14장. 그래프 알고리즘

❑ 그래프 알고리즘

- 위상정렬, 최소신장 트리, 최단 경로, 이행폐쇄, 이중 연결, 유니언 파인드, 네트워크 플로우

❑ 학습목표

- 그래프 관련 용어를 이해한다.
- 그래프를 표현하기 위한 두 가지 자료구조를 이해한다.
- 신장트리, 최소 신장트리 알고리즘 들을 이해한다.
- 이행폐쇄, 최단경로 알고리즘 들을 이해한다.
- 이중연결, 유니언 파인드, 네트워크 플로우 알고리즘 들을 이해한다.

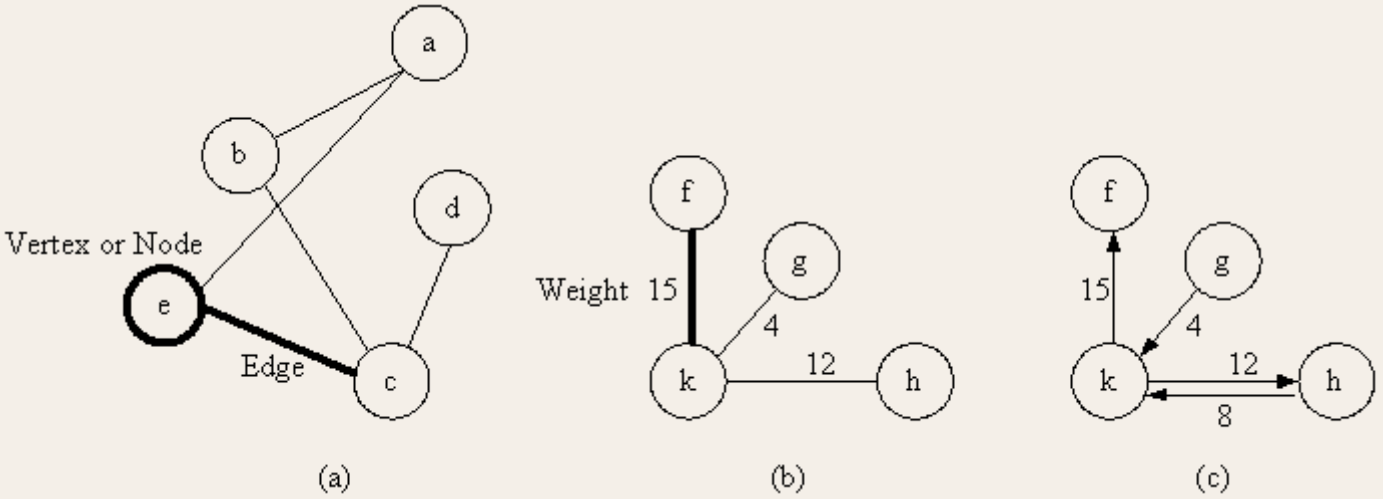
그래프

- 유관한 사물(Object) 또는 개념(Concepts)을 연결. 연결망 (Connection Network 또는 Network). 프로젝트 단위작업 (Task) 사이의 순서, 분자 연결 구조, 전자 부품 연결 구조. 전산학, 이산수학의 연구 주제

Graph	Vertices	Edges
통신	전화, 컴퓨터	광섬유 케이블
전기전자 회로	칩, 콘덴서, 저항	선
기계	이음새	스프링, 빔, 막대
급수시스템	저수지, 정화시설	배관
교통	교차로, 공항	고속도로, 항로
일정표	단위작업	선결 조건
소프트웨어 시스템	함수	함수호출
인터넷	웹 페이지	하이퍼 링크
게임	말의 위치	허용된 움직임
분자구조	분자	화학적 연결
사회적 관계	사람	친구관계
경제	주식, 현금	거래

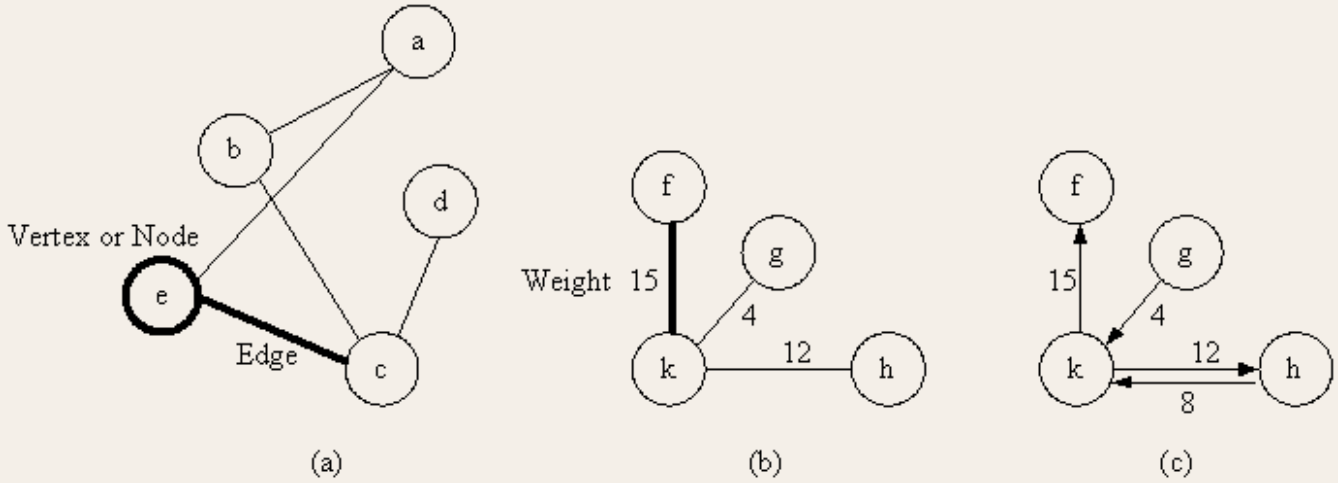
그래프 용어

- G = 정점(Vertex, Node)의 집합 V + 간선(Edge)의 집합 E
- 서브 그래프 G'(Subgraph)
- 인접(Adjacency)



