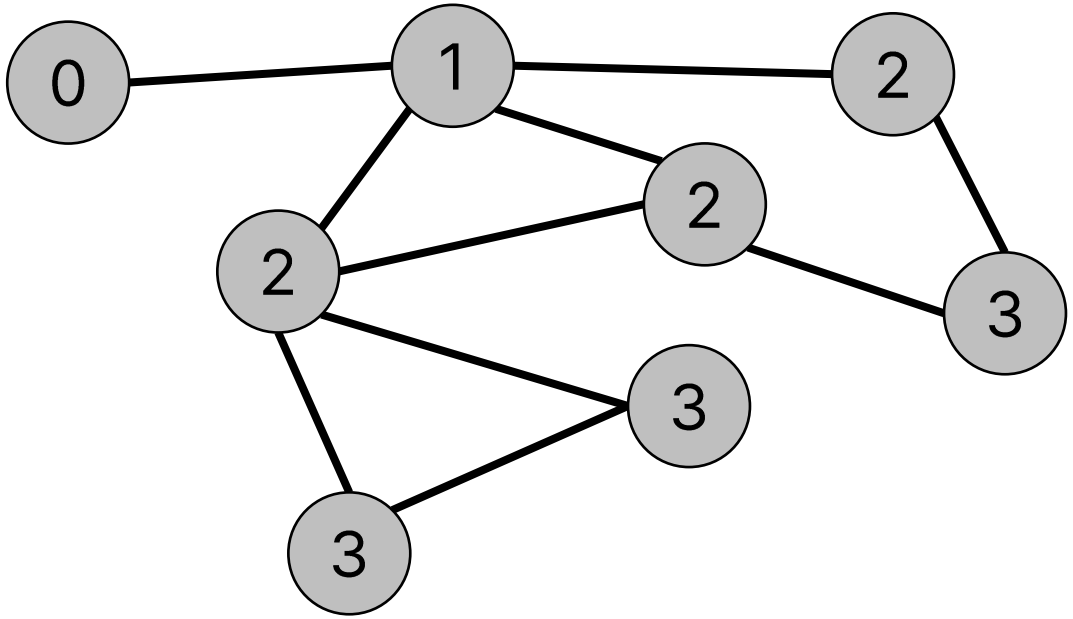
Shortest Path

Shortest Path (최단 거리)

- 그래프에서 한 정점으로부터 다른 정점까지 도달하는 데 거쳐야 하는 **간선의 최소 개수**
- 너비 우선 탐색(BFS)를 사용하면 $O(V + E)$에 해결 가능



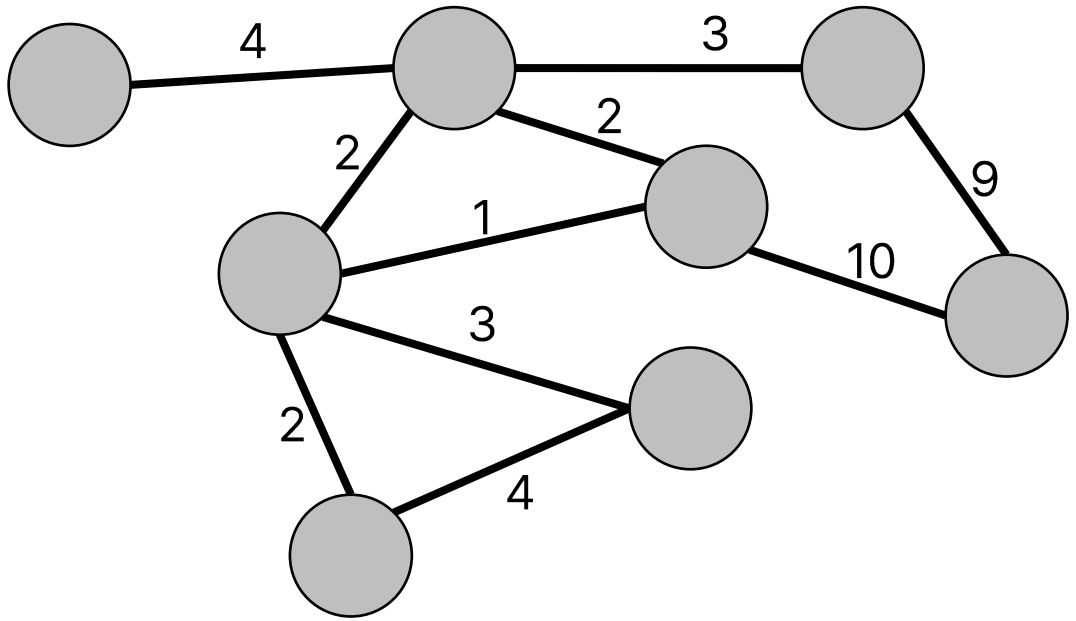
SSSP & APSP

- Single-Source Shortest Path: 한 점으로부터 다른 모든 점에 대한 최단거리 구하기
  - Dijkstra, Bellman-Ford
- All-Pair Shortest Path: 서로 다른 모든 두 점에 대한 최단거리 구하기
  - ~~SSSP 알고리즘 V번 돌리기~~, Floyd-Warshall

- 이외에도 그래프의 개형(양방향/단방향, 트리, DAG 등)에 따라서 여러 방법을 사용

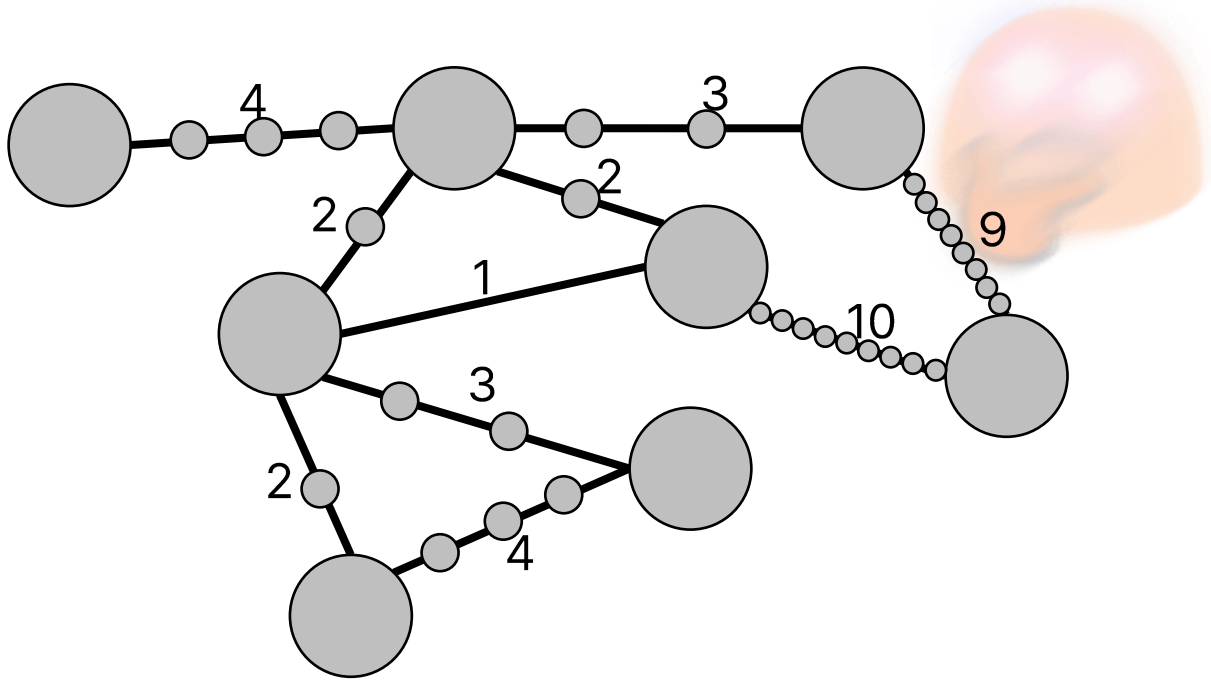
Shortest Path in weighted graph

- 간선에 가중치가 존재한다면?



