#중급반 3주차

최단 경로

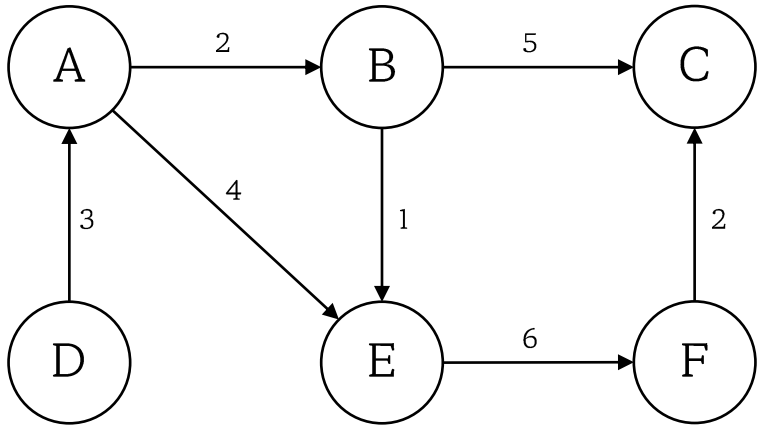
<Dijkstra’s Algorithm>

#다익스트라 알고리즘

<Dijkstra's Algorithm>

저번 시간에 BFS를 이용해 가중치가 없는 그래프에서 최단 거리를 구하는 방법을 배웠습니다.

그렇다면 가중치가 있는 그래프에서는 최단 거리를 어떻게 찾을 수 있을까요?



#다익스트라 알고리즘

<Dijkstra's Algorithm>

정해진 하나의 시작 정점에서부터

다른 모든 정점으로 가는 최단 경로를 구하는 알고리즘

1. 음수 가중치 간선이 존재할 경우 사용할 수 없어요.
2. 시간 복잡도는 O(E log V) / (E = Edge, V = Vertex)

#다익스트라 알고리즘

<Dijkstra's Algorithm>

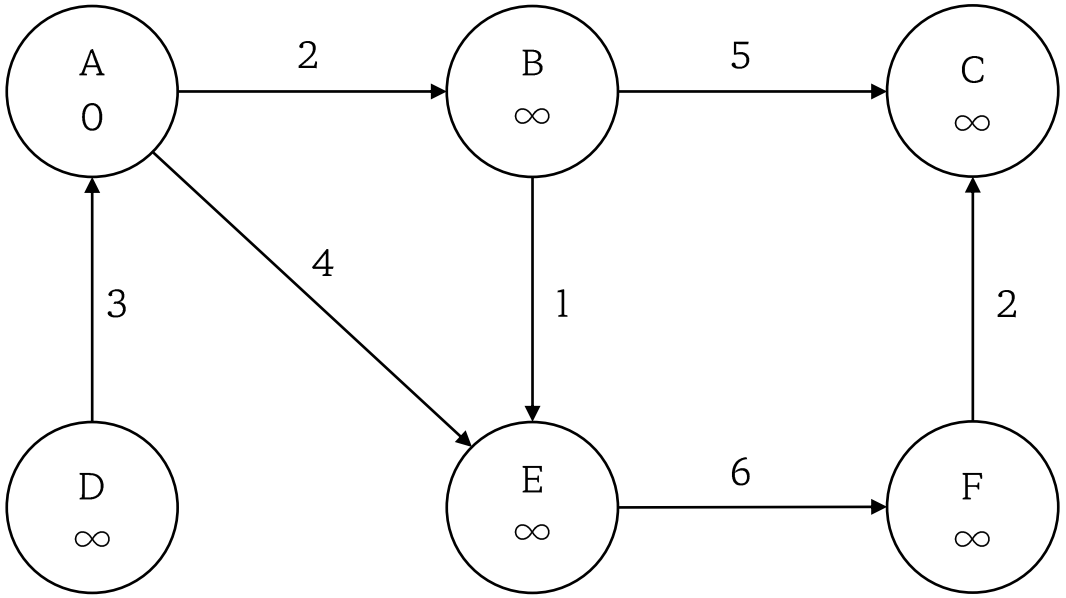
1. 시작 정점의 거리를 0으로, 나머지 정점들의 거리를 inf로 초기화한다.
2. 모든 정점들을 미방문 상태로 놓고, 모든 정점을 방문할 때 까지 아래 과정을 반복한다:
    1. 미방문 정점들 중 거리가 가장 작은 정점을 '현재 정점'으로 한다.
    2. 현재 정점에 방문 표시를 한다.
    3. 연결된 미방문 정점들의 거리를 간선의 가중치를 이용해 갱신한다.

inf는 적당히 큰 값(문제 조건 상 거리의 최댓값보다 큰 값) 이라고 생각하면 됩니다.

