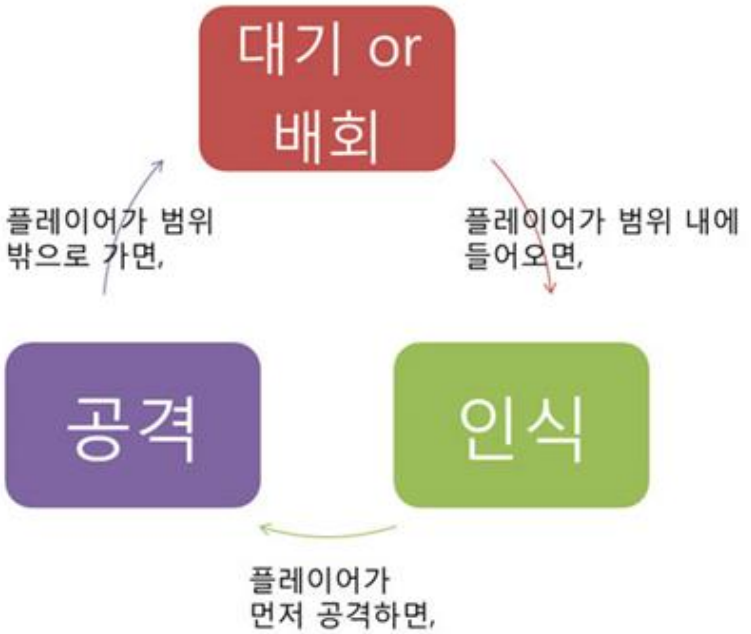
2017 SCSC Workshop

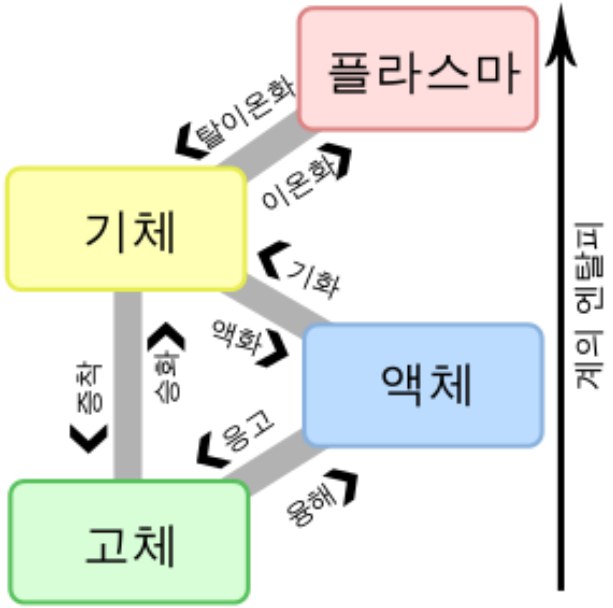
Sogang univ.
System modeling & Optimization Lab
Sangdurk Han

Graph

Search

State


대기 or
배회
플레이어가 범위
밖으로 가면,
플레이어가 범위 내에
들어오면,
공격
인식
플레이어가
먼저 공격하면,


플라스마
기체
액체
고체
탈이온화
이온화
기화
액화
응고
승화
승화
계의 엔탈피

State

- 하나의 경우

Ex) 현재 퍼즐의 모양

비밀번호 (1234)

집에서 학교까지의 거리

Search

- 모든 경우의 수를 탐색 (Backtracking)

Ex) 퍼즐을 최소 횟수로 완성해라.

비밀번호 풀기.

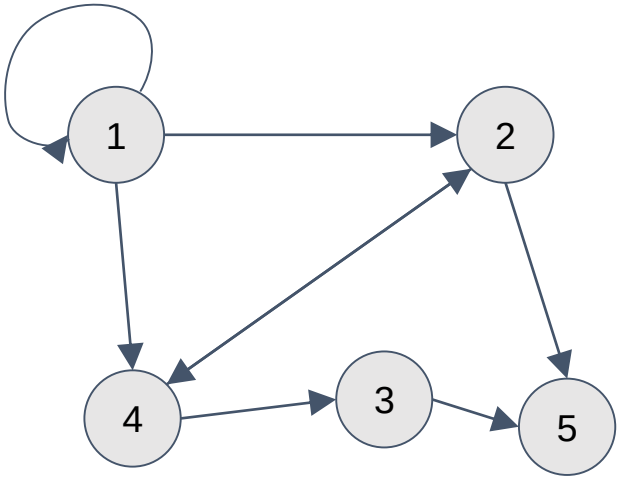
신촌에서 강남역까지 최단 경로는?

Graph

- Graph를 이용하여 표현

경우의 수를 정점(vertex)

그 관계를 간선(edge)

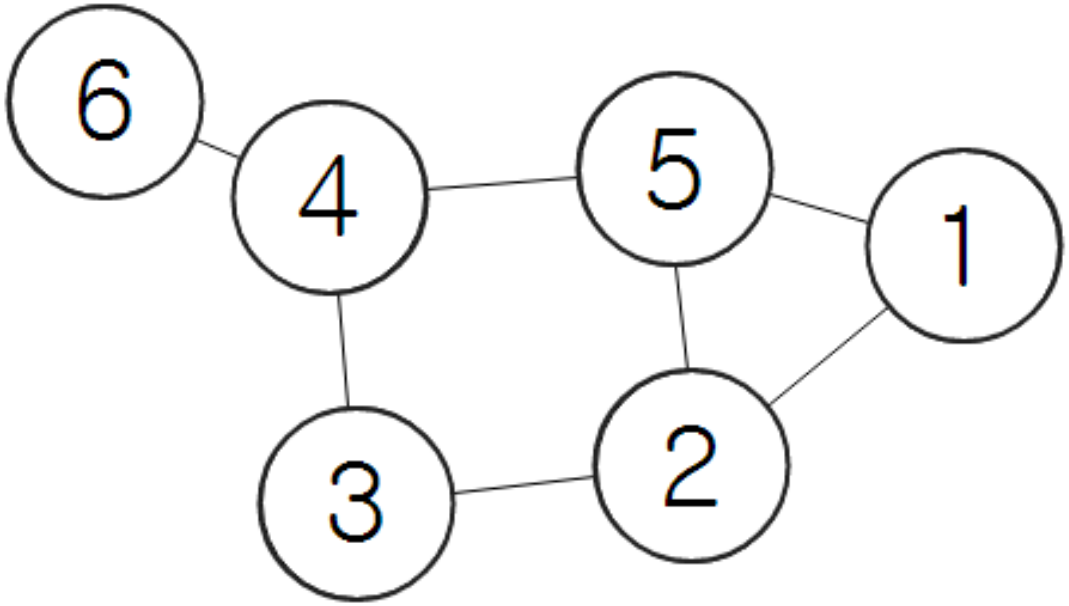


Ex) 정점-역, 간선-선로

정점-캐릭터 상태, 간선-캐릭터 행동

What is a graph?