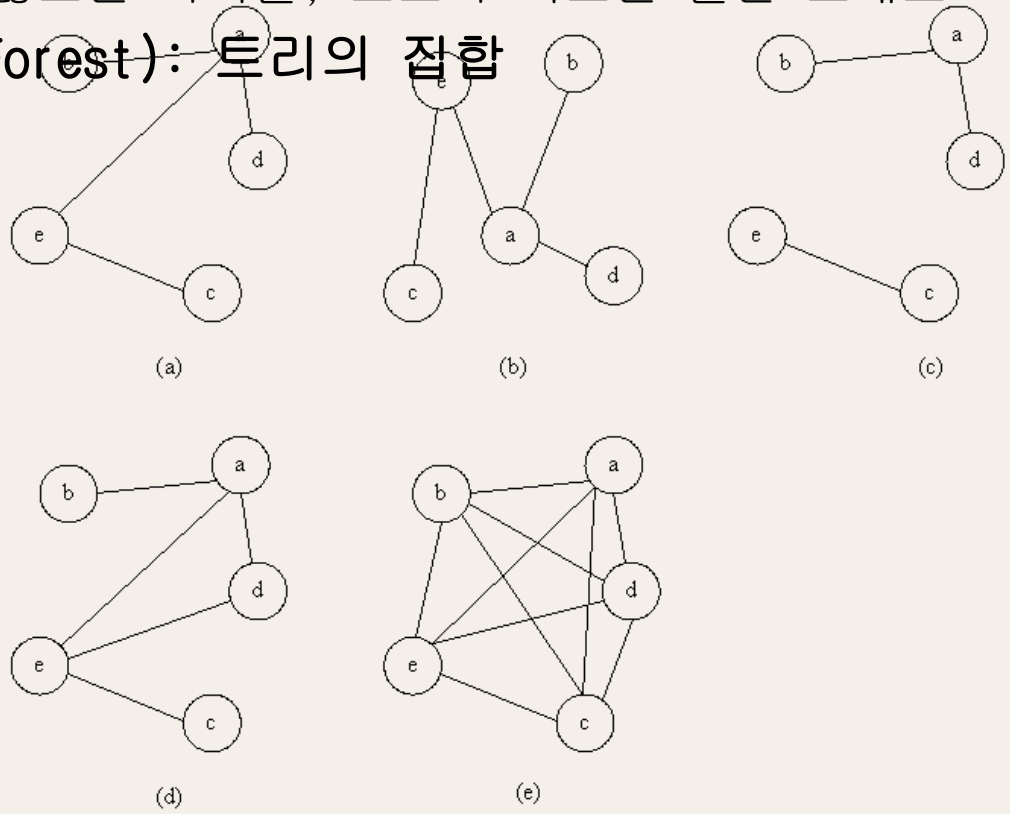
그래프 용어

- 가중치 그래프 (Weighted Graph)
- 무방향 그래프, 방향 그래프(Directed Graph, Undirected Graph)
- 경로(Path)
- 단순경로(Simple Path)
- 사이클(Cycle)
- 단순 사이클(Simple Cycle)



그래프 용어

- 연결 그래프(Connected Graph), 절단 그래프(Disconnected Graph)
- 완전 그래프(Complete Graph)
- 트리(Tree) : 연결된,사이클 없는(Directed Acyclic) 그래프
  - V 개의 정점, 항상 (V-1)개의 간선
  - 그보다 많으면 사이클, 그보다 적으면 절단 그래프
- 포리스트(Forest): 트리의 집합



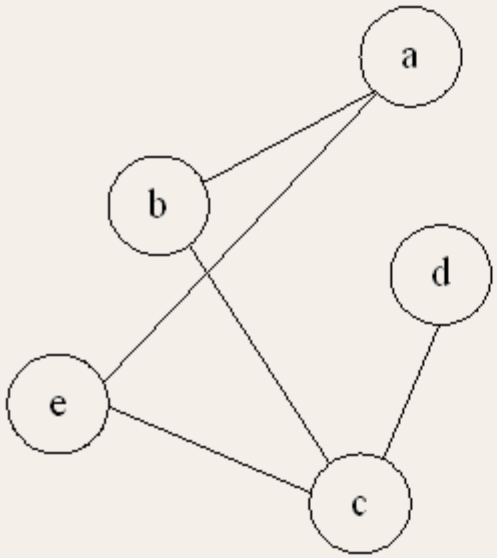
추상자료형 그래프

- Create: 새로운 그래프를 만들기
- Destroy: 사용되던 그래프를 파기하기
- InsertVertex: 새로운 정점을 삽입하기
- InsertEdge: 새로운 간선을 삽입하기
- DeleteVertex: 기존 정점을 삭제하기
- DeleteEdge: 기존 간선을 삭제하기
- RetrieveVertex: 키에 의해 정점 레코드를 검색하기
- IsAdjacent: 인접해 있는지 확인하기
- IsEmpty: 비어있는 그래프인지 확인하기

그래프 표현

❑ 인접행렬(Adjacency Matrix)

- 2차원 행렬 : 2차원 배열
- 직접 연결된 간선이 있으면 해당 값을 1(True)
- 대각선 요소는 무의미. 0 또는 1.

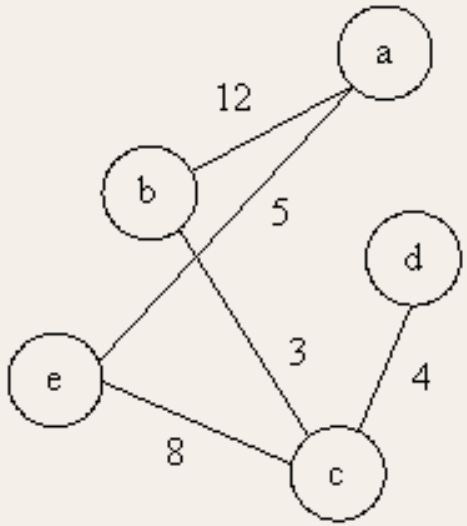


	a	b	c	d	e
a	0	1	0	0	1
b	1	0	1	0	0
c	0	1	0	1	1
d	0	0	1	0	0
e	1	0	1	0	0

그래프 표현

❑ 인접행렬

- 가중치 그래프: 간선의 가중치
- 가중치: 정점 사이를 이동하는데 필요한 비용이나 거리
- 인접하지 않은 정점: 비용 무한대.



from \ to	a	b	c	d	e
a	0	12	inf	inf	5
b	12	0	3	inf	inf
c	inf	3	0	4	8
d	inf	inf	4	0	inf
e	5	inf	8	inf	0

그래프 표현

❑ 인접행렬

- 무방향 그래프의 인접행렬은 대각선을 중심으로 대칭
- 메모리 절약을 위해 배열의 반쪽만을 사용할 수 있음.
- 방향성 그래프의 인접행렬은 일반적으로 대칭이 아님.
- 배열의 인덱스는 정점을 의미
- 정점 ID를 배열 인덱스로 매핑 시키는 테이블이 필요

Vertex	배열 인덱스
a	0
b	1
c	2
d	3
e	4

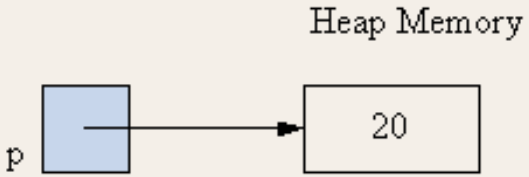
그래프 표현

- 정적배열에 의한 인접행렬
  - 필요한 정점의 수를 예측. int AdjacencyMatrix[Max][Max]; 라고 선언
  - 스택 메모리
- 동적배열에 의한 인접행렬
  - 힙 메모리
  - 미리 배열 크기를 결정할 필요가 없음.