#다익스트라 알고리즘

<Dijkstra's Algorithm> - 동작 원리

1. 시작 정점의 거리를 0으로, 나머지 정점들의 거리를 inf로 초기화한다.



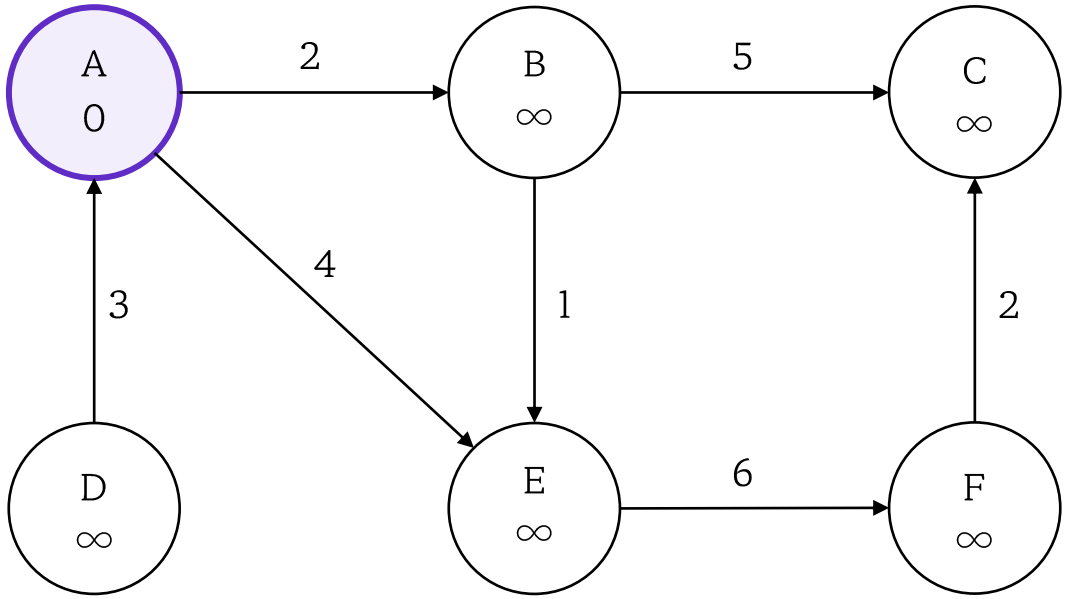
미방문 정점 집합:

{A, B, C, D, E, F}

#다익스트라 알고리즘

<Dijkstra's Algorithm> - 동작 원리

1. 미방문 정점들 중 거리가 가장 작은 정점을 '현재 정점'으로 한다.
2. 현재 정점을 미방문 정점 집합에서 제거한다.