Shortest Path in weighted graph

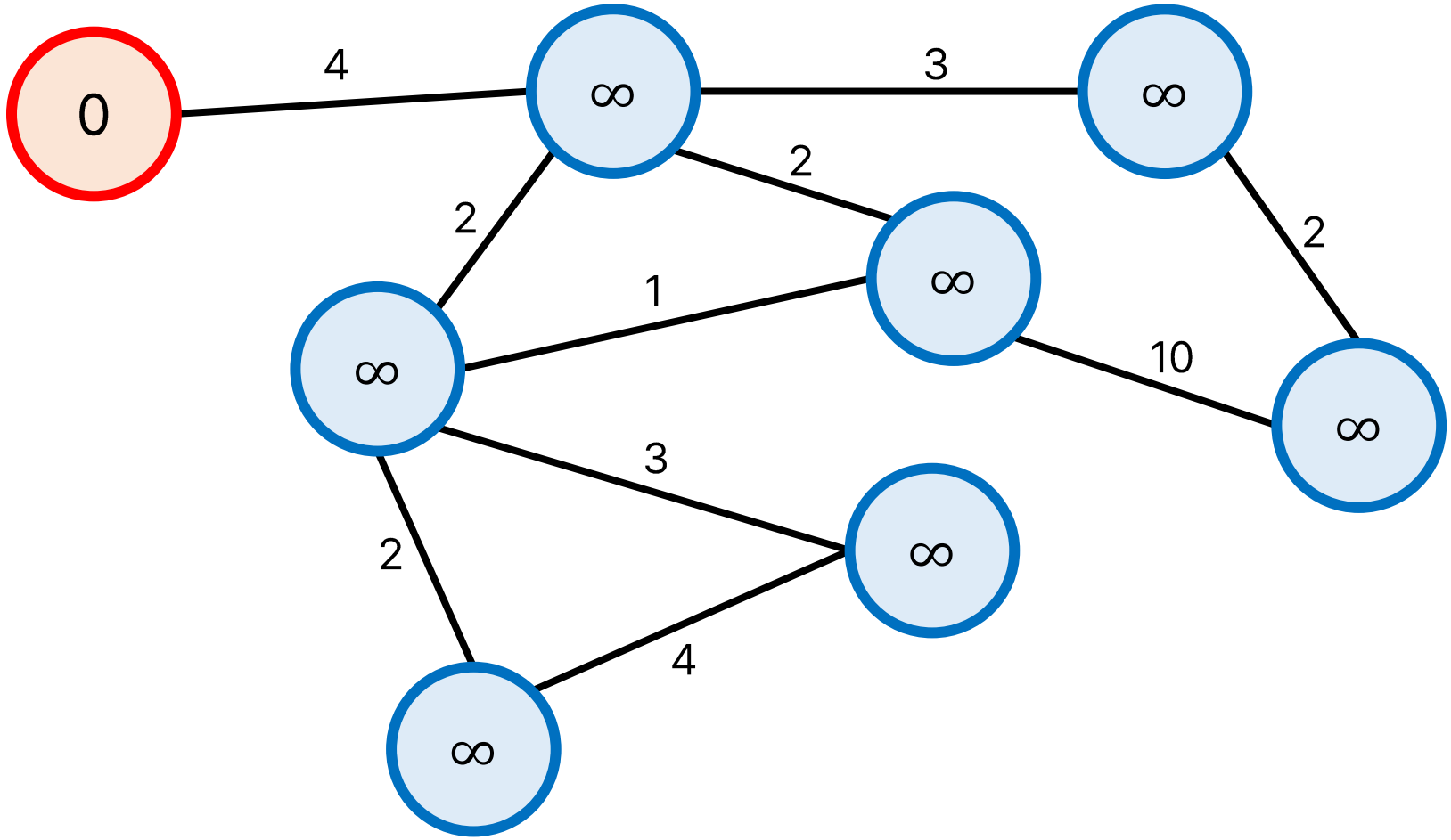
- 가중치가 k라면, 간선 사이에 노드를 $k-1$개 놓은 뒤 BFS
- 구현도 까다롭고, 가중치가 커진다면 더 오래 걸린다



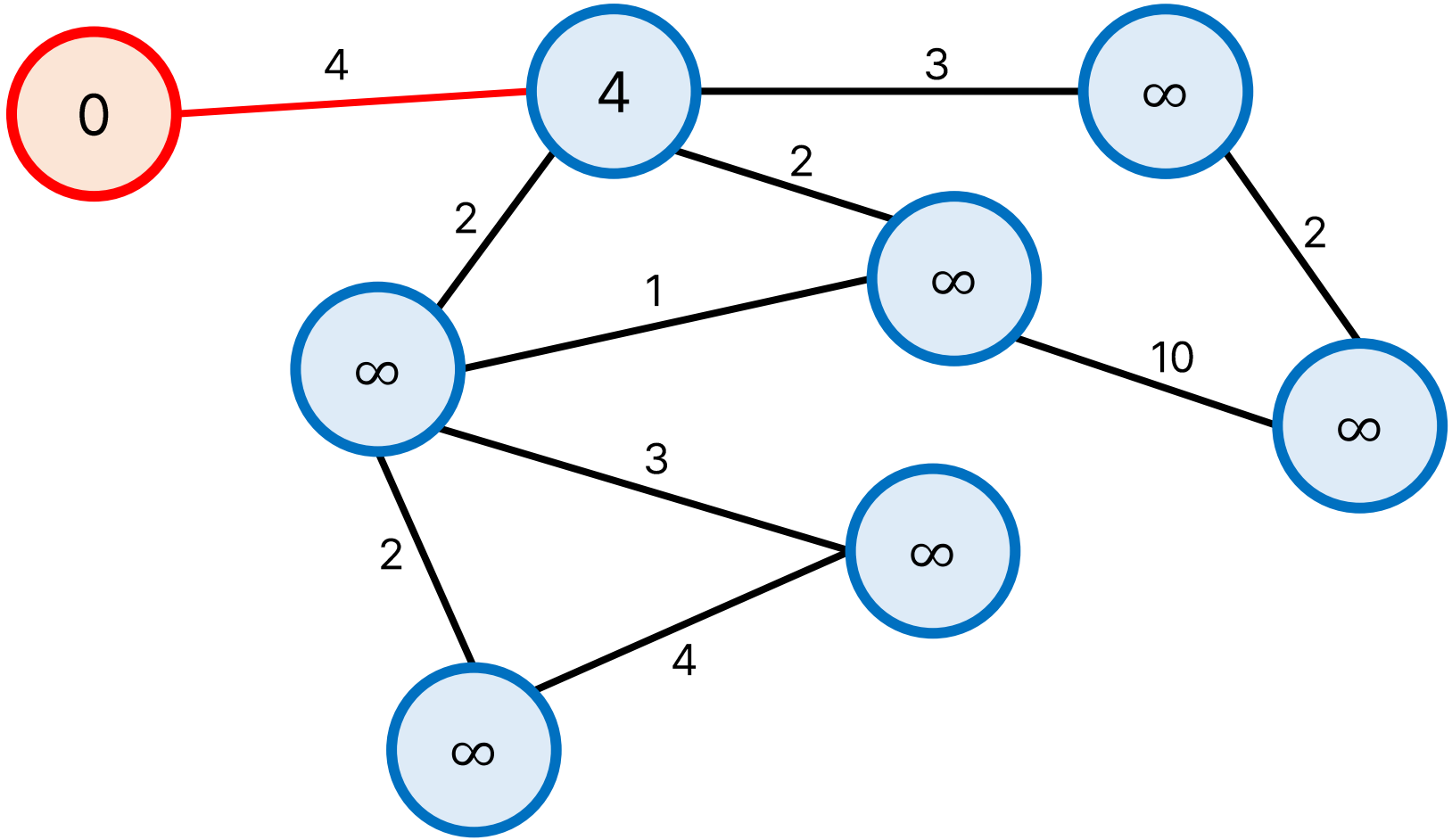
Dijkstra's Algorithm: Greedy Approach

- 가중치가 0 이상인 그래프에서 최단 거리(SSSP)를 구하는 알고리즘
- 정점의 종류를 두 개로 나눈다 (최단거리가 **확정된 정점**, **확정되지 않은 정점**)
- 시작점의 거리는 0, 다른 정점들은 ∞로 초기화 ($R = \{v_0\}, dist[v_0] = 0$)
  - 0번 정점에서 시작, $\boldsymbol{R}$은 최단거리가 확정된 정점의 집합
- $|R| < V$인 동안, $u \notin R$이면서 $dist[u]$가 최소인 u를 고른 뒤, $R = R \cup \{u\}$
- u와 인접하고 $w \notin R$인 $\boldsymbol{w}$에 대해서, $dist[w] = \min\left(dist[w], dist[u] + e_{u,w}\right)$로 갱신