동적배열에 의한 인접행렬

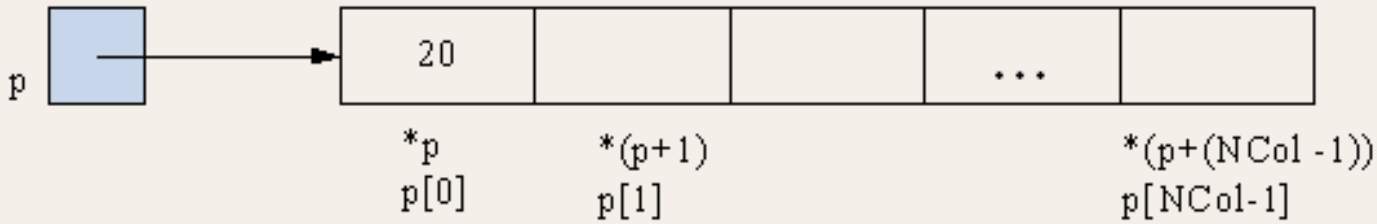
❑ 정수공간

- typedef int * ptrType;
- ptrType p = malloc(sizeof(int));
- *p = 20;



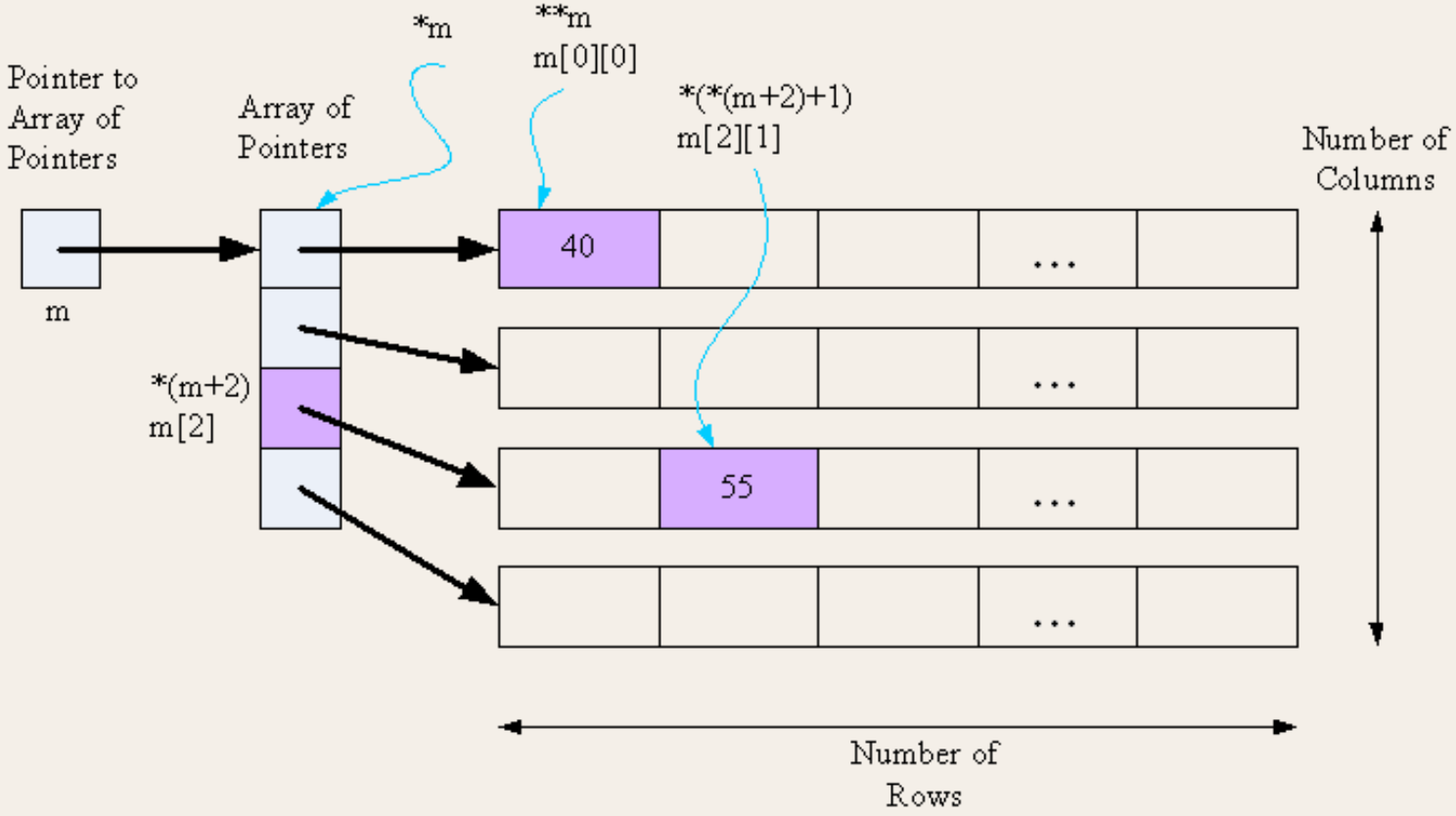
❑ 연속된 정수공간

- ptrType p = malloc(NCol * sizeof(int));
- NCol 개의 정수가 들어갈 수 있는 공간.
- 이 변수공간을 마치 배열처럼 사용
- 포인터 기호 또는 배열 기호에 의해 접근가능



동적배열에 의한 인접행렬

- 행렬: NRow * NCol 개의 정수공간



동적배열에 의한 인접행렬

- 코드 14-1: 동적배열에 의한 행렬

```
int ** InitMatrix(int NRow, int NCol, int Value)   Value는 초기값
{ int **m;
  m = malloc(NRow * sizeof(int *));        NRow개의 포인터 배열을 만듦
  for (int i = 0; i < NRow; i++)                   포인터 배열의 각 요
    소가
     m[i] = malloc(NCol * sizeof(int));    NCol개의 정수 배열을 가리킴
  for (int i = 0; i < NRow; i++)                   모든 행에 대해서
      for (int j = 0; j < NCol; j++)               모든 열에 대해서
          m[i][j] = Value;                         Value 값으로 초기화
  return m;                                포인터의 포인터를 리턴 값으로
}
```

동적배열에 의한 인접행렬

❑ 코드 14-2: 동적배열의 공간반납

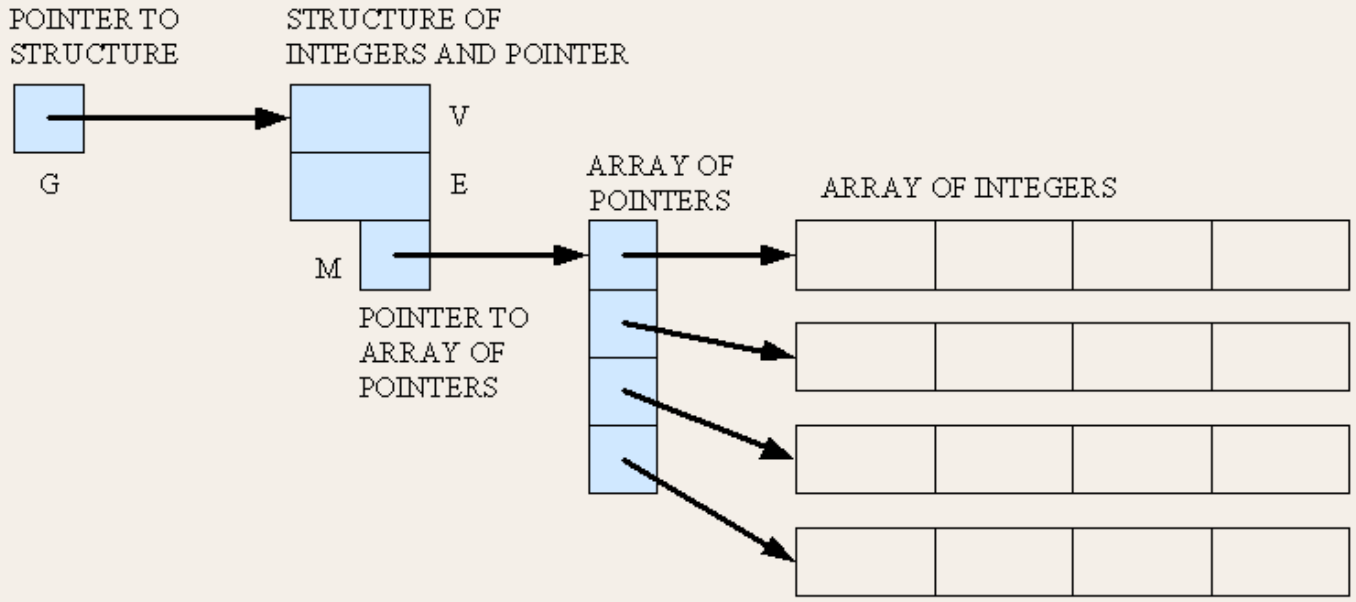
```
void FreeMatrix(**m)                    동적배열 m의 파기
{ for (int i = 0; i < NRow; i++)              포인터 배열 각각
   에 대해
    free(m[i]);                 가리키는 전체 공간을 반납
  free(m);                      포인터 배열 공간을 반납
}
```