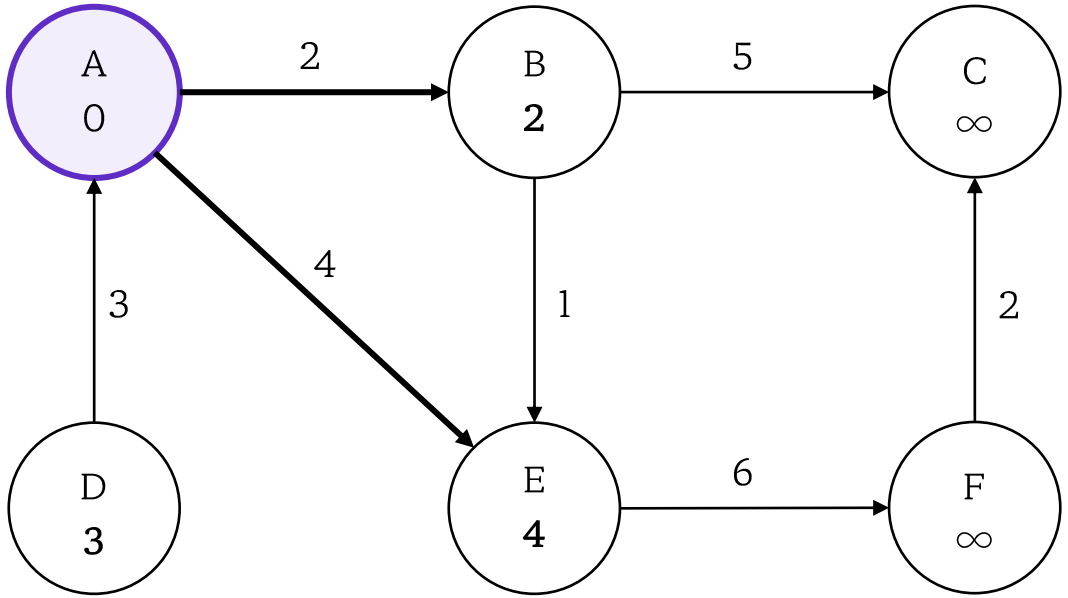
미방문 정점 집합:

{~~A~~, B, C, D, E, F}

#다익스트라 알고리즘

<Dijkstra's Algorithm> - 동작 원리

1. 미방문 정점들 중 거리가 가장 작은 정점을 '현재 정점'으로 한다.
2. 현재 정점을 미방문 정점 집합에서 제거한다.
3. 연결된 미방문 정점들의 거리를 간선의 가중치를 이용해 갱신한다.



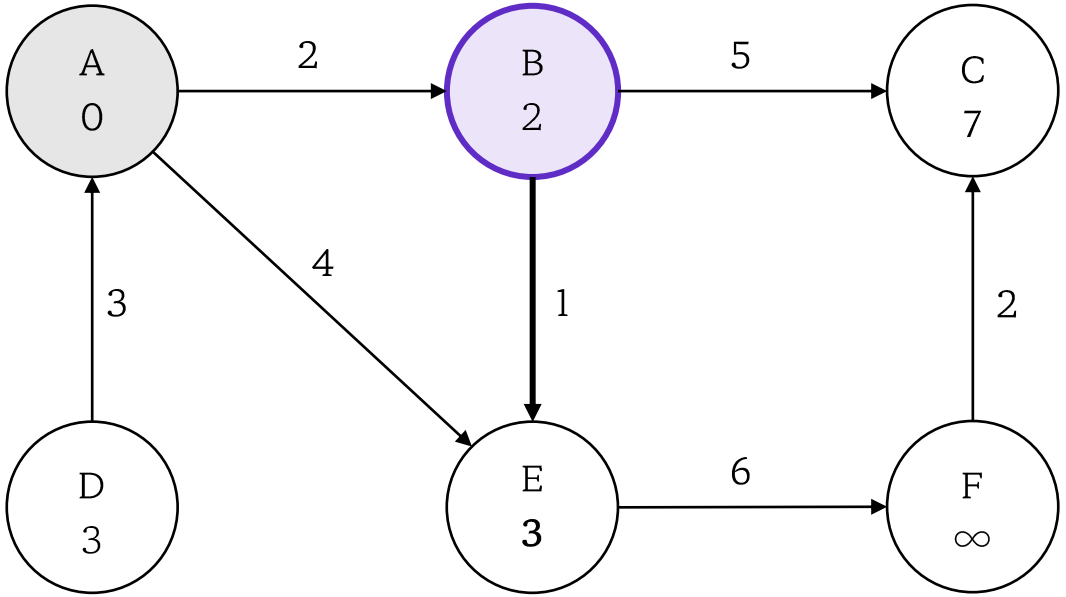
미방문 정점 집합:

{~~A~~, B, C, D, E, F}

#다익스트라 알고리즘

<Dijkstra's Algorithm> - 동작 원리

1. 미방문 정점들 중 거리가 가장 작은 정점을 '현재 정점'으로 한다.
2. 현재 정점을 미방문 정점 집합에서 제거한다.
3. 연결된 미방문 정점들의 거리를 간선의 가중치를 이용해 갱신한다.