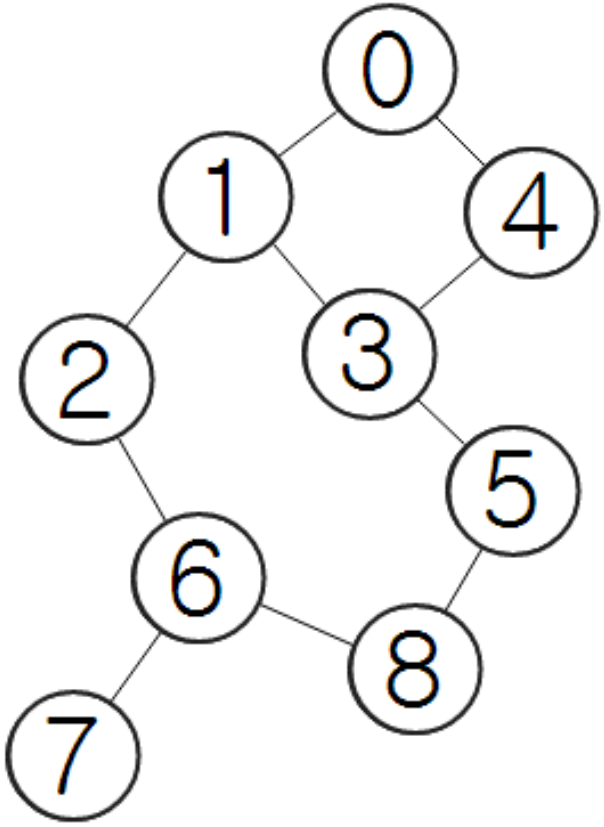
몇 개의 정점과 간선으로 이루어져 있는가?



6개의 정점과 7개의 간선으로 이루어져 있다

Non-directed graph

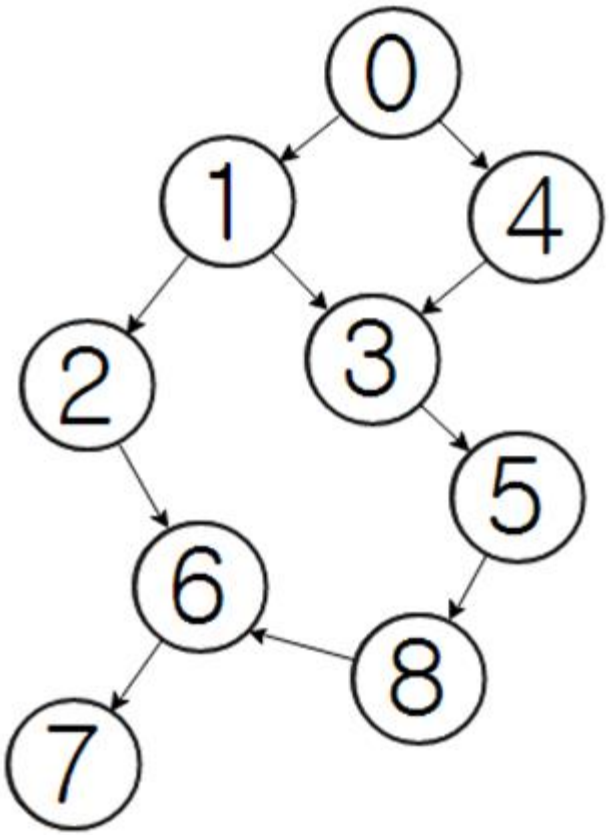
- 무(방)향 그래프(non-directed graph)
  간선의 방향성이 없는 그래프



0번에서 1번으로 갈 수 있고 1번에서 0번으로도 갈 수 있다.

Directed graph

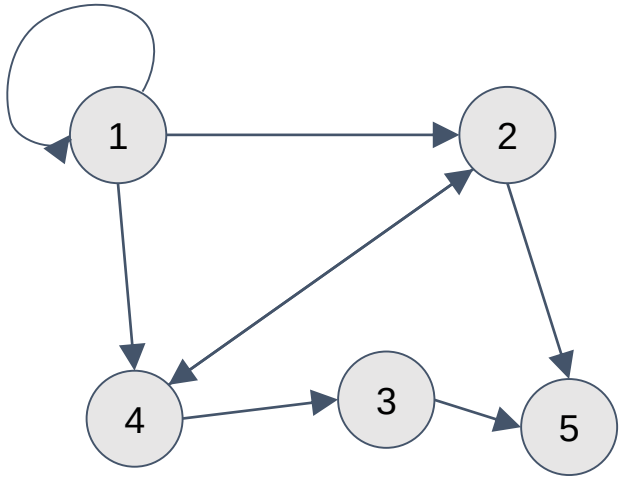
- 방향(유향) 그래프(directed graph)
  간선의 방향성이 있는 그래프



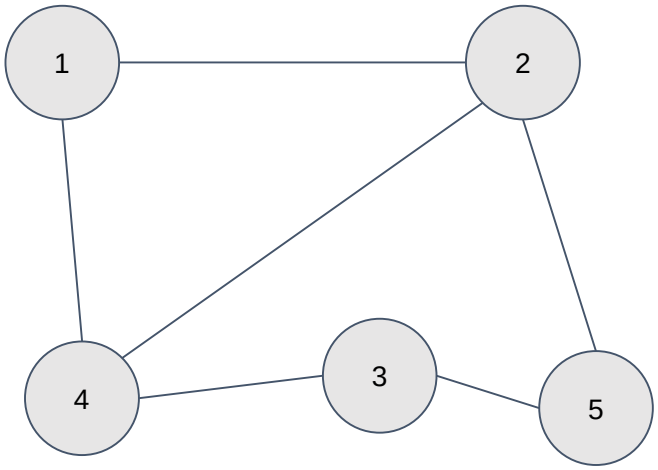
0번에서 1번으로 갈 수 있지만 1번에서 0번 으로 갈 수 없다

Graph 표현

- 1. 인접 행렬 (Adjacency Matrix)
- 2. 인접 리스트 (Adjacency List)