❑ 그래프 표현

- 정방형 인접 행렬(Square Adjacency Matrix)
- 정점 및 간선 개수를 추적
- 그래프 자료구조는 구조체와 그 구조체를 가리키는 포인터로 선언

그래프 표현

```
typedef struct {
   int V;          그래프 내부 정점의 수
   int E;          그래프 내부 간선의 수
   int **M;        인접행렬을 가리키는 포인터
} graphType;
typedef graphType *graphPtr;          그래프를 가리키는 포인터
graphPtr InitGraph(int V)             함수 프로토타입
void InsertEdge(graphPtr g, int V1, int V2) 함수 프로토타입
```



그래프 함수

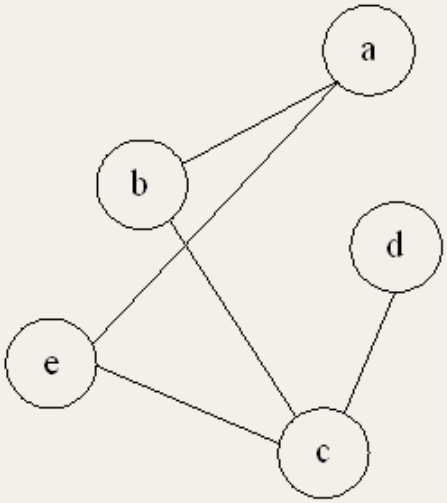
- 코드 14-4: Graph.c

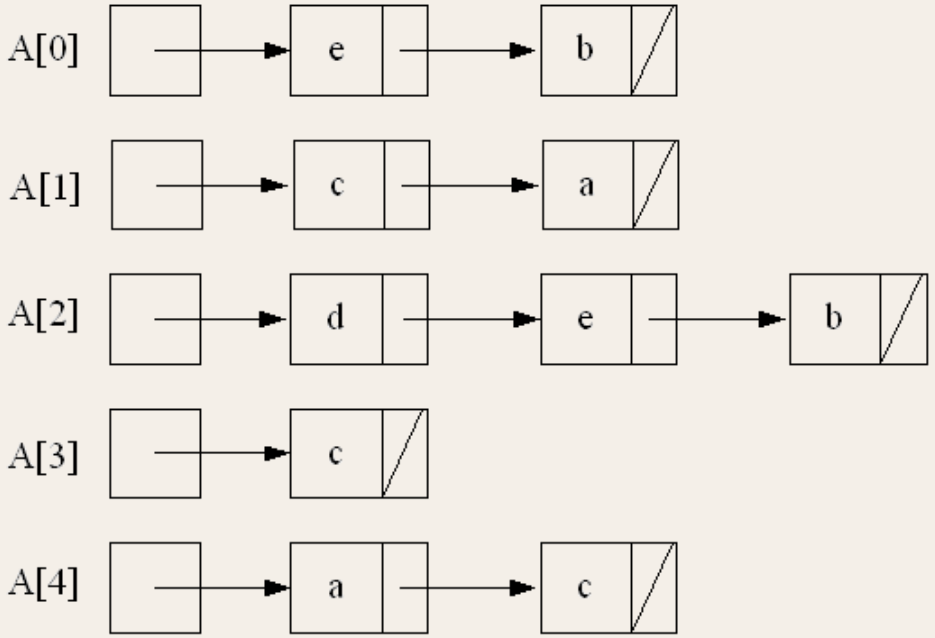
```
#include <GraphByMatrix.h>
graphPtr InitGraph(int V)      정점 V개가 들어갈 수 있는 그래프 생성
{ graphPtr G = malloc(sizeof(graphType));  그래프 구조체를 동적으로
   생성
  G->V = V;                          정점의 수를 V개로 확정
  G->E = 0;                          생성 시의 간선 수 0
  G->M = InitMatrix(V, V, 0);  크기 V 제곱인 동적배열 만들고 0으로 초
   기화
  return G;
}

void InsertEdge(graphPtr g, int V1, int V2) 정점 V1, V2를 연결하는 간
   선 삽입
{ if (g->M[V1][V2] = = 0)      인접행렬 값이 0이면
  {  g->E++;                         간선의 수를 늘리고
    g->M[V1][V2] = 1;          V1->V2를 1로 바꿈
    g->M[V2][V1] = 1;          무방향 그래프이면 V2->V1도 1로 바꿈
  }
}
```

그래프 표현

❑ 인접 리스트(Adjacency List)

- 하나의 정점에 인접한 모든 노드를 연결 리스트 형태로 표시
- 경로에 관한 정보가 아님.
- 가중치 정보를 표시하려면 노드에 가중치 필드를 추가.
- 연결 리스트를 가리키는 포인터 배열





인접 리스트

❑ 인접 리스트

- 무방향 그래프: 하나의 간선에 대해 두 개의 노드가 나타남.
- 인접 리스트의 노드 수는 간선 수 E의 2배

❑ 인접 리스트와 인접 행렬

- 정점 i와 정점 j가 인접해 있는가: 인접행렬이 유리
- 정점 i 에 인접한 모든 노드를 찾아라: 인접 리스트가 유리
- 공간 면에서 인접행렬은 V^2 개의 공간, 인접 리스트는 2E 개의 공간이 필요

❑ 희소 그래프, 조밀 그래프

- 간선 수가 적은 그래프를 희소 그래프(稀少, Sparse Graph)
- 간선 수가 많은 그래프를 조밀 그래프(稠密, Dense Graph)
- 조밀 그래프라면 인접행렬이 유리
- 희소 그래프라면 인접 리스트가 유리

인접 리스트

- 코드 14-5: GraphbyLinkedList.h

```
typedef struct {
   int Data;            정점의 ID 번호
   node* Next;          다음 노드를 가리키는 포인터
} node;
typedef node* Nptr;

typedef struct {
   int V;               그래프 내부 정점의 수
   int E;               그래프 내부 간선의 수
   node **L;            포인터 배열을 가리키는 포인터
} graphType;
typedef graphType *graphPtr;              그래프를 가리키는 포인터

graphPtr InitGraph(int V)                 함수 프로토타입
void InsertEdge(graphPtr g, int V1, int V2)       함수 프로토타입
```