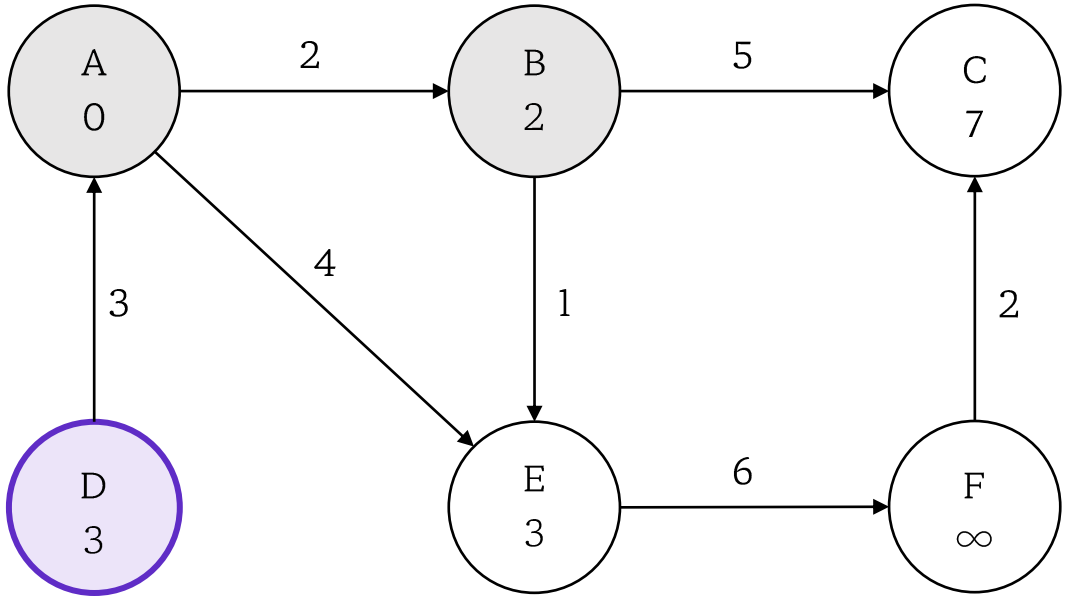
미방문 정점 집합:

{~~B~~, C, D, E, F}

#다익스트라 알고리즘

<Dijkstra's Algorithm> - 동작 원리

1. 미방문 정점들 중 거리가 가장 작은 정점을 '현재 정점'으로 한다.
2. 현재 정점을 미방문 정점 집합에서 제거한다.
3. 연결된 미방문 정점들의 거리를 간선의 가중치를 이용해 갱신한다.



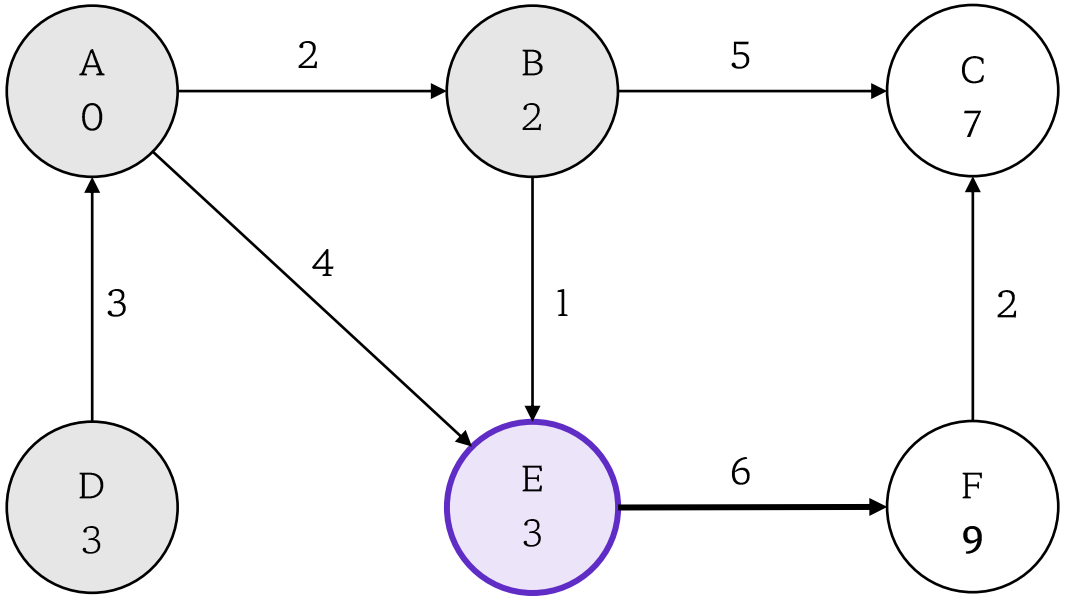
미방문 정점 집합:

{C, ~~D~~, E, F}

#다익스트라 알고리즘

<Dijkstra's Algorithm> - 동작 원리

1. 미방문 정점들 중 거리가 가장 작은 정점을 '현재 정점'으로 한다.
2. 현재 정점을 미방문 정점 집합에서 제거한다.
3. 연결된 미방문 정점들의 거리를 간선의 가중치를 이용해 갱신한다.



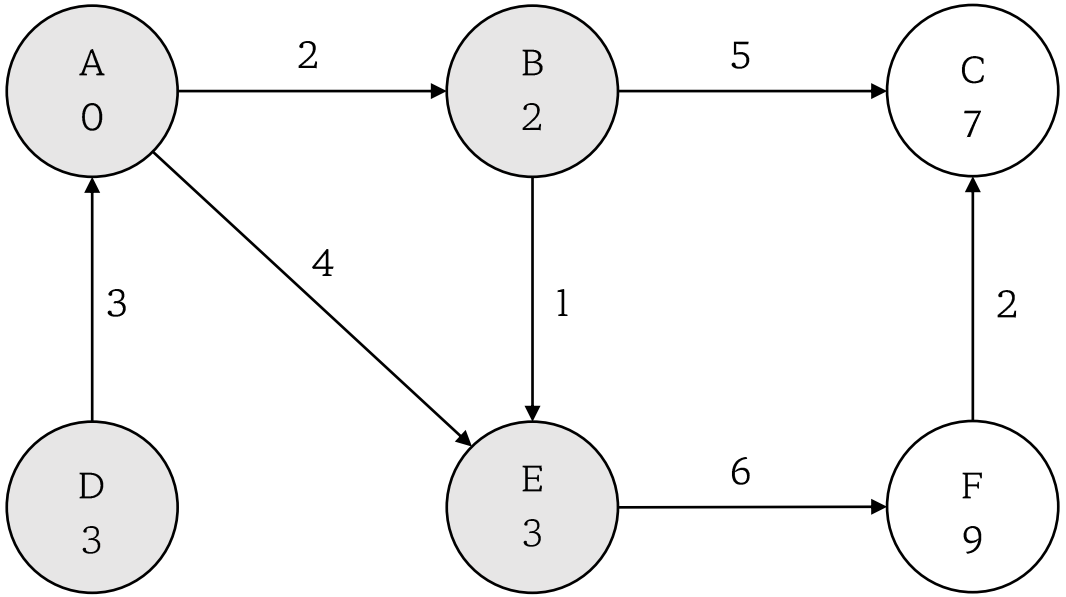
미방문 정점 집합:

{C, ~~E~~, F}

#다익스트라 알고리즘

<Dijkstra's Algorithm> - 동작 원리

1. 미방문 정점들 중 거리가 가장 작은 정점을 '현재 정점'으로 한다.
2. 현재 정점을 미방문 정점 집합에서 제거한다.
3. 연결된 미방문 정점들의 거리를 간선의 가중치를 이용해 갱신한다.



미방문 정점 집합:

{}

#다익스트라 알고리즘

<Dijkstra's Algorithm> - 구현 코드

1. 시작 정점의 거리를 0으로, 나머지 정점들의 거리를 inf로 초기화한다.
2. 모든 정점들을 미방문 상태로 놓고, 모든 정점을 방문할 때 까지 아래 과정을 반복한다.
    1. 미방문 정점들 중 거리가 가장 작은 정점을 '현재 정점'으로 한다.
    2. 현재 정점에 방문 표시를 한다.
    3. 연결된 미방문 정점들의 거리를 간선의 가중치를 이용해 갱신한다.

```
int adj[1010][1010];
int dist[1010];
bool vis[1010];

void dijkstra(int source) {
    fill(dist, dist + 1010, inf);
    dist[source] = 0;
    while (1) {
        int curr = -1;
        for (int i = 1; i <= V; i++) {
            if (vis[i]) {
                continue;
            }
            if (curr == -1 || dist[i] < dist[curr]) {
                curr = i;
            }
        }
        if (curr == -1) {
            break;
        }
        vis[curr] = 1;
        for (int i = 1; i <= V; i++) {
            if (!vis[i] && dist[i] > dist[curr] + adj[curr][i]) {
                dist[i] = dist[curr] + adj[curr][i];
            }
        }
    }
}
```