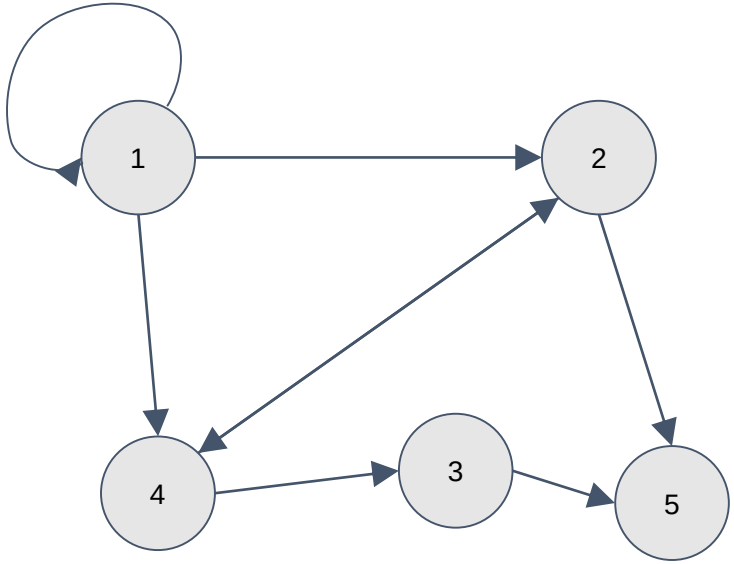


Graph의 표현 - (1) 인접 행렬



	1	2	3	4	5
1	0	1	0	1	0
2	1	0	0	1	1
3	0	0	0	1	1
4	1	1	1	0	0
5	0	1	1	0	0

Graph의 표현 - (1) 인접 행렬

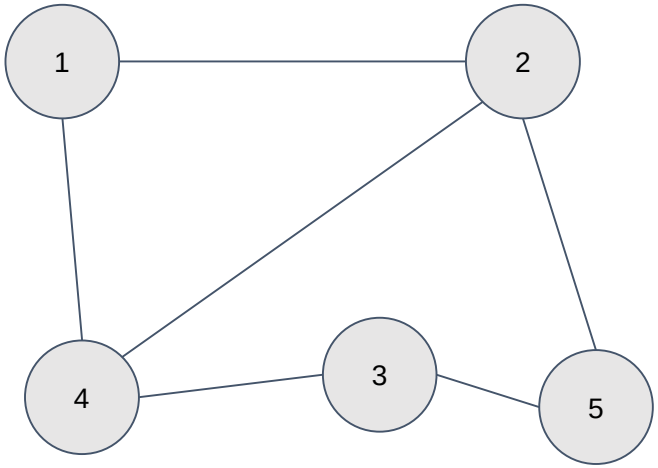


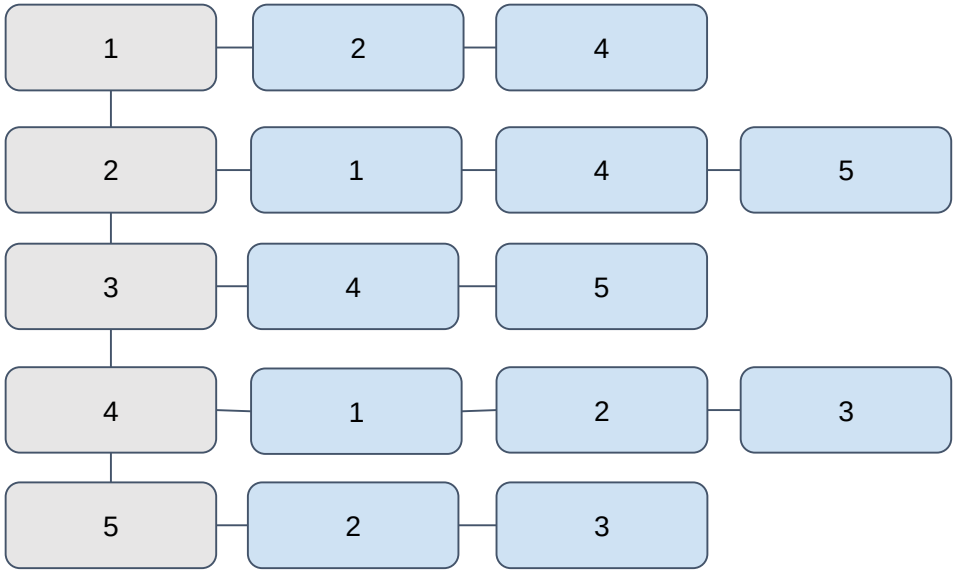
	1	2	3	4	5
1	1	1	0	1	0
2	0	0	0	1	1
3	0	0	0	0	1
4	0	1	1	0	0
5	0	0	0	0	0

인접 행렬

```cpp
#include <stdio.h>
#include <vector>

using namespace std;

const int N = 101; // number of vertex
int e[N][N];

int main(){
    int n, m; // number of vertex, edge
    scanf("%d %d", &n, &m);
    for(int i=0; i<m; i++){
        int a, b;
        scanf("%d %d", &a, &b); // edge a to b
        e[a][b] = 1;
        // e[b][a] = 1; 양방향 일 경우
    }
}
```

Graph의 표현 - (2) 인접 리스트



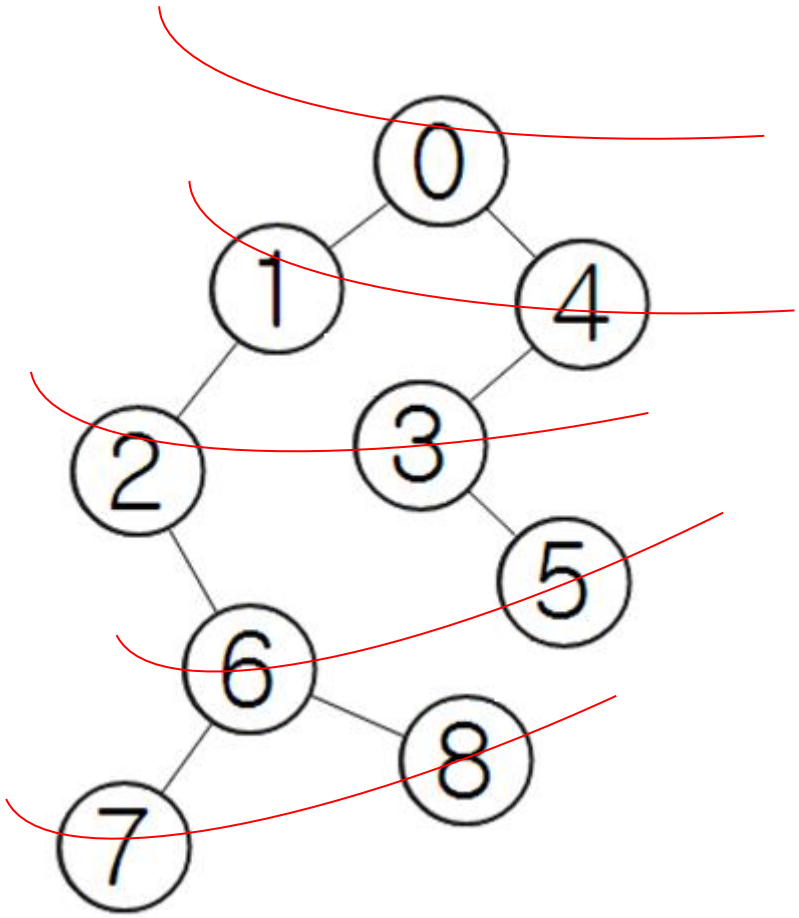


인접 리스트

```cpp
#include <stdio.h>
#include <vector>

using namespace std;

const int N = 101; // number of vertex
vector <int> e[N];

int main(){
    int n, m; // number of vertex, edge
    scanf("%d %d", &n, &m);
    for(int i=0; i<m; i++){
        int a, b;
        scanf("%d %d", &a, &b); // edge a to b
        e[a].push_back(b);
        // e[b].push_back(a); 양방향 일 경우
    }
}
```

BFS animated example

- 가로로 먼저 탐색합니다.