인접 리스트

- 코드 14-6: GraphbyLinkedList.c

```
#include <GraphbyLinkedList.h>
graphPtr InitGraph(int V)          정점 V개가 들어갈 수 있는 그래프 생성
{ graphPtr G = malloc(sizeof(graphType)); 구조체 공간을 동적으로 생성
  G->V = V;                        정점의 수를 V개로 확정
  G->E = 0;                        생성 시에 간선 수 0
  G->L = malloc(V * sizeof(node *)); V개  정수 포인터 배열을 동적으로
    생성
  for (int i = 0; i < V; i++)      배열 내 모든 포인터 값을
    G->L[i] = NULL;                널로 초기화
  return G;                        포인터의 포인터를 리턴 값으로
}
void InsertEdge(graphPtr g, int V1, int V2) 정점 V1, V2를 연결하는 간
    선 삽입
{ Nptr temp = malloc(sizeof(node)); 새로운 노드를 만들고
  temp->Data = V2;                 거기에 V2의 ID 번호를 넣음
  temp->Next = g->L[V1];           새 노드가 현재 첫 노드를 가리키게
  g->L[V1] = temp;                 L[V1]이 새로만 든 노드를 가리키게
}                                  무방향 그래프이면 V2에도 V1을 삽입.
```

그래프 순회

- 깊이우선 탐색(DFS)과 너비우선 탐색(BFS)
  - 깊이우선 탐색은 스택과, 너비우선 탐색은 큐와 직접 연관
  - 트리의 전위순회를 일반화 한 것이 깊이우선 탐색
  - 트리의 레벨순회를 일반화 한 것이 너비우선 탐색

- 두가지 모두 방문된 노드로는 가지 않음
  - 확인을 위한 자료구조가 필요
  - 배열 값이 TRUE이면 방문된 상태
  - 미 방문 상태로 초기화하기 위해 FALSE를 쓰는 작업. 방문 될 때마다 TRUE를 쓰는 작업. 따라서 O(2V)번의 작업이 필요.

IsVisited[ ]

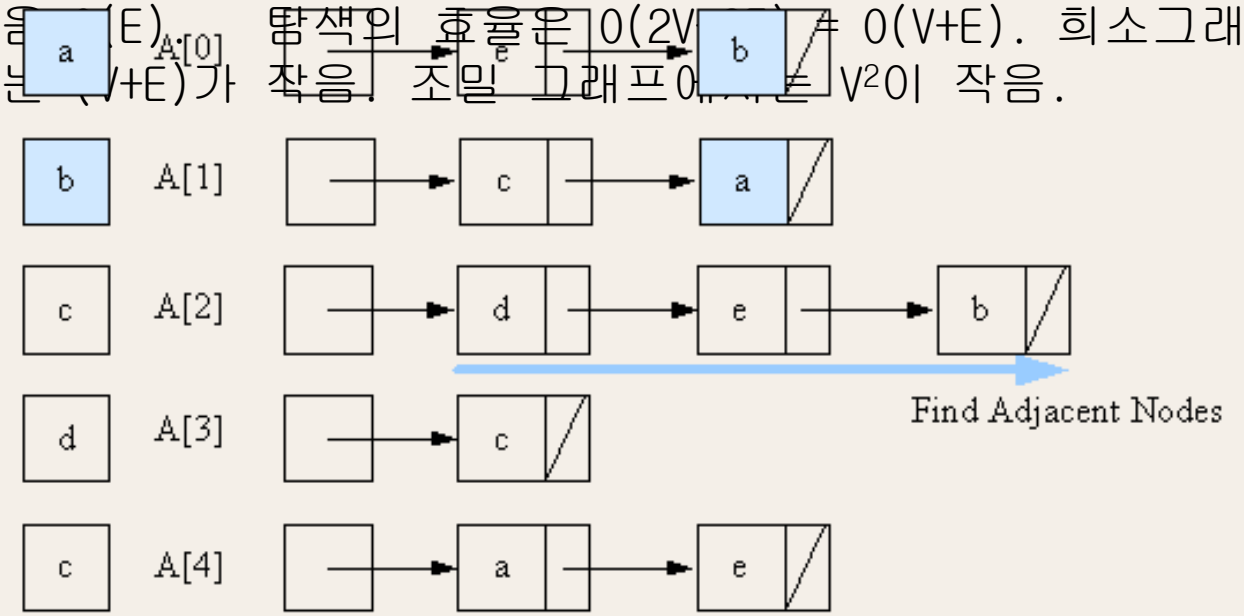
0	1	2	3	4	5	..	V-1
F	F	F	F	F	F	...	F

Mark FALSE(Unvisited)

Mark TRUE(Visited)

그래프 순회의 효율

- **인접행렬**
  - 주어진 노드에 인접한 모든 노드를 찾기 위해서 그 노드를 행 번호로 하는 행 전체를 스캔 O(V). 모든 행을 스캔 $O(V^2)$. 탐색의 효율은 $O(2V + V2) = O(V^2)$
- **인접 리스트**
  - 현재 노드가 A일 때, A를 헤드로 해서 B가 방문 되었는지 확인하기 위해 리스트 스캔 O(E). 나중에 현재 노드가 B일 때, B를 헤드로 해서 A가 이미 방문되었는지 확인하기 위해서 한 번 더 읽음 O(E). 탐색의 효율은 O(2V + E) = O(V+E). 희소그래프에서는 (V+E)가 작음. 조밀 그래프에서는 V^2이 작음.



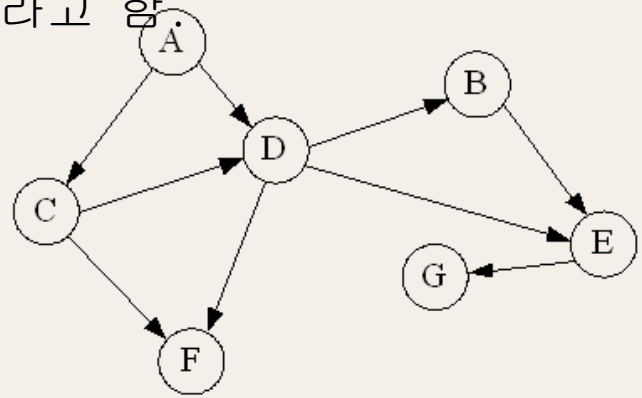
위상정렬

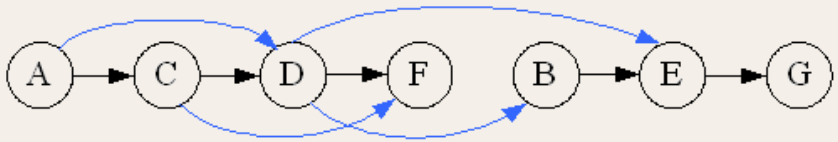
AOV 네트워크(Activity on Vertex Network)

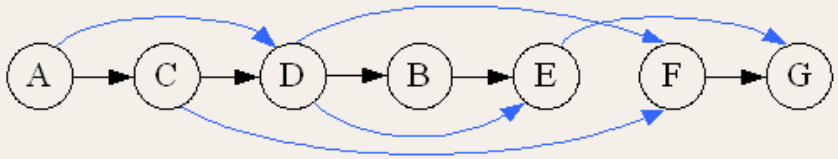
- 정점에 단위공정을 표시
- 방향성 그래프의 간선은 일의 순서를 의미
- A->B는 A 공정이 완료되어야 B 공정을 수행할 수 있음을 의미
- 수강신청을 위한 선수과목 이수 개념과 동일함.
- 어떤 공정이 어떤 공정보다 앞서야 하는지가 문제
- 위상정렬: 먼저 수행해야 될 공정부터 시작해서 일렬로 모든 공정을 나열