#다익스트라 알고리즘

<Dijkstra's Algorithm> - 구현 코드

while 문은 총 V번 돌아가고,

while 루프 안에서 $O(V)$번의 연산을 수행하니

총 시간복잡도는 $O(V^2)$가 됩니다.

몇가지 최적화를 해 봅시다.

```
int adj[1010][1010];
int dist[1010];
bool vis[1010];

void dijkstra(int source) {
    fill(dist, dist + 1010, inf);
    dist[source] = 0;
    while (1) {
        int curr = -1;
        for (int i = 1; i <= V; i++) {
            if (vis[i]) {
                continue;
            }
            if (curr == -1 || dist[i] < dist[curr]) {
                curr = i;
            }
        }
        if (curr == -1) {
            break;
        }
        vis[curr] = 1;
        for (int i = 1; i <= V; i++) {
            if (!vis[i] && dist[i] > dist[curr] + adj[curr][i]) {
                dist[i] = dist[curr] + adj[curr][i];
            }
        }
    }
}
```

#다익스트라 알고리즘

<Dijkstra's Algorithm> - 구현 코드

먼저, 그래프를 인접 행렬이 아닌 인접 리스트 형태로 저장합시다.

가중치 w의 u → v 간선이 주어지면:

`adj[u].push_back({v,w});`

이렇게 하면 3. 연결된 정점의 거리를 갱신하는 과정에서, V개 전부 다 확인하는 것이 아닌 연결된 간선들만 확인하게 되므로 시간을 줄일 수 있습니다. 또한 그래프를 저장하는 데에 쓰이는 메모리도 절약됩니다.

정확히 이야기하자면, 3번 과정의 시간복잡도와 그래프를 표현하는 데에 쓰이는 메모리가 $O(V^2)$에서 $O(E)$로 바뀝니다.

```
vector<pair<int, int>> adj[1010];
int dist[1010];
bool vis[1010];

void dijkstra(int source) {
    fill(dist, dist + 1010, inf);
    dist[source] = 0;
    while (1) {
        ...
        for (pair<int, int> i : adj[curr]) {
            int nxt = i.first, cost = i.second;
            if (!vis[nxt] && dist[nxt] > dist[curr] + cost) {
                dist[nxt] = dist[curr] + cost;
            }
        }
    }
}

int main() {
    ...
    for (int i = 0; i < E; i++) {
        int u, v, w;
        cin >> u >> v >> w;
        adj[u].push_back({v, w});
    }
    ...
```

#다익스트라 알고리즘

<Dijkstra's Algorithm> - 구현 코드

1. 미방문 정점 중 거리가 가장 작은 정점을 찾는 과정에서 V개의 정점을 다 확인하기 때문에 시간복잡도는 여전히 $O(V^2)$입니다.

STL에는 삽입과 대표값 조회/삭제를 O(logN)에 지원하는 priority_queue 라는 자료구조가 있습니다.

```
// #include <queue>
priority_queue<int> pq; // 기본적으로 최댓값이 대표값
pq.push(1);
pq.push(2);
pq.push(3);
pq.top(); // = 3
pq.pop(); // top에 있는 원소 삭제
```

이를 활용해 봅시다.

#다익스트라 알고리즘

<Dijkstra's Algorithm> - 구현 코드

`pair<int, int>` 자료형의 `priority_queue`를 선언해, {해당 정점까지의 거리, 정점 번호} pair 정보를 담도록 합니다.