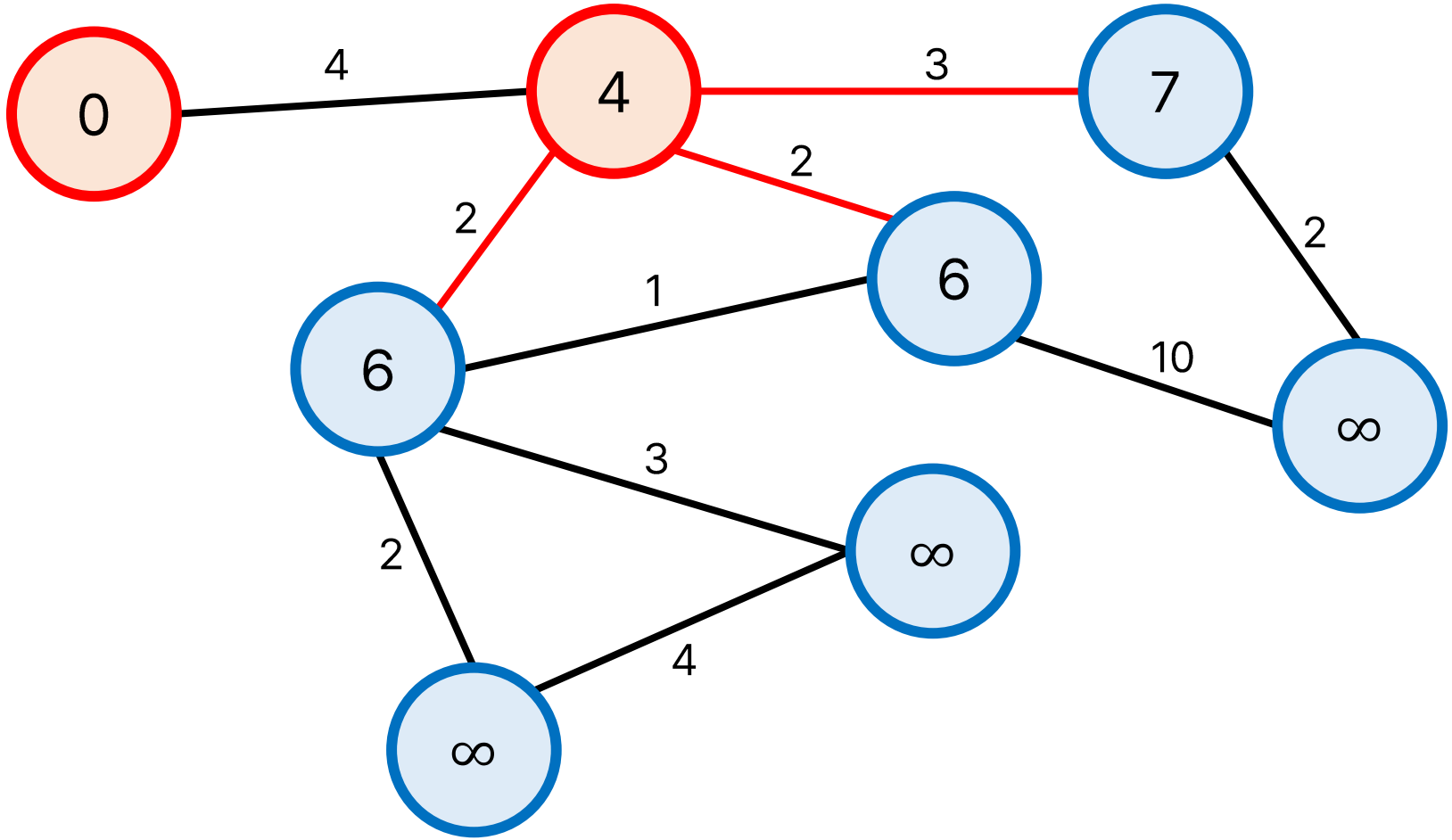
Dijkstra's Algorithm



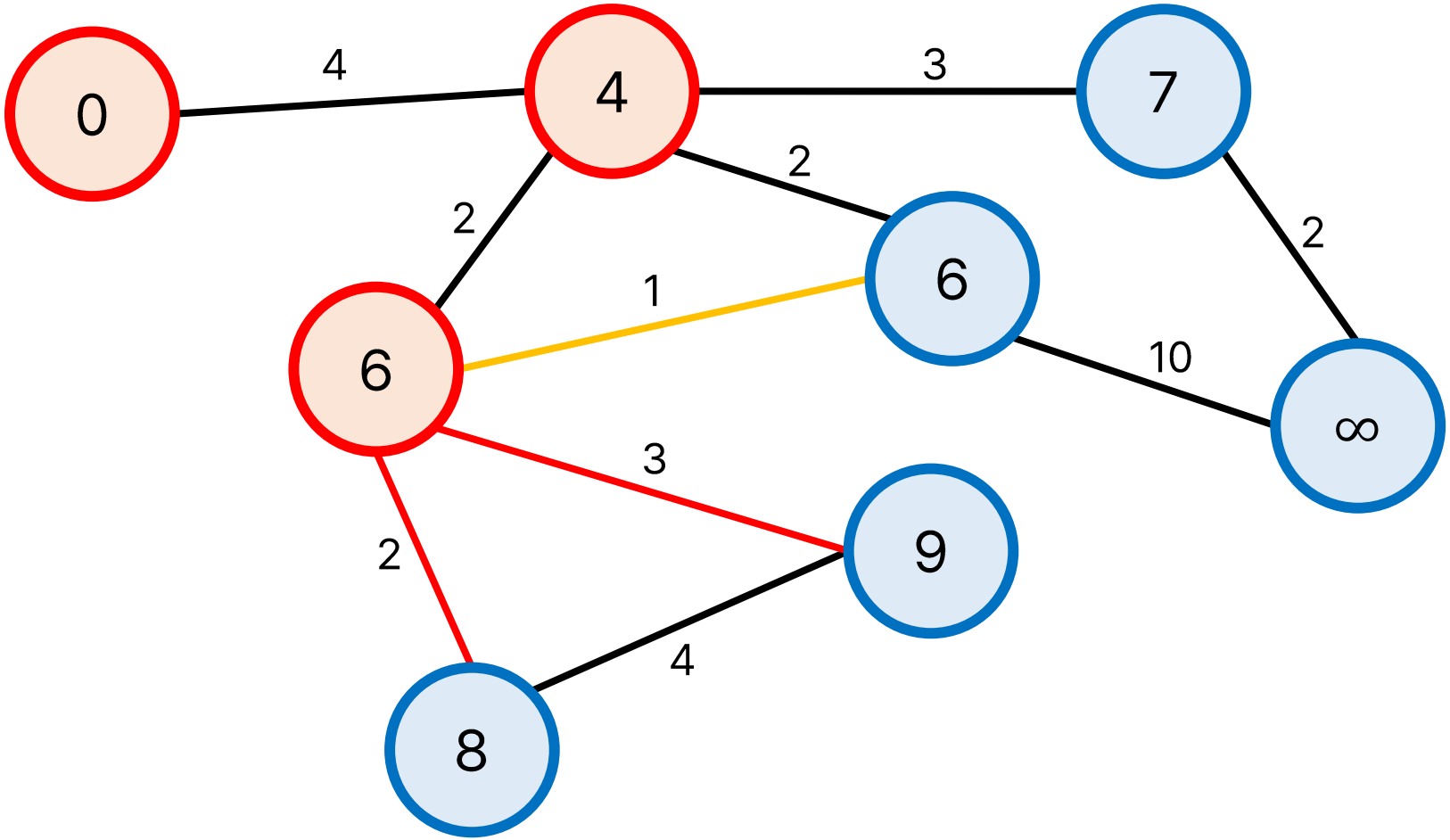
Dijkstra's Algorithm



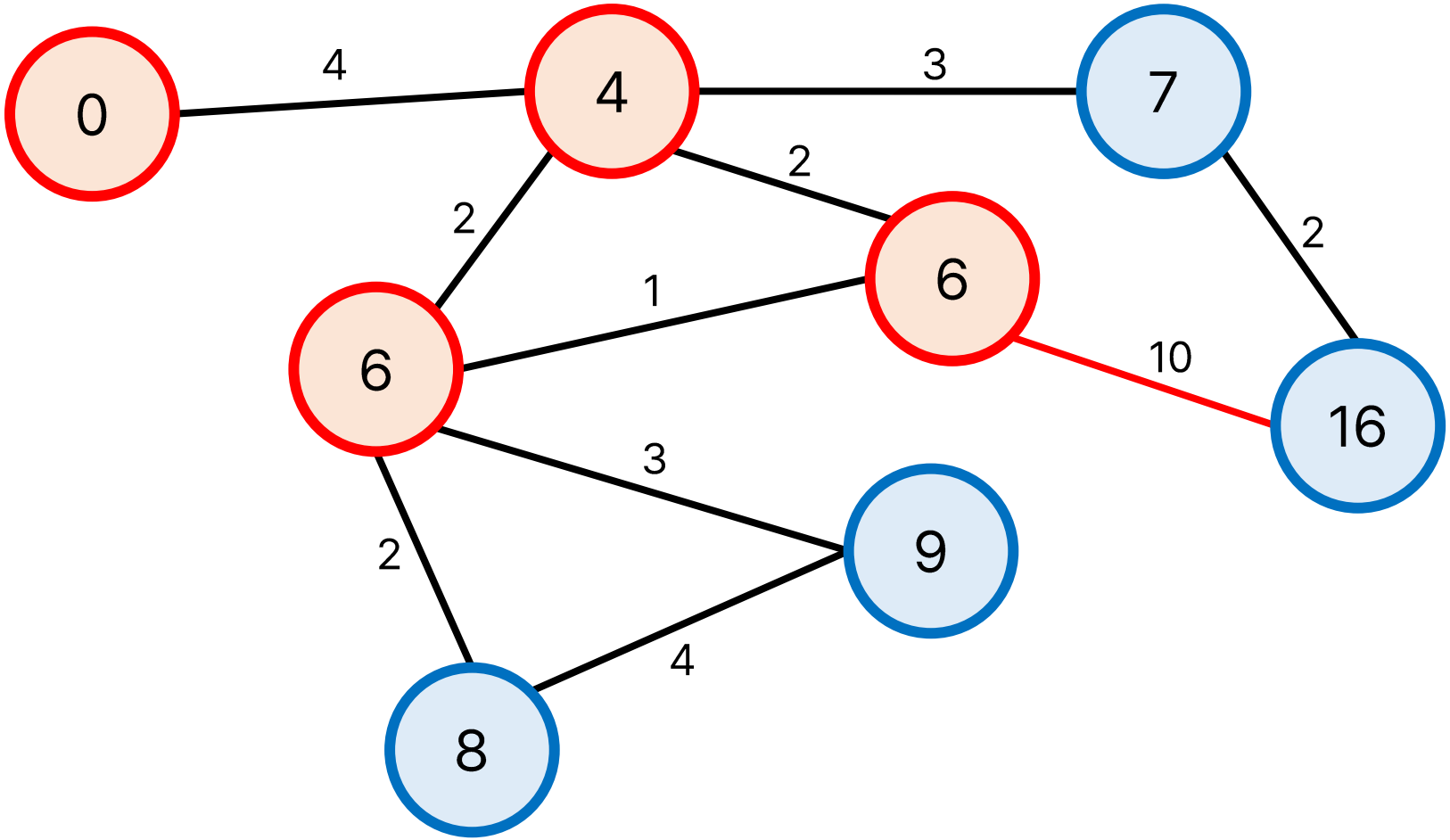
Dijkstra's Algorithm



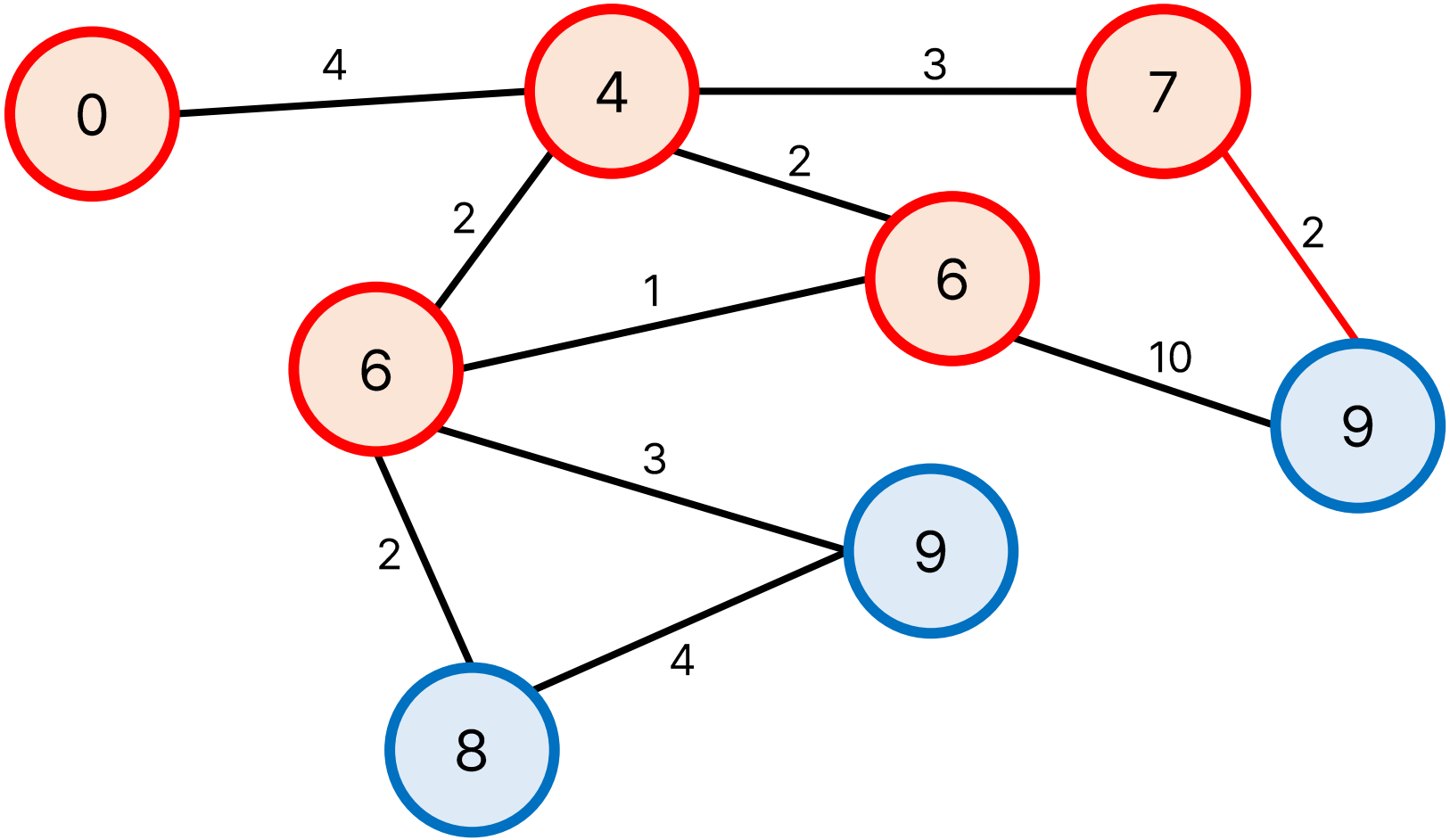
Dijkstra's Algorithm



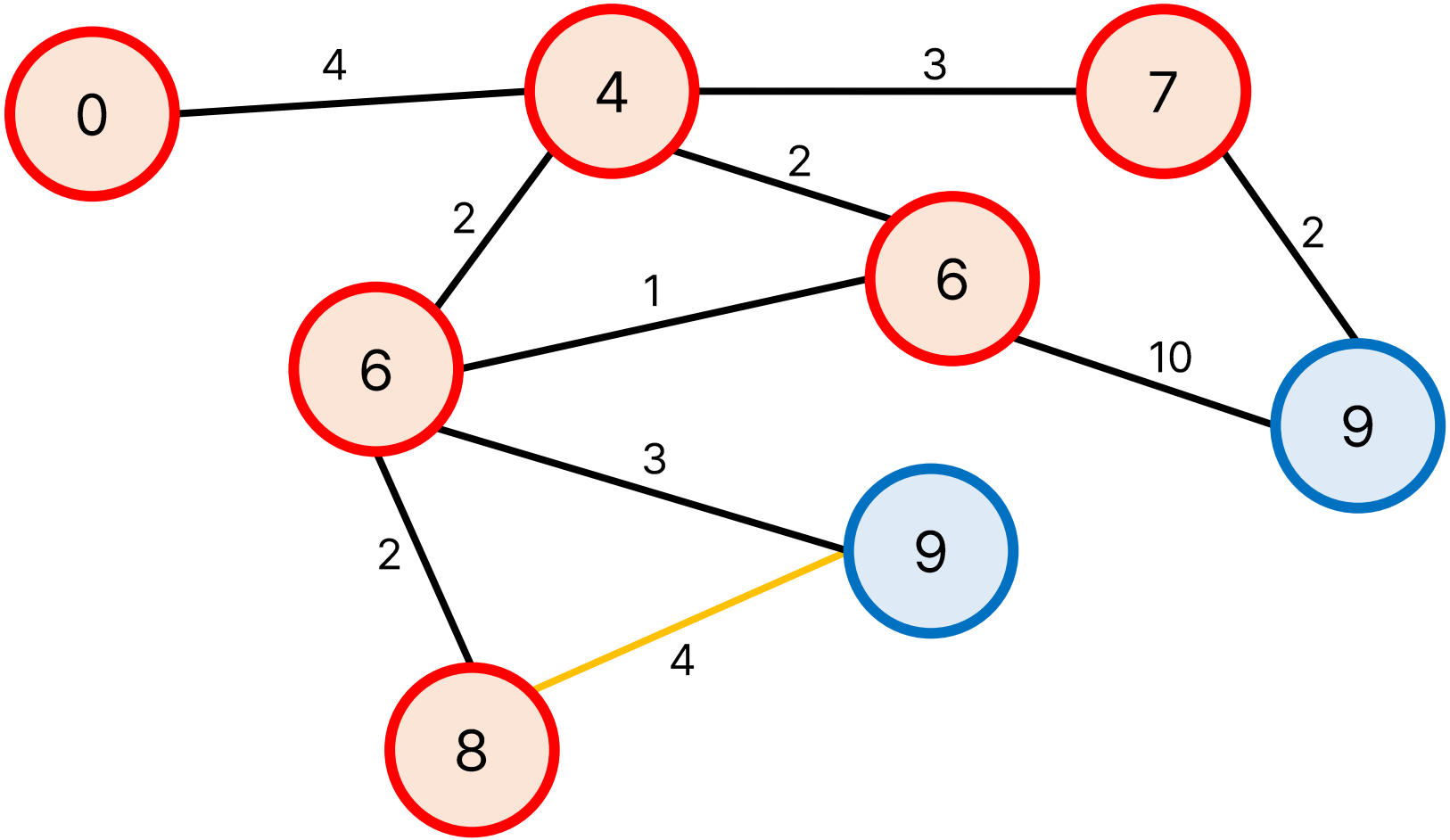
Dijkstra's Algorithm



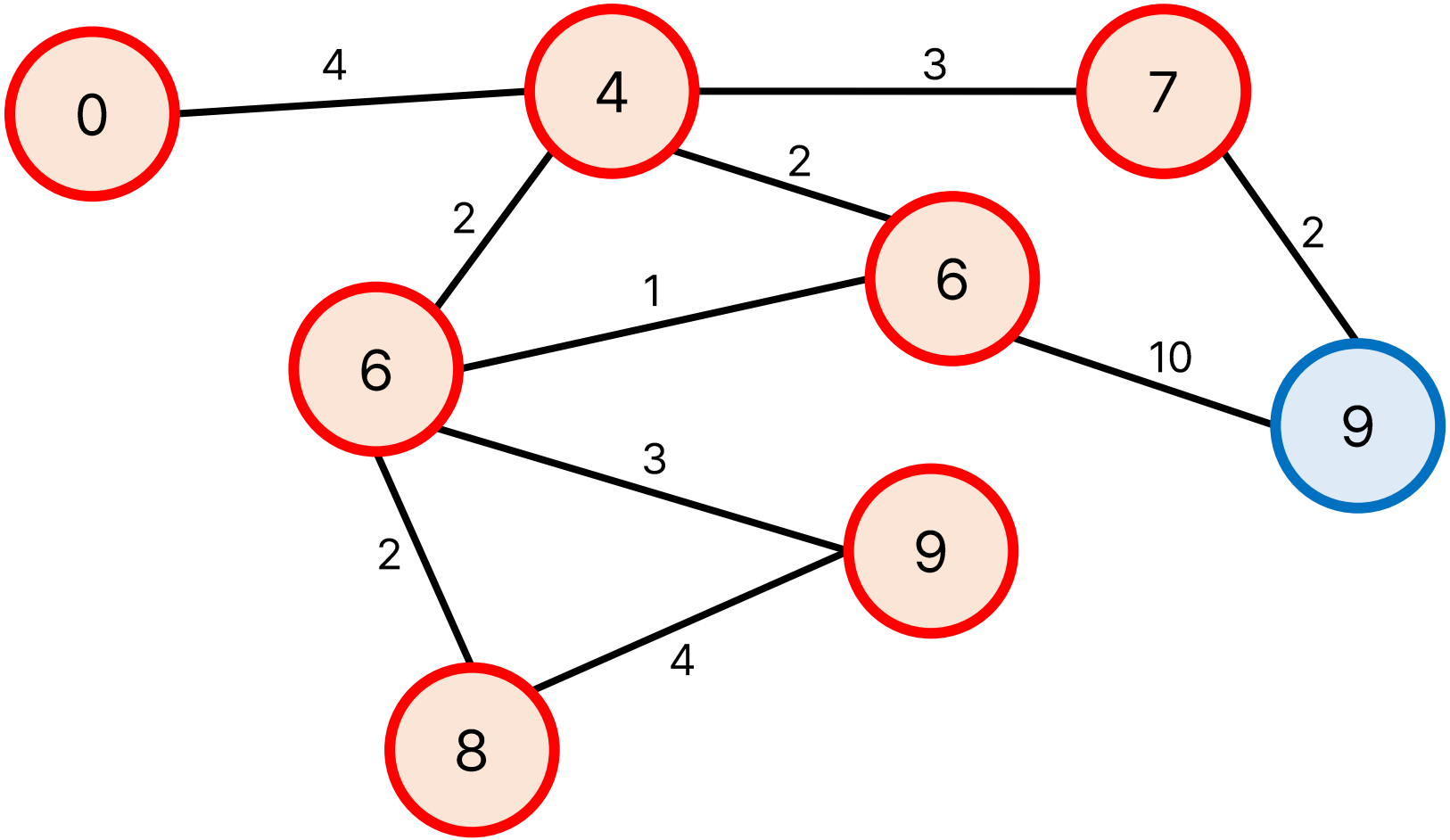
Dijkstra's Algorithm



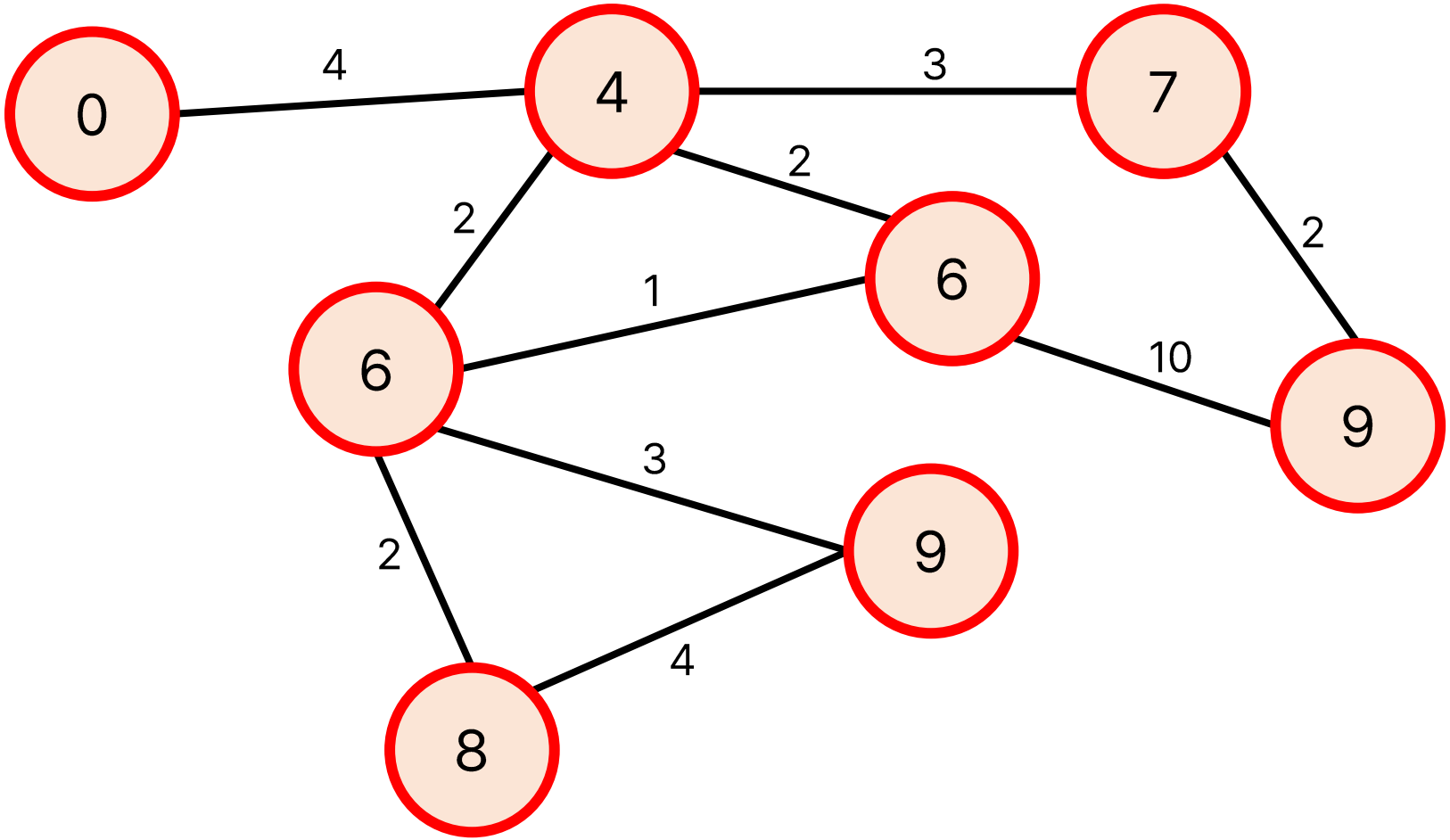
Dijkstra's Algorithm



Dijkstra's Algorithm



Dijkstra's Algorithm

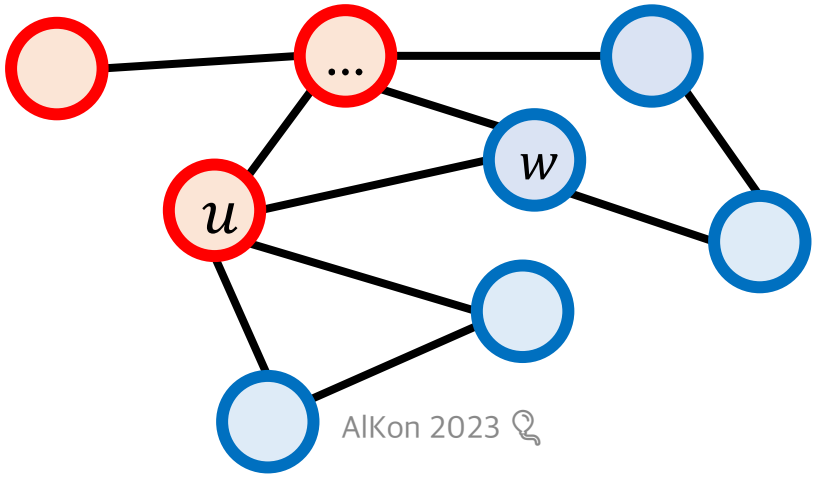


Dijkstra's Algorithm: 정당성 증명

- $u \notin R$이면서 $dist[u]$가 최소인 $\boldsymbol{u}$를 고른 뒤, $\boldsymbol{R} = R \cup \{u\}$
- $dist[u]$가 최단거리가 아니라면, $v \notin R$인 $\boldsymbol{v}$를 거치는 더 좋은 최단경로가 존재함.