위상정렬

- 정렬 결과가 유일하지는 않음.
- 첫 번째 위상정렬 결과는 A, C, D, F, B, E, G 순서로 일을 진행. 결과 화살표는 노드를 건너 뛸 수는 있으나 오른쪽으로 가는 것만 허용. A-D-E-G로 가든 A-C-D-B-E-G로 가든 무방
- 전제조건: 사이클이 존재하면 서로 어떤 것이 먼저인지 판별하기가 불가능. 사이클이 없는 방향 그래프를 대그(DAG: Directed Acyclic Graph)라고 함







소스, 싱크

싱크

- 마지막 공정은 그 공정 다음에 오는 공정이 없음.
- 싱크(Sink, 가라앉는 곳)
- 그 노드로 들어가는 간선은 있어도, 거기서 나가는 간선은 없는 노드

소스

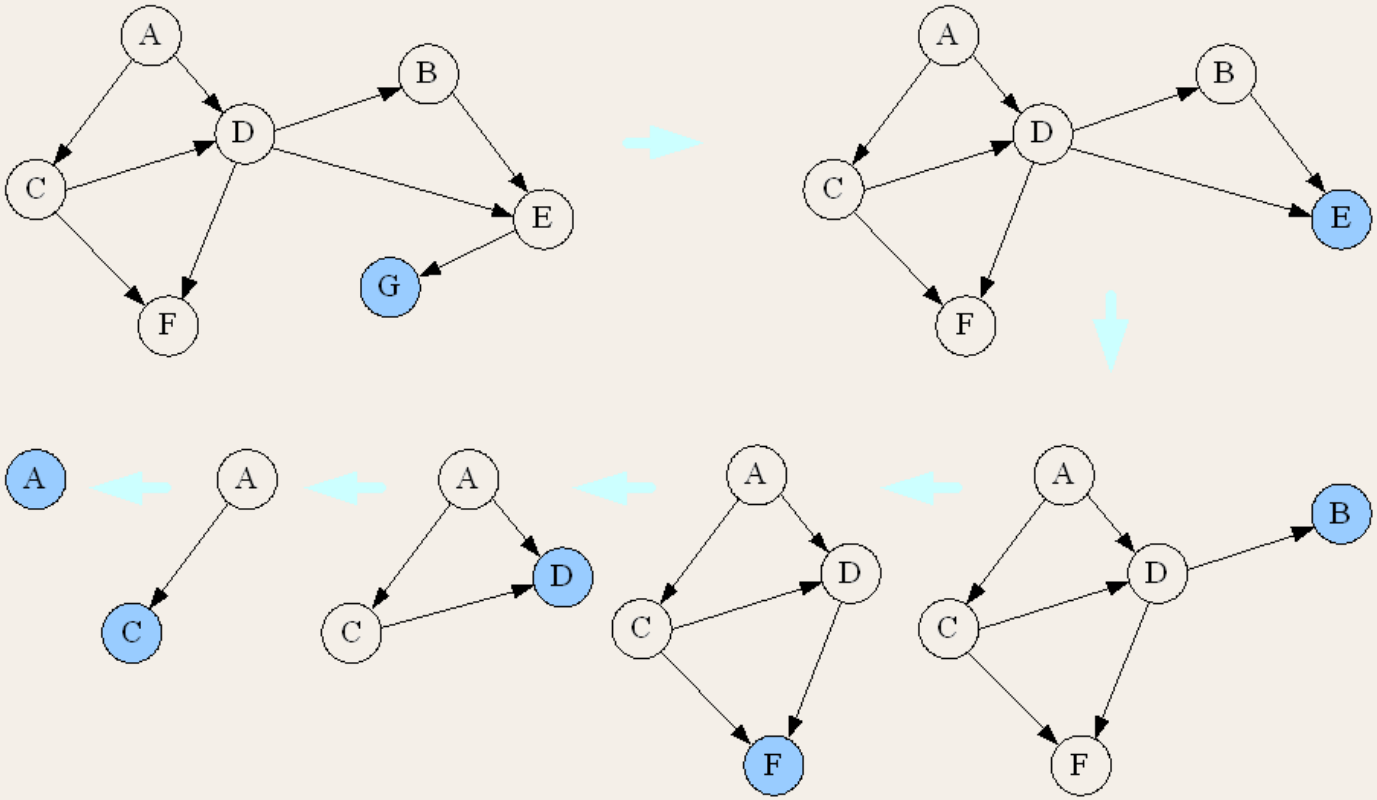
- 첫 공정은 그 공정에 앞서 오는 공정이 없음.
- 소스(Source, 떠오르는 곳)
- 그 노드로부터 나가는 간선은 있어도, 그리고 들어오는 간선이 없는 노드

진입차수, 진출차수

- 진입차수(進入, In-Degree): 어떤 노드로 들어가는 간선의 수
- 진출차수(進出, Out-Degree): 노드에서 나가는 간선의 수
- 싱크는 진출차수가 0인 노드, 소스는 진입차수가 0인 노드.

싱크 제거에 의한 위상정렬

- 싱크인 G를 제거하고 G와 연결된 간선들을 모두 제거. 결과 그래프의 싱크는 E와 F. E와 거기에 연결된 간선들을 제거.
- 순차적으로 제거된 싱크를 연결 리스트의 맨 처음에 삽입
- 연결 리스트를 처음부터 끝까지 따라가면 그 순서가 위상정렬 순서(A, C, D, F, B, E, G)