미방문 정점들의 정보가 우선순위 큐에 들어가 있다면, 단순히 `top()`을 확인하는 것으로 거리가 가장 작은 정점을 찾을 수 있습니다.

```
void dijkstra(int source) {
    fill(dist, dist + 1010, inf);
    dist[source] = 0;
    priority_queue<pair<int, int>,
                   vector<pair<int, int>>,
                   greater<pair<int, int>>>
        pq;
    pq.push({dist[source], source});
    while (!pq.empty()) {
        int d = pq.top().first, curr = pq.top().second;
        pq.pop();
        if (d > dist[curr]) {
            continue;
        }
        for (pair<int, int> i : adj[curr]) {
            int nxt = i.first, cost = i.second;
            if (dist[nxt] > dist[curr] + cost) {
                dist[nxt] = dist[curr] + cost;
                pq.push({dist[nxt], nxt});
            }
        }
    }
}
```

#다익스트라 알고리즘

<Dijkstra's Algorithm> - 구현 코드

priority_queue의 기본적으로 최대값을 top으로 두는데, 우리는 최소값이 필요합니다. 따라서 parameter를 아래와 같이 주면 최소값이 top에 오도록 할 수 있습니다.

```
priority_queue<pair<int, int>,
vector<pair<int, int>>,
greater<pair<int, int>>> pq;
```

```
void dijkstra(int source) {
    fill(dist, dist + 1010, inf);
    dist[source] = 0;
    priority_queue<pair<int, int>,
                   vector<pair<int, int>>,
                   greater<pair<int, int>>>
        pq;
    pq.push({dist[source], source});
    while (!pq.empty()) {
        int d = pq.top().first, curr = pq.top().second;
        pq.pop();
        if (d > dist[curr]) {
            continue;
        }
        for (pair<int, int> i : adj[curr]) {
            int nxt = i.first, cost = i.second;
            if (dist[nxt] > dist[curr] + cost) {
                dist[nxt] = dist[curr] + cost;
                pq.push({dist[nxt], nxt});
            }
        }
    }
}
```

#다익스트라 알고리즘

<Dijkstra's Algorithm> - 구현 코드

pq에는 거리가 확정되지 않은 정점들이 들어가 있습니다.

pq에 시작점을 삽입하고, 아래 과정을 반복합니다.

1. 거리가 가장 작은 정점을 찾고(pq.top()), 해당 정점을 pq에서 삭제합니다.
2. 이미 거리가 확정된 정점일 수도 있습니다. 이 경우 넘어갑니다.
3. 연결된 정점들의 거리를 갱신하고, pq에 삽입합니다.

```
void dijkstra(int source) {
    fill(dist, dist + 1010, inf);
    dist[source] = 0;
    priority_queue<pair<int, int>,
                   vector<pair<int, int>>,
                   greater<pair<int, int>>>
        pq;
    pq.push({dist[source], source});
    while (!pq.empty()) {
        int d = pq.top().first, curr = pq.top().second;
        pq.pop();
        if (d > dist[curr]) {
            continue;
        }
        for (pair<int, int> i : adj[curr]) {
            int nxt = i.first, cost = i.second;
            if (dist[nxt] > dist[curr] + cost) {
                dist[nxt] = dist[curr] + cost;
                pq.push({dist[nxt], nxt});
            }
        }
    }
}
```

#다익스트라 알고리즘

<Dijkstra's Algorithm> - 구현 코드

1. 거리가 가장 작은 정점을 찾고(pq.top()), 해당 정점을 pq에서 삭제합니다.
2. 이미 거리가 확정된 정점일 수도 있습니다. 이 경우 넘어갑니다.
3. 연결된 정점들의 거리를 갱신하고, pq에 삽입합니다.

각 간선에 대해 최대 한번 pq에 원소를 삽입하므로, $O(E)$번 원소가 삽입됩니다. 따라서 3번 과정의 총 합은 $O(ElogV)$입니다.

pq가 빌 때 까지 반복하고, 총 $O(E)$번 원소가 삽입되므로 1번 과정의 총 합은 $O(ElogV)$입니다.