탐색 순서

0 1 4 2 3 6 5 7 8

BFS code

인접 행렬

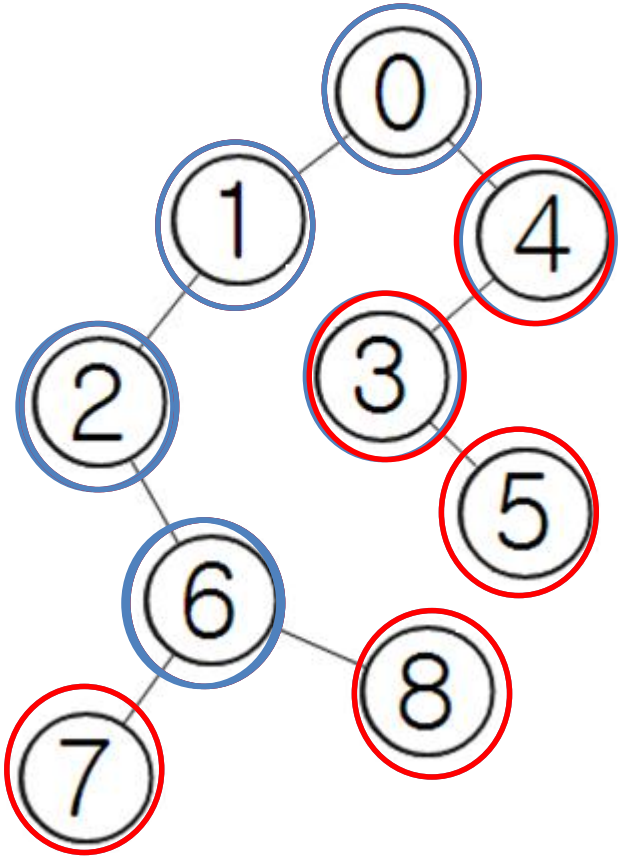
```
int visited[1001]; // 정점의 개수, 이미 방문했다면 1
int s;// 탐색을 시작할 정점
queue<int> q;
q.push(s);
visited[s] = 1;
while (! q.empty()){
    int cur = q.front();
    q.pop();
    for (int i=0; i<N; ++i){
        if (!visited[i] && matrix[cur][i])
        {
            visited[i] = 1;
            q.push(i);
        }
    }
}
```

인접 리스트

```
int visited[1001]; // 정점의 개수, 이미 방문했다면 1
int s;// 탐색을 시작할 정점
queue<int> q;
q.push(s);
visited[s] = 1;
while (! q.empty()){
    int cur = q.front();
    q.pop();
    for (int i=0; i<list[cur].size(); ++i){
        int u = list[cur][i];
        if (!visited[u])
        {
            visited[u] = 1;
            q.push(u);
        }
    }
}
```

DFS animated example

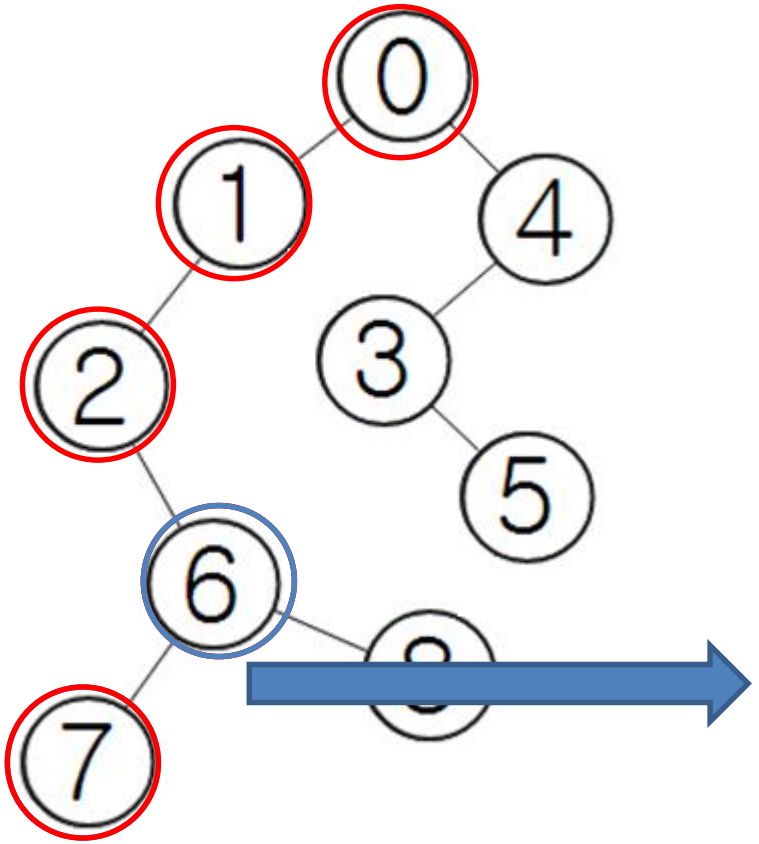
- DFS(Depth First Search, 깊이 우선 탐색)
  깊어질 수 있을 때 까지 계속 탐색



탐색 순서

0 1 2 6 7 8 4 3 5

DFS backtracking



더 이상 탐색 할 수 없을때 자신의 이전 노드 로 돌아가는 과정을 backtracking 이라고 한다

DFS code

- Iterative 구현 방법은 BFS에서 queue대신 stack을 사용하면 된다.
  아래는 recursive 한 방법

인접 행렬

```
int visited[1001];//정점이 1000개라고 가정
int matrix[1001][1001];
int N;// 정점의 개수
void dfs(int cur){
    visited[cur] = 1;
    for (int i=0; i<N; ++i){
        if (!visited[i] && matrix[cur][i])
            dfs(u);
    }
}
```

인접 리스트

```
int visited[1001];//정점이 1000개라고 가정
vector<int> list[1001];

void dfs(int cur){
    visited[cur] = 1;
    for (int i=0; i<list[cur].size(); ++i){
        int u = list[cur][i];
        if (!visited[u])
            dfs(u);
    }
}
```

DFS, BFS efficiency

- DFS와 BFS의 시간 복잡도

인접 행렬 : $O(V^2)$
인접 리스트 : $O(V + E)$

V : 정점의 수 E: 간선의 수

Applications

- DFS와 BFS는 모든 노드를 탐색하게 된다. 이러한 성질을 이용하여 모든 경우를 생각하는 문제를 해결 할 수 있다.

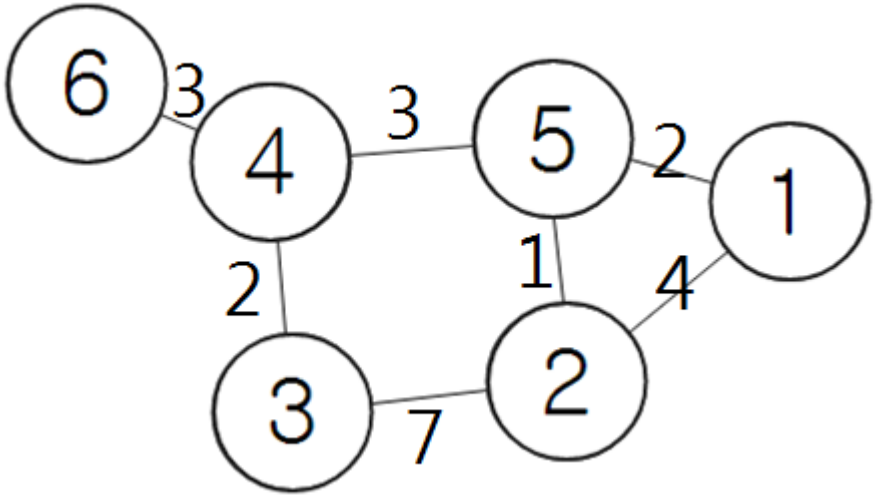
- BFS를 이용하여 가중치가 없는 그래프에서 최단 경로 문제를 풀 수 있다.