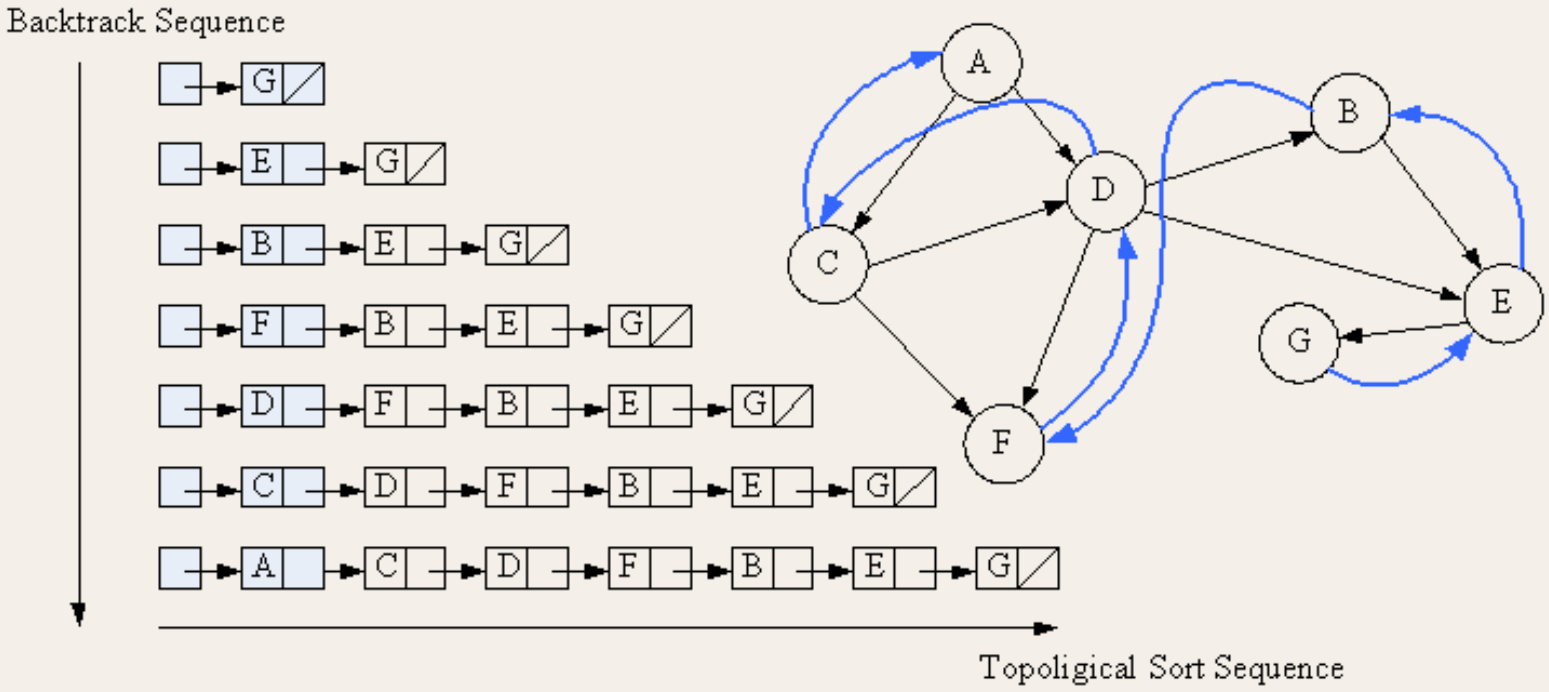
싱크 제거에 의한 위상정렬

❑ 코드 14-7: 싱크 제거에 의한 위상정렬

```
TopologicalSort(G)
{  for (step =1 through N)          N은 전체 노드의 개수
   { Select a Sink T;               싱크가 여럿이면 아무거나
     L.Insert(1, T);                연결 리스트의 맨 앞에 삽
   입
     Remove the Sink T and Edges Connected to It: 싱크 및 연결
   된 간선을 삭제
   }
}
```

깊이우선 탐색에 의한 위상정렬

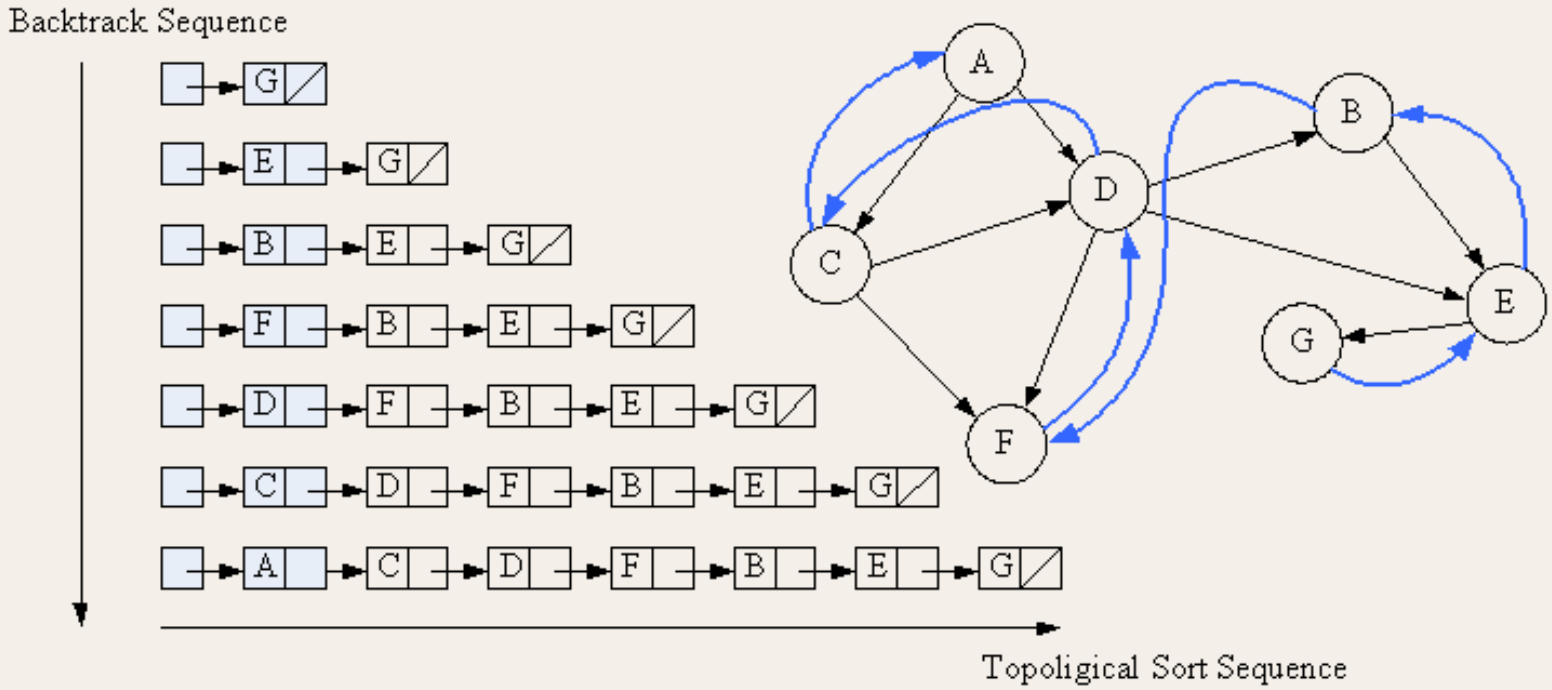
- 소스에서 출발하여 방향성 간선(Directed Edge)을 계속 따라가면 그 순서가 바로 위상정렬 순서
- 마지막 노드인 싱크로부터 백트랙 하는 작업이 바로 싱크를 제거하는 작업에 해당.



깊이우선 탐색에 의한 위상정렬

❑ 유의점

- 깊이우선 탐색의 순서는 A-D-B-E-G-F-C
- 백 트랙이 발생한 순서대로 기록. 탐색순서 A-D-B-E-G-F-C에서 가장 먼저 백 트랙이 발생한 곳은 G. 이후 G, E, B.
- 이후 가정 먼저 백 트랙이 발생한 곳은 F.



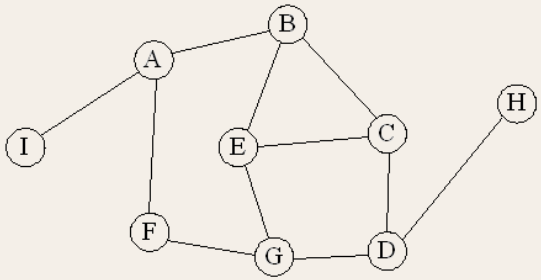
깊이우선 탐색에 의한 위상정렬

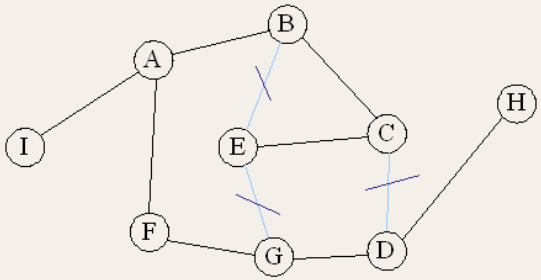
❑ 코드 14-8: 깊이우선 탐색에 의한 위상정렬

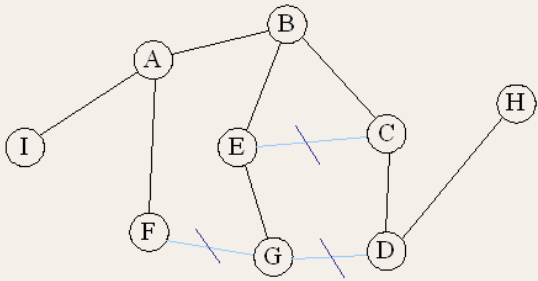
```
TopologicalSort(V)
{  for All Nodes T Adjacent to V      안 가본 모든 노드에 대해
     if (T Is Unvisited)                더 이상 갈 곳이 없을 때
   까지
       TopologicalSort(T);              재귀호출
   L.ListInsert(1, V);                  재귀호출 끝나면 리스트
   맨 처음에 삽입
}
```