- $dist[u] = dist[v] + e_{u,v}$ 이고, 모든 가중치는 0 이상이므로 $dist[v] \leq dist[u]$
- $dist[v] = dist[u]$ 인 경우, 바꿔도 답
- $dist[v] < dist[u]$ 인 경우, $u \notin R$이면서 $dist[u]$가 최소인 $\boldsymbol{u}$를 고르므로 모순이다.

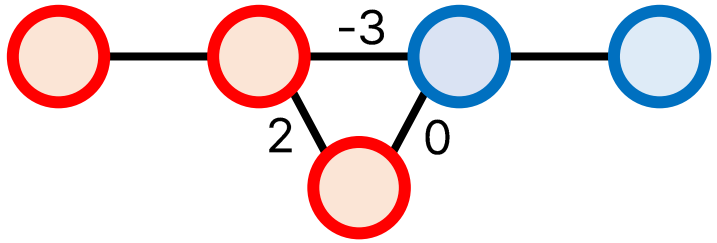
Dijkstra's Algorithm: 정당성 증명

- **u와 인접하고** $w \notin R$인 w에 대해서, $dist[w] = \min(dist[w], dist[u] + e_{u,w})$로 갱신
- 0 – … - u - … - w 와 같은 경우는 고려하지 않아도 되는가?
- u를 포함하지 않는 Source -> … 경로가 존재함, u를 거쳐가면 거리가 증가하게 된다.
- 이전에 확정될 때 고려된 적 있기 때문에 지금 확정된 u와 인접한 노드만 확인하면 된다.



음수 가중치 간선이 존재하면 안 되는 이유

- Dijkstra Algorithm: **그리디** 전략 사용, 하나의 전략을 알고리즘 전체에서 사용
- 한 번 확정한 최단거리는 **다시 갱신하지 않음**
- 그래프가 음의 사이클을 가지고 있는 경우 (간선의 가중치의 합이 음수인 사이클)
- 간선을 무한히 돌아 **가중치를 계속해서 낮출 수 있음**
- Bellman-Ford Algorithm: 음수 가중치에서도 작동, $O(VE)$