쉬는 시간~

Weighted Graph

- Weighted Graph

  간선에 비용(가중치)가 추가 된 그래프



Weighted Graph

- 인접 행렬로 가중치 그래프 표현 하기.

Adj[i][j]= { cost , i와 j사이 간선이 존재
infinity, 간선이 존재하지 않으면

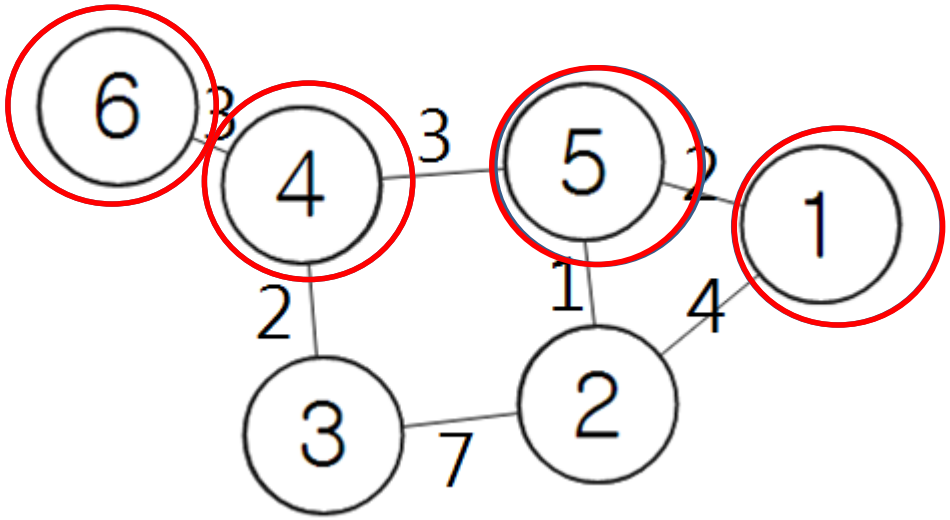
- 인접 리스트로 가중치 그래프 표현

```
vector <ii> e[MAX_NODE + 1];
e[a].push_back( ii(b, cost) );
```

// a에서 b로 cost만큼

Weight Graph

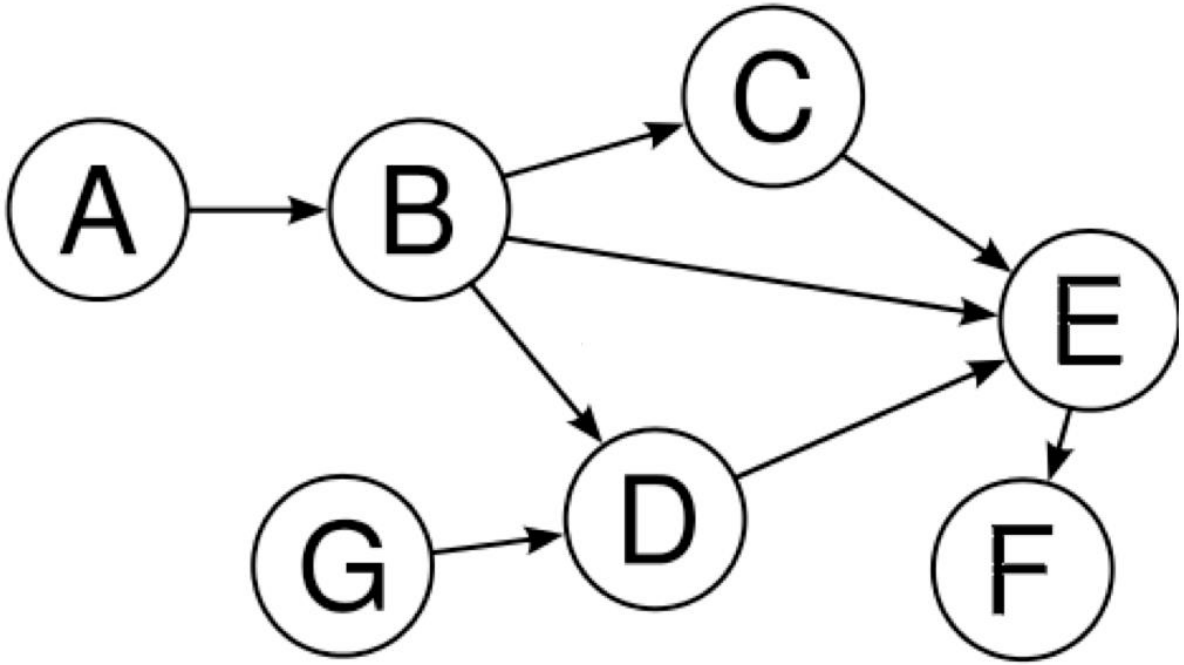
각각의 정점을 도시 라고 한다면 1번 도시에서 6번 도시까지 갈 때 필요한 최소 비용은 얼마일까?