따라서 총 시간복잡도는 $O(ElogV)$입니다.

```
void dijkstra(int source) {
    fill(dist, dist + 1010, inf);
    dist[source] = 0;
    priority_queue<pair<int, int>,
                   vector<pair<int, int>>,
                   greater<pair<int, int>>>
        pq;
    pq.push({dist[source], source});
    while (!pq.empty()) {
        int d = pq.top().first, curr = pq.top().second;
        pq.pop();
        if (d > dist[curr]) {
            continue;
        }
        for (pair<int, int> i : adj[curr]) {
            int nxt = i.first, cost = i.second;
            if (dist[nxt] > dist[curr] + cost) {
                dist[nxt] = dist[curr] + cost;
                pq.push({dist[nxt], nxt});
            }
        }
    }
}

int main() {
    ...
    for (int i = 0; i < E; i++) {
        int u, v, w;
        cin >> u >> v >> w;
        adj[u].push_back({v, w});
    }
    ...
```

#다익스트라 알고리즘

<Dijkstra's Algorithm> - 연습 문제

최단 경로 (BOJ #1753)

<문제 설명>

- 정점 V개와 간선 E개로 이루어진 그래프가 주어진다.
- 각 간선에는 가중치가 존재한다.
- K번 정점에서 각 정점까지의 최단거리를 구하기

<제약 조건>

- $1 \leq V \leq 20,000$
- $1 \leq E \leq 300,000$

#다익스트라 알고리즘

<Dijkstra's Algorithm> - 연습 문제

파티 (BOJ #1238)

<문제 설명>

- 마을 N개와 지나는데 T_i 의 시간이 걸리는 도로 M개가 있다.
- X번 마을에서 파티가 열린다.
- 각 마을에 학생들이 살고 있고, 이들은 최단거리로 X번 마을을 방문했다가 다시 돌아온다.
- 오고 가는 데 가장 많은 시간을 소비하는 학생은 누구인가?

<제약 조건>

- $1 \leq N \leq 1{,}000$
- $1 \leq M \leq 10{,}000$

#다익스트라 알고리즘

<Dijkstra's Algorithm> - 연습 문제

파티 (BOJ #1238)

<문제 해설>

각 정점 i에 대해 (i → X 최단거리) + (X → i 최단거리) 가 최대인 i를 찾는 문제입니다.

1. (X → i 최단거리)는 X를 출발점으로 다익스트라 알고리즘을 사용하면 한번에 모든 i에 대해 구할 수 있습니다.
2. (i → X 최단거리)가 문제입니다.
    1. 각 i에 대해 i를 출발점으로 다익스트라 알고리즘을 사용하는 방법으로 해결할 수 있지만, 총 $O(NMlogN)$의 시간복잡도로 구현에 따라 시간초과를 받을 수도 있습니다.
    2. (i → X 최단거리) 는 출발점이 다 다른 반면 도착점이 다 같습니다.
    3. 따라서 모든 간선들을 뒤집은 뒤, X를 출발점으로 다익스트라 알고리즘을 사용하면 (i → X 최단거리) 를 한번에 전부 구할 수 있습니다.
3. 총 두번의 다익스트라 알고리즘을 실행하므로, $O(MlogN)$의 시간복잡도에 해결됩니다.

#연습 문제 도전

4 특정한 최단 경로 (BOJ #1504)

경유지 추가

1 노트 조각 (BOJ #24888)

줍줍

3 최소비용 구하기 2 (BOJ #11779)

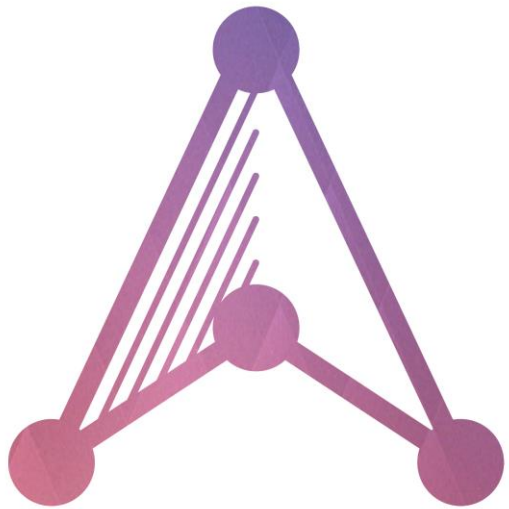
경로까지 알려주세요

5 거의 최단 경로(BOJ #5719)

동등하거나 그 이상

1 도로포장 (BOJ #1162)

마술사 준영이



ALOHA

The algorithm club.