Dijkstra's Algorithm: Implementation

- $|R| < V$인 동안, $\boldsymbol{u \notin R}$이**면서 $\boldsymbol{dist[u]}$가 최소인 $\boldsymbol{u}$를 고른 뒤**, $R = R \cup \{u\}$
  - 최소인 것을 고르는 것 $O(V)$, 확정하는 연산 $O(1)$
  - 최소인 것을 고르는 것 $O(logV)$: Priority Queue
- **$\boldsymbol{u}$와 인접하고 $\boldsymbol{w \notin R}$인 $\boldsymbol{w}$**에 대해서, $dist[w] = \min\left(dist[w], dist[u] + e_{u,w}\right)$로 갱신
  - 최대 E번 갱신함, 각 노드를 한 번씩만 보고, 해당 노드에 붙어있는 간선만큼 확인해야 하므로
- Naïve implementation: $O(VE)$, Using Prority queue: $O(ElogV)$

Dijkstra's Algorithm: Implementation

```cpp
int dijkstra(int start){
    memset(dist, INF, sizeof dist);

    pq.emplace(0, start);
    dist[start] = 0;

    while(!pq.empty()){
        auto [dist_here, here] = pq.top(); pq.pop();
        dist_here *= -1;

        if (dist_here > dist[here]) continue;

        for(auto& [there, cost_there] : g[here]){
            int dist_there = dist_here + cost_there;
            if (dist_there < dist[there]){
                dist[there] = dist_there;
                pq.emplace(-dist_there, there);
            }
        }
    }
}
```

구현할 때 자주 하는 실수

- INF 값이 충분히 크지 않은 경우
- 이미 방문한 정점을 재방문하는 경우
- 우선순위 큐에 지금까지의 누적 거리가 아닌, 간선의 가중치를 넣는 경우
- 등등...

최단경로 BOJ 1753

- **방향 그래프**가 주어질 때, 시작점으로부터 다른 **모든 정점**에 대한 최단 경로를 구하기
- Single-Source Shortest Path
- Dijkstra를 직접 구현해 보고 나의 코드를 확인하자!

최소비용 구하기 2 BOJ 11779

- 한 정점으로부터 다른 정점까지의 최단거리를 구해야 한다
- 단, 최단경로를 이루는 정점을 순서대로 나열해야 한다, Dijkstra의 역추적
- 어떻게 할 수 있을까?