정답 : 8

Weight Graph

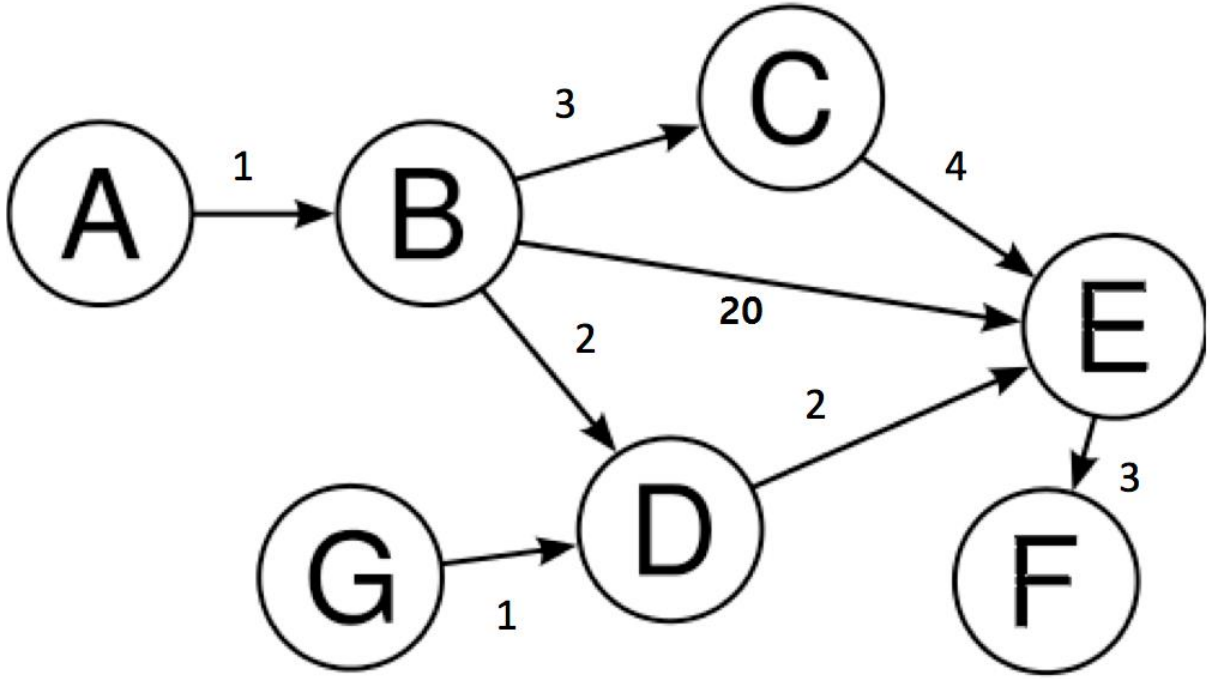
BFS를 사용할 수 있는 경우



A → B → E → F

Weight Graph

BFS를 사용하면 안 되는 경우



A → B → D → E → F

Weight Graph

- 어떻게 해야 효율적으로 최단 경로를 찾을 수 있을까?

- Dijkstra's algorithm (단일 시작점)
- Floyd's algorithm (모든 쌍)
- Bellman-Ford algorithm (단일 시작점)

Dijkstra's algorithm

- Dijkstra's algorithm

http://www.cs.sunysb.edu/~skiena/combinatorica/animations/anim/dijkstra.gif

Dijkstra’s algorithm

- Dijkstra’s algorithm

  하나의 시작점에서 다른 모든 정점까지 가는 최단 거리를 구한다

조건 : 모든 간선은 음이 아닌 가중치를 갖는다.

Idea

- 정점 u, v, k에 대해 u에서 v로 가는 경로를 p_1, u에서 k로 가는 경로를 p_2라 하자.

- 정점 u에서 v로 가는 경로 p_1보다 빠른 경우가 존재하고, 그 경로가 $p_2(u \sim k) \rightarrow v$이면, $p_2 < p_1$이다.

- 결국 경로의 길이가 작은 것부터 탐색

How to work?

1. 시작점을 방문

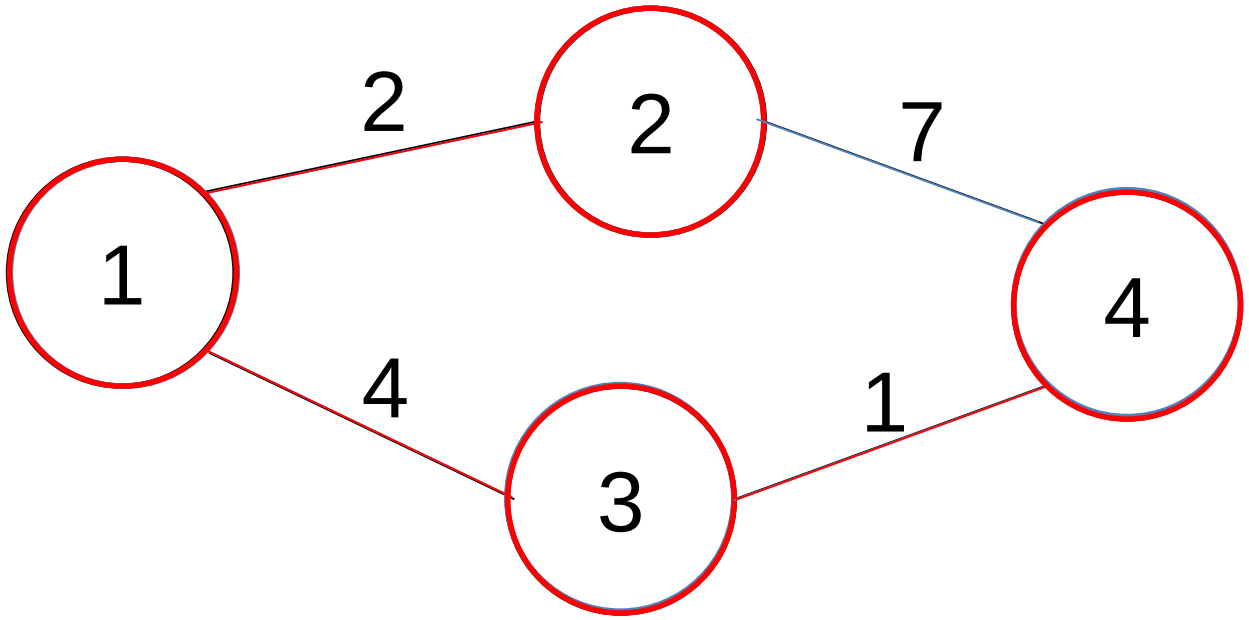
2. 방문한 정점들과 이어진 모든 간선들 중 경로의 길이가 가장 작은 지점(v)을 방문

3. u까지의 거리를 최단 거리로 갱신

4. 2~3번 반복

Dijkstra animated example

- 1번에서 각각의 정점까지 최단 거리를 구해야 한다면??



dist[1] = 0

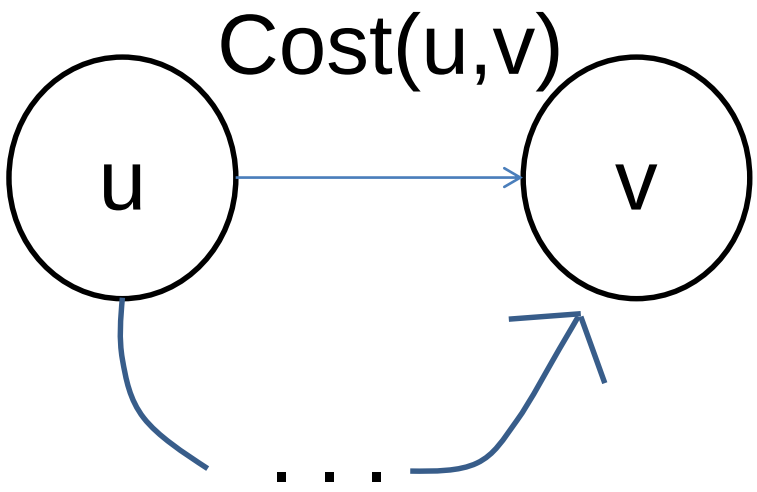
dist[2] = 2

dist[3] = 4

dist[4] = 5

Why it works?

- 이런 작동 방법이 과연 최단거리가 될까요??



귀류법을 통해 증명을 하면 됩니다.

Cost(u,v) < cost_a 인 상황에서 cost_a + cost_b + … + cost_z < cost(u,v)라고 가정을 합니다. Cost(u,v)-cost_a < 0 이므로 Cost(u,v)-cost_a > cost_b+…+cost_z가 됩니다. 맨 처음에 모든 간선의 가중치는 음수가 아니라고 가정을 하였으므로 cost_b + … + cost_z >= 0 이 되고 cost(u-v)-cost_a > 0 이 되므로 가정의 모순이 되므로 cost(u,v) 부터 탐색을 하는 것이 정답임을 보장 받습니다.

