**그래프**

**(1) 그래프 G = (V,E)의 쌍**

- V는 vertex의 set, E는 edge의 set

- 방향성 그래프 vs 비방향성 그래프

**(2) 인접(Adjacency) : (a,b)라는 간선이 있으면 정점 a와 b가 인접하다 성립**

- 무방향성 그래프에서 인접관계는 동일하다 but, 방향성 그래프에서는 (1,2)인접해도 (2,1)인접을 보장하지 않음

**(3) 차수(Degree)**

- 진출차수(out-degree), 진입차수(in-degree) : 정점을 나가거나 들어오는 간선의 수

**(4) 경로(Path)**

- 경로 = 정점 a에서 b까지 가는 **"정점의 순서"**

- 경로의 길이 = 경로에 있는 **"간선의 수"**

- 단순경로(Simple path) : 경로에 있는 모든 정점이 서로 다른 경우 -> 사이클을 만들지 않는다.

**(5) 순환**

- 순환 : 경로 <V0, V1, V2, ..., Vk>에서 V0=Vk면 순환

- 단일순환 : 순환중에서 V1~Vk가 서로 다르면 단일순환 (작은원 없는 큰원)

**(6) 연결**

- 비순환그래프(acyclic graph) : 순환없는 그래프

- 연결그래프(connected graph) : 정점의 모든 쌍이 경로를 갖는 무방향그래프 (끊어진애 없는)

- 강한 연결(Strongly connected) : 방향성 그래프에서 정점의 각 쌍이 서로 도달가능

- 완전그래프(Complete graph) : 무방향성 그래프에서 모든 정점의 쌍이 서로 인접함

- Dag : 비순환 방향성 그래프(Directed acyclic graph)

- 포레스트(Forest) : 비순환 무방향성 그래프

- 트리(Tree) : 연결된 비순환 무방향성 그래프(Connected, Acyclic, Undirected Graph)

**※ 트리가 되기 위한 그래프 조건 : Connected, Undirected, Acyclic(순환없음) : 쭉 잡아올려서 트리되면 ㅇㅋ**

**=> 두 정점은 항상 Unique Simple Path로 연결됨! (트리의 가장 큰 특징)**

**=> 트리가 연결되어 있고 비순환이면 E = V-1를 만족!**

**=> 트리에서 어떤 간선을 제거하면 트리는 더이상 연결되지 않는다.**

**=> 트리가 비순환일때 간선 하나를 추가하면 트리는 순환을 갖는다.**

**(7) 그래프에서 간선의 갯수**

**- 방향성 그래프 : E <= V^2**

**- 비방향성 그래프 : E <= V(V-1)/2**

**2) 그래프의 표현 - 인접리스트 vs 인접행렬**

- 인접리스트 공간복잡도 O(V+E)

- 인접행렬 공간복잡도 O(V^2)

**(1) 저장공간 :**

- G가 성기면 인접리스트가 낫다. 왜냐하면 E<V^2

- G가 촘촘하면 인접행렬이 낫다. 행렬은 1비트만 사용하므로.

**(2) 간선 찾는데 걸리는 시간 :**

- 인접리스트 : O(V) time

- 인접행렬 : O(1) time

**리스트의 단점은 동적이기 때문에, 검색에서 성능이 안좋다. 엣지의 수가 많으면 성능안좋음**

**인접행렬은 엣지의 수가 적으면 0이 너무 많아서 공간복잡도가 높다. 검색속도 좋다.**

**그러나, 엣지가 5%만 차 있더라도 행렬이 낫다.**

**(3) 가중그래프**

- 간선이 숫자로 표현되는 값을 가지는 그래프.

- 인접행렬은 상관없으나, 인접리스트에서는 정점 외에 간선의 값을 추가저장한다.

**3) 트리탐색**

​- BFS는 오버로드, DFS는 SCV

**(1) BFS(넓이우선탐색)**

**a. 탐색을 하면서 시작점으로부터 거리도 계산한다.**

- 시작점으로부터의 거리

- 직전정점그래프(predecessor vertex graph) Gk = (Vk, Ek)

**b. 정점의 색 구분**

- 검은색: 방문됨, 회색:이전정점에서 발견됨, 흰색:방문안됨

**c. 수도코드**

=====================================================

BFS(G,s)

//값들 초기화

for each vertex u : G.V

u.color = WHITE

u.d = INFINITY

u.파이 = NULL // 직전정점 왜 저장하지?

//BFS 시작점

s.color = GRAY

s.d = 0

s.파이 = NULL

//BFS 수행

Q = EMPTY

ENQUEUE(Q, s)

while (Q != EMPTY)

u = DEQUQUE(Q)

for each v : G.adj[u]

if v.color == WHITE

v.color = GRAY

v.d = u.d+1

v.파이 = u // 직전정점

ENQUEUE(Q,v)

u.color = BLACK

======================================================

**d. 수행시간분석**

- 초기화시간 : O(V)

- 그래프탐색시간 : O(V+E)

정점은 최대 한번만 조사된다. 간선은 최대 두번 조사된다. 그러므로

- 전체 수행시간: O(V+E)

**(2) DFS(깊이우선탐색)**

- 각 정점은 타임스탬프를 두개씩 갖고있다 : v.d 발견시간(회색), v.f 완료시간(검은색)

- 직전정점그래프는 DF숲이 된다.