최소비용 구하기 2 BOJ 11779

- Dijkstra 를 구할 때, 한 가지 특징을 떠올리자: **거리가 확정되면 이후에 변하지 않는다**
- 거리가 확정되는 순간 어떤 정점으로부터 업데이트되었는지 기록해 보자
- BFS에서의 역추적을 떠올려 보고, Dijkstra에도 동일하게 적용할 수 있을지 고민하자

파티 BOJ 1238

- 단방향 그래프가 주어질 때, 특정 정점에서 파티가 열린다
- 해당 정점으로 다른 모든 정점이 최단 경로를 통해 도착한다
- 이후 다시 자신의 정점으로 돌아간다, 이때 왕복한 시간이 가장 큰 정점의 시간 구하기

파티 BOJ 1238

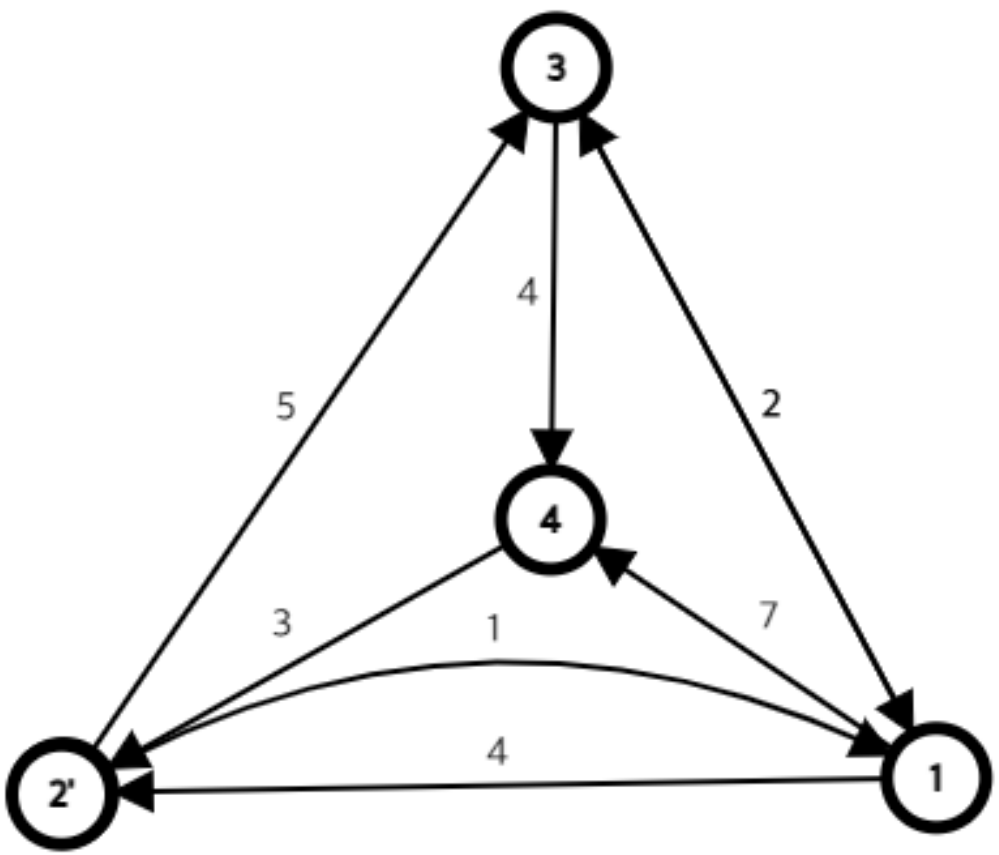
- 단방향 그래프가 주어질 때, 특정 정점에서 파티가 열린다
- 해당 정점으로 다른 **모든 정점이 최단 경로를 통해 도착**한다
- 이후 다시 **자신의 정점으로 돌아간다**, 이때 왕복한 시간이 가장 큰 정점의 시간 구하기

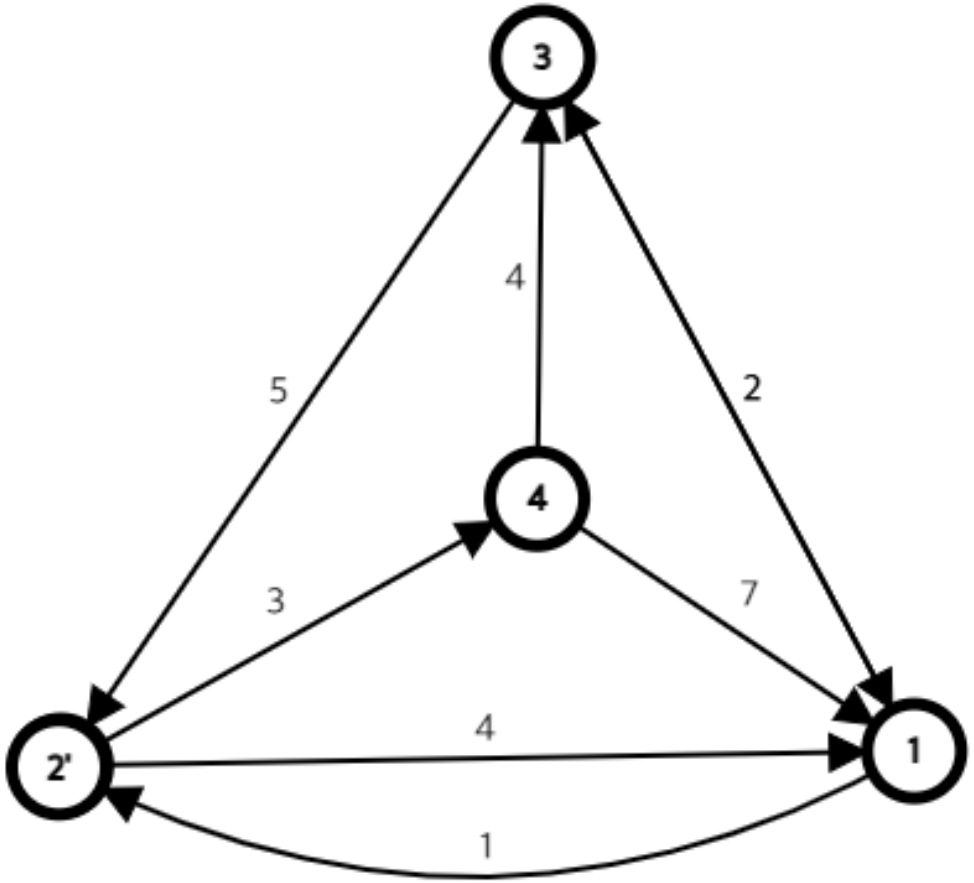
- 자신의 정점으로 돌아가는 것은 **파티 정점으로부터 Dijkstra**를 통해 구할 수 있다
- 모든 정점으로부터 파티 정점으로 향하는 것을 어떻게 빠르게 구할 수 있을까?

파티 BOJ 1238

- 돌아가는 것은 파티 정점 (Source)로부터 모든 정점으로 SSSP를 실행한다
- 반대로, 한 정점으로 모이는 것을 하나의 Source에서 출발하는 것으로 바꿀 수 있을까?
- 모든 정점을 뒤집어 보고, 어떻게 하면 좋을 지 생각해 보자

파티 BOJ 1238



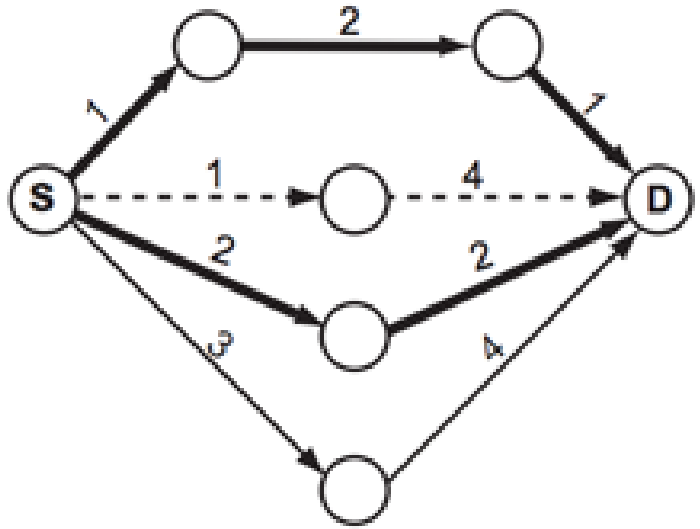


파티 BOJ 1238

- 전체 그래프를 뒤집은 하나의 그래프를 만들자
- 원래 그래프에서 파티 정점으로부터 Dijkstra를 돌린 결과는 돌아가는 길의 최단경로
- 뒤집은 그래프에서 파티 정점으로부터 Dijkstra를 돌린 결과는 모이는 길의 최단경로
- 각 노드에서의 두 가중치를 합한 것 중 최댓값을 찾자

거의 최단 경로 BOJ 5719

- 방향 그래프가 주어진다.
- 최단 경로를 이루는 간선을 사용하지 않고 최단 경로의 길이를 구해 보자



거의 최단 경로 BOJ 5719

- 어떤 간선이 **최단 경로를 이루는 간선인지** 어떻게 판단할 수 있을까?
- 어떤 정점이 최단 경로를 이루는 정점인지는 어떻게 판단할까?