Dijkstra’s algorithm

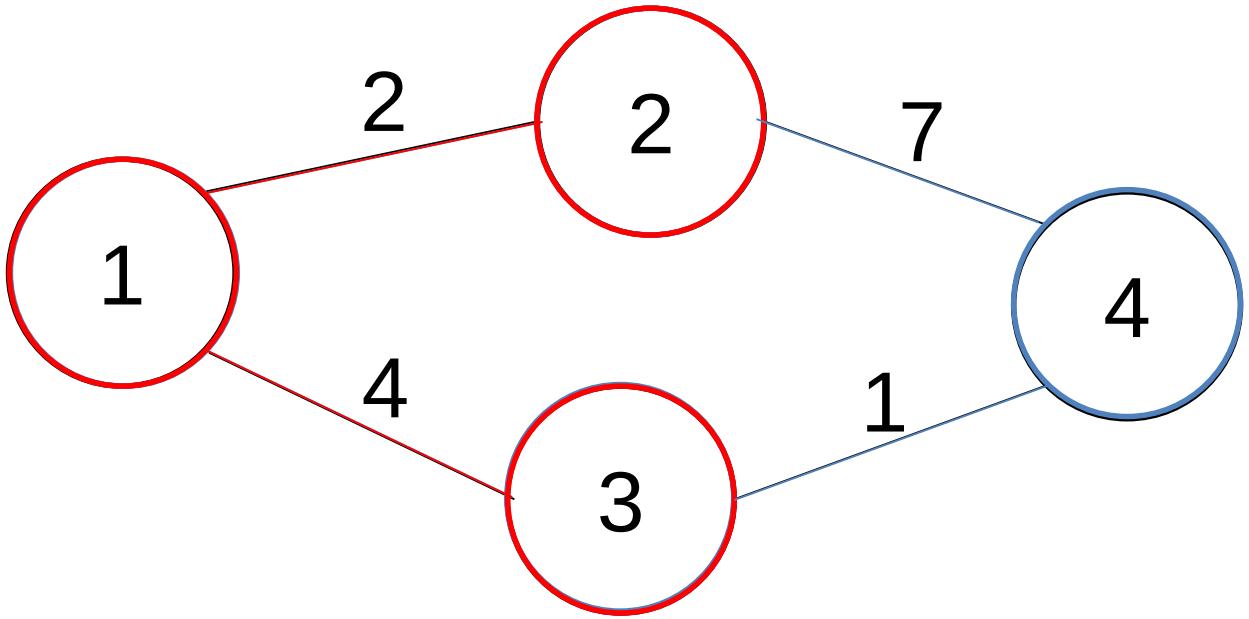
- 인접한 최소 비용의 간선을 고르는 방법
  인접한 간선을 전부 heap에다가 집어넣고
  가장 작은 경로를 Pop.

  나머지 간선은 그대로 heap에 있으므로
  다음 번에 필요한 간선만 Push.

가장 작은 경로 : heap, priority_queue

Dijkstra's algorithm

- 서로 다른 간선이 향하는 정점이 같을 때는 어떻게 해야 할까??



이러한 상황이 오면.... 우선순위 큐에 하나만 넣기도 그렇고... 먼저 집어 넣은게 비용 7을 가진 간선이면 그 값을 바꿔주어야 할까???

그냥 하나 더 우선순위 큐에 집어넣으면 된다. 방문을 하면 그 지점은 더 이상 방문하지 않아도 되는 것이 알고리즘의 동작 과정이기 때문이다.

Dijkstra code

- 그래프는 인접 리스트로 표현하여 구현합니다. (만일의 상황을 대비해서..)

```
#include <cstdio>
#include <queue>
#include <vector>
#define INF 987654321
using namespace std;
typedef pair<int,int> ii;
int main(){
    vector<ii> list[10001]; //pair-first:연결된
    in dist[10001];         //최단 거리를 저장할
    for (int i=0;i<n;++i)
        dist[i] = INF;      // 처음은 무한대로 초
    /* 간선을 잘 연결 합니다 */

    priority_queue<ii,vector<ii>,greater<ii> > p
    // first: 총 가중치, second : 다음 정점 .
    //first먼저 정렬하고 second를 정렬하기 때문에
    // 반드시 지켜주어야 합니다.

    int s=1; // 시작 지점
    pq.push(ii(0,s));
```

```
while (! pq.empty())
{
    ii cur = pq.top();
    int u = cur.second;     // 현재 지점
    int cost = cur.first;   // 현재 가중치
    pq.pop();
    if (dist[u] < cost) continue;   //작은게 있으면 무시
    for (int i=0; i<list[cur].size(); ++i)
    {
        int v = list[cur][i].first;//다음 정점
        int w = list[cur][i].second;// 가중치
        if (dist[v] > cost+w)
        {
            dist[v] = cost+w;
            pq.push(ii(dist[v],v));
        }
    }
}
```

Dijkstra efficiency

- Dijkstra's algorithm 시간 복잡도

  original : $O(|E| + |V|^{2)}$

  min-pq : $O(|E| + |V|\log|V|)$

- 빠르다

- 음수 가중치가 있다면 사용할 수 없다.

Floyd’s algorithm

- Floyd’s algorithm
  - 모든 쌍 최단 거리 알고리즘
  - 다이나믹 프로그래밍으로 동작
  - 음수 가중치가 있어도 사용 가능

Idea

- 1. 두 정점 u, v가 있을 때, $u \rightarrow v$ 로 가는 경로와 임의의 정점 k가 있을 때 $u \rightarrow k + k \rightarrow v$로 가는 경로를 비교하여 갱신

ex) $(1 \rightarrow 2) > (1 \rightarrow 3 \rightarrow 2) > (1 \rightarrow 3 \rightarrow 4 \rightarrow 2)$

- 2. 경로 + 정점 + 경로의 횟수는
  최대 $|V|$번보다 적음

How to work?