-

시간 혹은 ㅇㅇ를 기준으로 검색한다.

**4) 최단경로문제의 구분 : 공통적으로 경로와 최솟값 두 개를 구하는 문제이다.**

시작점(source)과 도착점(destination)의 수에 따라 다음과 같이 구분

- 1:1 Single-source & single-destination

- 1:n Single-source

- n:1 Single-destination

- n:n All pairs

- 완화(Relaxation) : 현재 경로값보다 더 적은 경로가 존재하면 값을 변경한다

**(1) 다익스트라 알고리즘**

   - 하나의 시작점에서 하나의 도착점으로 가는 최단경로를 찾는 알고리즘

   - 간선이 음수를 가지면 안된다. (정확히는, 경로에 음수간선과 싸이클을 가지면 안된다. 무한루프 돌면서 답이 계속 작아짐.)

   - 일종의 그리디 알고리즘이다.

**a. 수도코드**

======================================================

DIJKSTRA(G, w, s)

INITIALIZE\_SINGLE\_SOURCE(G, s)

S <- 0

Q <- V[G]

while Q!=0

do u <- EXTRACT\_MIN(Q) // s에서 제일 작은 값 선택

S<-S+u

for each vertex v : Adj[u]

do RELAX(u, v, w)

======================================================

**b. 수행시간**

- 배열로 구현한 경우 O(V^2)

- 힙구조로 구현한경우

- 피보나치힙으로 구현한경우

c. 음수 간선 값

d. 직전 정점 하위그래프

- 최단경로트리가 된다

- 최적해 구조를 가진다

**(2) 벨만-포드 알고리즘**

- 하나의 시작점에서 하나의 도착점으로 가는 최단경로를 찾는 알고리즘

- 음의 간선 있어도 문제해결 가능

edge를 기준으로 버텍스 추가할때마다 간선의 순서에 맞춰서 차례로 반복.

O(v\*2)

a. 수도코드

b. 수행시간: O(VE);

**(3) 플로이드-와샬 알고리즘**

- 인접행렬W, 최단경로행렬D, 직전정점행렬P 사용, 정점의 모든 경로를 실험한다.

- 직전정점행렬P의 값 ​Pij(k) = i에서 j까지 경로에서 k번째연산 중 경로의 j 직전정점의 값.

**최단경로행렬D에 V번의 연산을 통해 최단경로를 찾아감.**

**그 중 k번째연산에서 기존 Dij값과 i -> Vk -> j 의 경로 중 더 작은 경로를 최단경로행렬D에 차곡차곡 저장**

- 수행시간은 O(V^3) <- 행렬크기 V^2, 연산V번

**a. 수도코드**

======================================================

//최단경로 구하기

//※ 최단경로행렬D와 함께 행렬P에도 차곡차곡 저장하는것인가? 수도코드에는 W구현안됨.

FLOYD\_WARSHALL(W)

n <- rows[W]

D(0) <- W

for k <- 1 to n

do for i <- 1 to n

do for j <- 1 to n

do Dij(k) <- min(Dij(k-1), Dik(k-1)+Dkj(k-1))

return D(n)

//직전정점행렬을 통해 경로 출력하기

//πij는 정점 j의 직전정점

PRINT\_ALL\_PAIRS\_SHORTEST\_PATH(π,i,j)

if(i = j)

then print i

else if πij = NULL

then print "no path from" i "to" j "exists"

else PRINT\_ALL\_PAIRS\_SHORTEST\_PATH(P,i,πij)

print j

======================================================

* **그리디 알고리즘 항목**

**최소신장트리(Minimum Spanning Trees)**

​- 가중 무방향성 그래프(Weighted undirected graph) : G(V,E) 의 집합E에 속하는 간선(u,v)는 (v,u)를 가진다.

- 그래프 G의 신장트리(spanning trees) : 그래프 G에 속하며 모든 정점을 포함하는 트리들

- 그리디 알고리즘을 통해서 최소신장트리를 구할 수 있다!

======================================================

a. 수도코드

MST\_PRIM(G, w, start\_index) // Vertex를 따라서 계산

for each u : G.V[]

do key[u] <- INFINITY // key : 각 정점마다의 weight합

P[u] <- NULL // P : 이전정점

key[start\_index] <- 0

Q <- G.V[]

while Q!=EMPTY

do u <- EXTRACT\_MIN(Q) // 그래프에서 간선 짧은놈

for each v : u.Adj[]

do if v isMemberOf(Q) and w(u,v) < key[v] //isMemberOf 확인이 필요하긴 한가?

then P[v] <- u

key[v] <- w(u,v)

MST\_KRUSKAL //Edge를 따라서 계산

======================================================