- 앞에서 진행했던 역추적을 사용하면 해결할 수 있지만, 구현량이 많아 어려울 수 있음
- 조금 더 쉽게 문제를 해결해 보자!

거의 최단 경로 BOJ 5719

- 한 정점을 기준으로 해당 정점이 최단 경로에 속하는지 확인하는 방법으로는 역추적하는 방법이 있다
- 다른 방법으로, 파티 문제에서 진행했던 것처럼 역방향 간선을 활용할 수도 있다
- 주어진 그래프의 시작 정점에서, 역방향 그래프의 도착 정점에서 각각 Dijkstra
- 구축되는 두 개의 Distance 배열은 무엇을 의미할까?
- 두 개의 배열에서 최단 경로에 속하는 정점을 어떻게 찾아낼까?

거의 최단 경로 BOJ 5719

- 두 개의 Distance 배열을 두고, 임의의 정점을 잡자
- 시작 정점에서 해당 정점까지의 최단거리 + 도착 정점에서 해당 정점까지의 최단거리
- 위 거리의 합이 시작 정점에서 도착 정점까지의 최단거리와 같다면 의미하는 것은?

거의 최단 경로 BOJ 5719

- 현재 정점을 u, 다음 정점을 v라고 하자.
- $dist_i$를 시작 정점에서 i번 정점까지의 최단거리라고 하면
- $dist_u + w_{u,v} + rdist_v = dist_v$라면, 해당 간선은 최단경로를 이루는 간선이다
- 해당 간선을 제외하고 Dijkstra를 실행해 보자!

All-Pair Shortest Path

- 그래프상의 모든 정점에 대해서, 서로 다른 두 정점 간의 거리를 구하기
- 앞에서 배운 Dijkstra를 모든 정점에 대해서 탐색: $O(ElogV \times V)$
- 간선은 $O(V^2)$개 존재, 시간복잡도는 $O(V^3 logV)$...

결과
시간 초과
시간 초과
시간 초과
시간 초과

Floyd-Warshall: DP Approach

- 정점 k개에 대한 정보를 알고 있다면, 그 정보로 $k+1$번째 정점에 대해서도 알 수 있을까?
- 정점에 번호를 $1, 2, \ldots, V$로 번호를 매긴 뒤, 다음과 같은 DP를 생각해 보자
- $d[k][i][j]$: 1번 정점부터 k번 정점까지만 중간 경로로 사용해서 i에서 j로 가는 최단거리
- $d[k-1][i][j]$로부터 $d[k][i][j]$를 도출해내 보자

Floyd-Warshall: DP Approach

- Base Case:
  - $d[0][i][j]$: i번 정점에서 j번 정점으로 가는 경로가 있다면 가중치, 그렇지 않다면 ∞
- Step: $i \rightarrow j$로 가는 가능한 모든 경로의 집합을 두 가지로 나눌 수 있다
  - 1. k번째 노드를 거쳐가는 경우
  - 2. k번째 노드를 거쳐가지 않는 경우
- 두 가지 경우밖에 없으므로, 이 중 작은 값으로 갱신하면 된다

Floyd-Warshall: DP Approach

- 1. k번째 노드를 거쳐가는 경우
  - k는 두 번 이상 나타나지 않는다. 만약 두 번 나타난다면, 한 번만 나타나도록 경로를 수정할 수 있고, 해당 경로가 이루는 거리가 더 작다.
  - 따라서 $i \to k + k \to j$ 와 같다
- 2. k번째 노드를 거쳐가지 않는 경우
  - $d[k][i][j]$는 1번 정점부터 k번 정점까지를 중간 경로로 사용해서 $i \to j$로 가는 최단거리
  - k번째 노드를 사용하지 않았으므로, $d[k-1][i][j]$와 같다

Floyd-Warshall: Implementation

- 시간복잡도가 $O(V^3)$이기 때문에, 문제에서 대부분 $V \leq 500$정도로 주어진다
- 인접 리스트로 구현하는 것보다 인접 행렬로 구현하는 것이 일반적이고 편리하다

```
for(int i = 1; i <= V; i++)
    for(int j = 1; j <= V; j++)
        d[0][i][j] = g[i][j];
for(int k = 1; k <= V; k++)
    for(int i = 1; i <= V; i++)
        for(int j = 1; j <= V; j++)
            d[k][i][j] = min(d[k-1][i][j], d[k-1][i][k] + d[k-1][k][j]);
```

Floyd-Warshall: 최적화

- 최단거리 배열을 만들어나갈 때, k번째 정점을 생각한다면 $k-1$번째만 확인하면 된다
- 모든 k에 대한 거리를 저장하는 것보다, 바로 직전 것$(k-1)$만을 저장하면 더 효율적
- 공간복잡도를 V^3에서 $2 \times V^2$로 줄일 수 있다!

```
for(int i = 1; i <= V; i++)
    for(int j = 1; j <= V; j++)
        D[i][j] = g[i][j];
for(int k = 1; k <= V; k++)
    for(int i = 1; i <= V; i++)
        for(int j = 1; j <= V; j++)
            D[k%2][i][j] = min(d[(k-1)%2][i][j],
                               d[(k-1)%2][i][k] + d[(k-1)%2][k][j])
```

Floyd-Warshall: 더 최적화...!

- 갱신되기 전 값을 가져간 뒤, 갱신한 값을 배열에 넣어야 한다
- 업데이트 이후 값을 가져가서 계산하면 정답이 나오지 않을 것
- 갱신 여부를 알지 못하기 때문에 두 개의 층을 번갈아가며 사용
- 하지만...

Floyd-Warshall: 더 최적화...!

- $d[k-1][i][k]$와 $d[k][i][k]$, $d[k-1][k][j]$와 $d[k][k][j]$가 서로 다를 수 있을까?
- $d[k][i][j]$: 1번 정점부터 k번 정점까지만 중간 경로로 사용해서 i에서 j로 가는 최단거리
- $i \to k$ 또는 $k \to j$로 갈 때, 중간 노드로 k를 사용할 수 없음
- 따라서 2차원으로 나타냈을 때 $d[i][j]$는 갱신되지 않았고, 공간복잡도를 V^2까지 줄인다
- 흔히 검색하면 보이는 Floyd-Warshall 형태

```
for(int k = 0; k < V; k++)
    for(int i = 0; i < V; i++)
        for(int j = 0; j < V; j++)
            dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
```

플로이드 BOJ 11404

- 기본 플로이드 알고리즘 구현 문제
- 같은 정점 쌍에 대해서 여러 개의 간선이 들어오므로 가장 작은 가중치만을 저장

```cpp
int main(){
    cin >> N >> M;

    for(int i = 0; i < 101; i++)
        for(int j = 0; j < 101; j++)
            if (i != j) d[i][j] = INF; // i -> i의 거리는 0
    while(M--){
        cin >> a >> b >> c;
        d[a][b] = min(d[a][b], c);
    }
    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]);

    for(int i = 1; i <= N; i++){
        for(int j = 1; j <= N; j++){
            if (d[i][j] == INF) cout << "0 ";
            else cout << d[i][j] << " ";
        }
        cout << '\n';
    }

    return 0;
}

```

허들 넘기 BOJ 23286

- 출발 정점에서 도착 정점까지의 경로 중, 허들의 높이의 최댓값을 최소화
- 플로이드를 잊고, 앞에서 설명한 생각의 흐름을 따라서 해당 문제를 모델링해 보자
- Minimum Bottleneck Path
- 반드시 d[i][j]가 덧셈만을 진행하는 게 아니라, 최소/최대에 대해서도 진행할 수 있음을 알아야 문제를 해결할 수 있다

References

https://tildesites.bowdoin.edu/~ltoma/teaching/cs231/fall14/Lectures/14-SSSP/shpaths.pdf

잘못 구현한 다익스트라: https://github.com/infossm/infossm.github.io/blob/master/_posts/2019-01-09-wrong-dijkstra.md