정의

$d[k][i][j]$: k번 정점까지 경유점으로 썼을때 i에서 j 로 가는 최단 거리

점화식

$$d[t][i][j] = \min(d[t-1][i][j], d[t-1][i][k] + d[t-1][k][j])$$

i에서 j로 가는 경로와 i에서 k로 가는 경로 + k에서 j로 가는 경로 비교하여 둘 중 최단 경로 선택

이러한 과정을 총 |V|번 진행

Floyd code

- 인접 행렬을 사용합니다.

```
#define MAX_V 1000
int V;
int adj[MAX_V][MAX_V];
int d[MAX_V][MAX_V][MAX_V];
void Floyd(){
    //d[0]을 초기화
    for (int i=0;i<V;++i)
        for (int j=0;j<V;++j)
            if (i != j)
                d[0][i][j] = min(adj[i][j],adj[i][0]+adj[0][j]);
            else
                d[0][i][j] = 0;

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

Reduce memory

- 메모리가 너무 낭비되는데 줄일 수 없을까?

$$d[t][i][j] = \min(d[t-1][i][j], d[t-1][i][k] + d[t-1][k][j])$$

t번째 항과 $t-1$번째 항의 관계로 나타나므로 배열의 크기를 $d[2][MAX_V][MAX_V]$로 줄일 수 있습니다.

Reduce memory

- 더 줄이고 싶어요.

- 점화식 계산에 사용되는 $d[k-1][i][k]$ 와 $d[k-1][k][j]$ 가 $d[k][i][k], d[k][k][j]$와 얼마나 다른지 생각해 봅니다.

- 도착점 이 경유점으로 추가 되는 것은 아무 의미가 없습니다. 따라서 두 식은 같습니다.

Reduce memory

- 더 줄이고 싶어요.

- $d[k-1][i][k] \leq d[k][i][k]$
- $d[k-1][k][j] \leq d[k][k][j]$

- $d[t][i][j] = \min(d[t-1][i][j], d[t-1][i][k] + d[t-1][k][j])$
- $d[t][i][j] = \min(d[t-1][i][j],$
  $d[t \text{ or } t-1][i][k] + d[t \text{ or } t-1][k][j])$

Floyd code

- 인접 행렬 배열에 최단 거리를 계산하여 메모리를 줄입니다.

```
#define MAX_V 1000
int V;
int adj[MAX_V][MAX_V];
void Floyd(){
    //d[0]을 초기화
    for (int i=0; i<V; ++i)
        adj[i][i] = 0;

    // 계산
    for (int k=0; k<V; ++k)
        for (int i=0;i<V;++i)
            for (int j=0;j<V;++j)
                adj[i][j] = min(adj[i][j],adj[i][k]+adj[k][j]);
}
```

Floyd efficiency

- Floyd's algorithm 시간 복잡도

  $O(|V|^3)$

- 반복문 내부가 간단해서 다익스트라, 벨만 포드를 각각 V번 돌리는 것보다 빠르다

- 코드가 간결

Bellman-Ford algorithm

– Bellman-Ford algorithm

최단거리를 확정해 나가는 폴로이드 와셜 알고리즘과 유사하게 최단 거리 의 오차를 반복적으로 줄여가는 방식으로 동작한다.

단일 시작점 최단거리 알고리즘.

– 오차를 줄여나가기 위해

$dist[v] \leq dist[u] + w(u, v)$

식을 항상 만족 시켜야 한다.

Relaxation

- 오차를 줄이는 중간 과정.

```
for (int i=0;i<V;++i)
{
    for (int j=0;j<list[i].size();++j)
    {
        int e = list[i][j].first;  // 다음 정점
        int w = list[i][j].second; // 가중치
        if (upper[e] > upper[i]+w)
            upper[e] = upper[i]+w;
    }
}
```

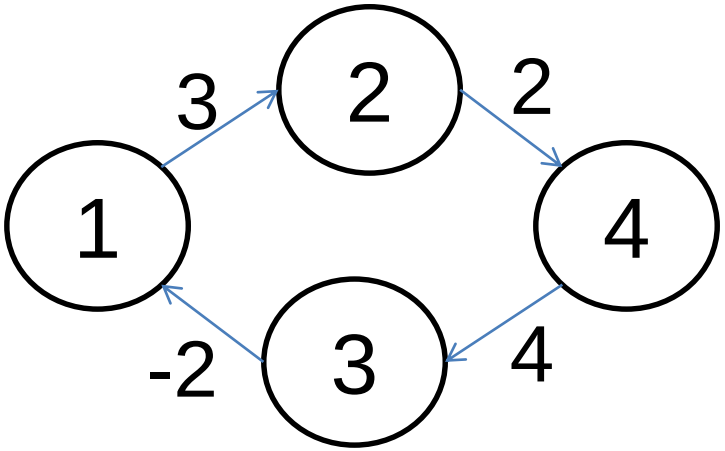
Bellman-Ford algorithm

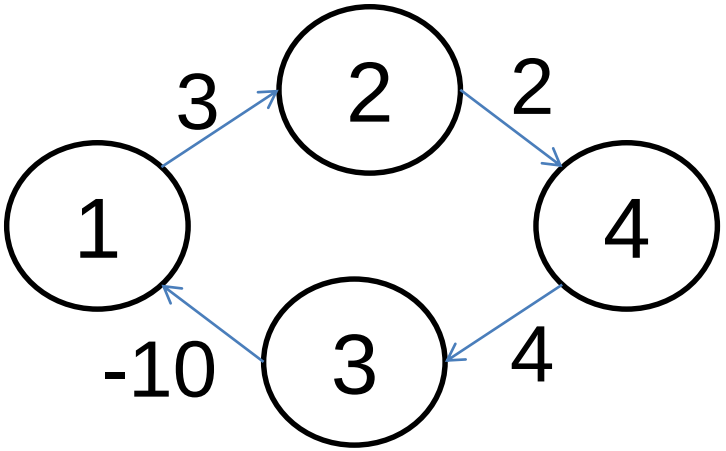
- 오차를 최대 몇 번 줄여야 할까?
  V-1번 줄이면 됩니다.

최단 거리가 되려면 최대 V-1개의 간선으로 연결 되야 합니다. V-1개를 초과하여 최단 거리가 찾아진다면 이는 어떤 정점을 2번 이상 경유해 갔다는 말이죠.(모순)

What is a negative cycle?

- V-1개를 초과하여 방문한 경우가 최단 거리가 되는 경우가 과연 있을까요?





두 그래프의 차이점은 무엇일까요?

How to check a negative cycle?

- V-1개를 초과 방문 하여 최단 거리가 된다면 음수 사이클이 존재한다.

```
bool updated;
for (int iter=0;iter<V;++iter)
{
    updated = false;
    for (int i=0;i<V;++i)
    {
        for (int j=0;j<list[i].size();++j)
        {
            int e = list[i][j].first;  // 다음 정점
            int w = list[i][j].second; // 가중치
            if (upper[e] > upper[i]+w)
            {
                updated = true;
                upper[e] = upper[i]+w;
            }
        }
    }
    if (!updated)
        break;
}
if (updated)
    printf("negative cycle!!");
```

Bellman-Ford code

```
int upper[10001];
for (int i=0;i<10001;++i)
    upper[i] = 987654321;
uper[src] = 0;
bool updated;
for (int iter=0;iter<V;++iter)
{
    updated = false;
    for (int i=0;i<V;++i)
    {
        for (int j=0;j<list[i].size();++j)
        {
            int e = list[i][j].first;  // 다음 정점
            int w = list[i][j].second; // 가중치
            if (upper[e] > upper[i]+w)
            {
                updated = true;
                upper[e] = upper[i]+w;
            }
        }
    }
    if (!updated)
        break;
}
if (updated)
    printf("negative cycle!!");
else
    for (int i=0;i<10001;++i)
        printf("the shortest path from src to %d is %d",i,upper[i]);
```

Bellman-Ford efficiency

- Bellman-Ford algorithm 시간 복잡도

  $O(|V|*|E|)$

- 장점: 음수 가중치가 포함되어도 최단 거리를 찾을 수 있다.

- 단, 음수 Cycle은 제외 $(a \rightarrow b = -inf)$

Q&A