트리

- 트리
  - 연결된(Connected), 사이클 없는(Acyclic) 그래프
- 신장트리
  - 주어진 그래프 G의 신장트리는 G의 서브 그래프로서 G의 모든 정점과 트리를 이루기에 충분할 만큼만의 간선으로 구성.
  - 주어진 연결 그래프에서 사이클을 없앰으로써 얻을 수 있는 트리. 정점수 N개일 때, 간선 수 (N-1)이면 트리
  - 신장트리의 모습은 유일하지는 않다.



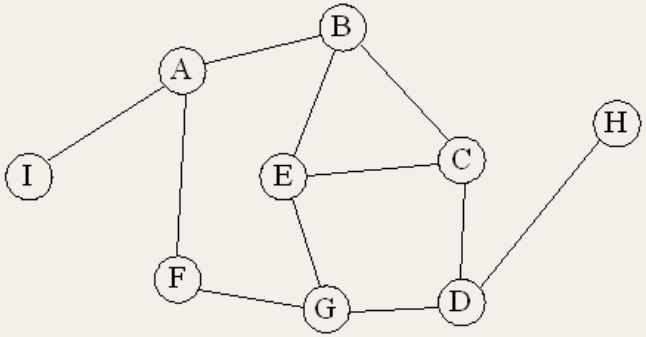


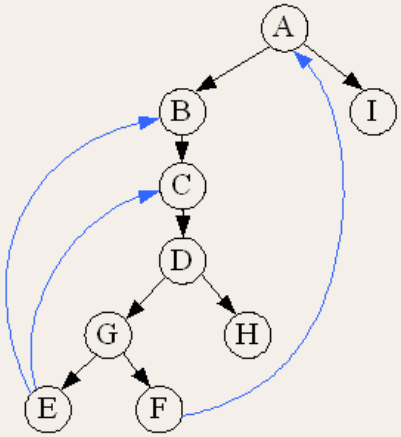


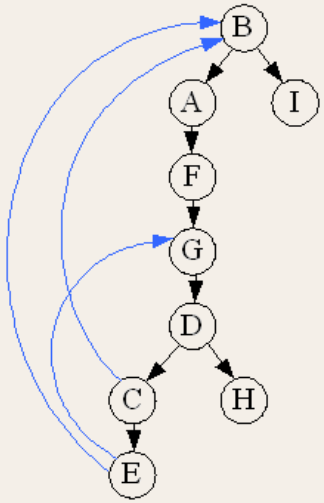
깊이우선 탐색에 의한 신장트리

- 깊이우선 탐색트리(Depth First Search Tree)
  - A에서 시작해서 A-B-C-D-G-E-F-H-I 로 가는 탐색순서
  - B에서 시작해서 B-A-F-G-D-C-E-H-I로 가는 탐색순서

- 아래에서 위로 가는 화살표
  - 지나온 노드이기 때문에 새로 방문하지는 않지만 보이기는 보임.
  - 화살표에 해당하는 간선을 제거하면 신장트리







최소 신장트리

- 전철 망의 설계
  - 모든 역을 서로 직접 연결하면 가장 이상적
  - 장애물이 있을 때에는 직접 연결이 가능한 곳이 제한됨

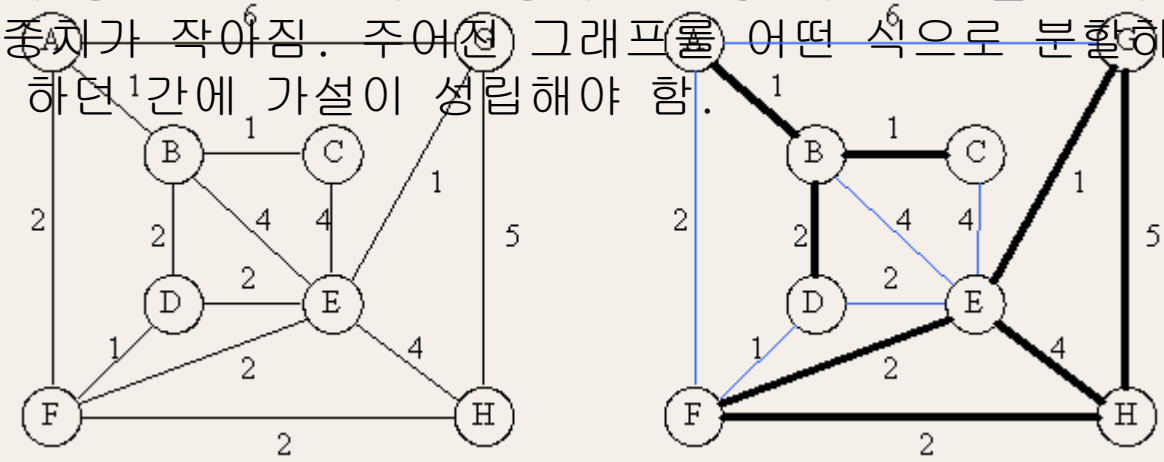
최소 신장트리

- 모든 역 사이에 서로 갈 수 있는 길이 있는가.
  - 다른 역을 거쳐서 가도 됨.
  - 연결 그래프 이어야 함.

- 선로 가설비용을 최소화하라.
  - 사이클이 존재한다면 그것을 제거함으로써 가설비용을 줄임
  - 연결성은 그대로 유지. 따라서 신장 트리 만들면 됨.
  - 선로마다 가설비용이 다른데, 어떤 간선을 제거해야 하는 것이 전체 가설비용을 최소화하는가. 즉, 어떤 신장 트리가 가장 경제적인가의 문제

최소 신장트리

- 최소 신장트리(MST: Minimum Spanning Tree)
  - 최소비용 신장트리.
  - 신장트리 중 전체 가중치의 합이 최소가 되는 것
- 최소 신장트리 가설
  - “그래프의 정점들을 두 개의 그룹으로 나누었을 때, 두 그룹 사이를 연결하는 간선들 중에서 가장 가중치가 작은 간선은 반드시 최소 신장트리에 포함되어야 한다.”
  - {A, B, C, D}와 {E, F, G, H} 그룹. 사이를 잇는 것은 B-E, C-E, D-E, D-F, A-F, A-D 등 6개의 다리. 2개의 다리를 모두 트리에 포함시키면 사이클.
  - 다리 중 하나만 선택. 가중치가 가장 작은 D-F를 선택해야 전체 가중치가 작아짐. 주어진 그래프를 어떤 식으로 분할하여 그룹화 하더라도 이 가설이 성립해야 함.



프림 알고리즘

프림 알고리즘(Prim's Algorithm)

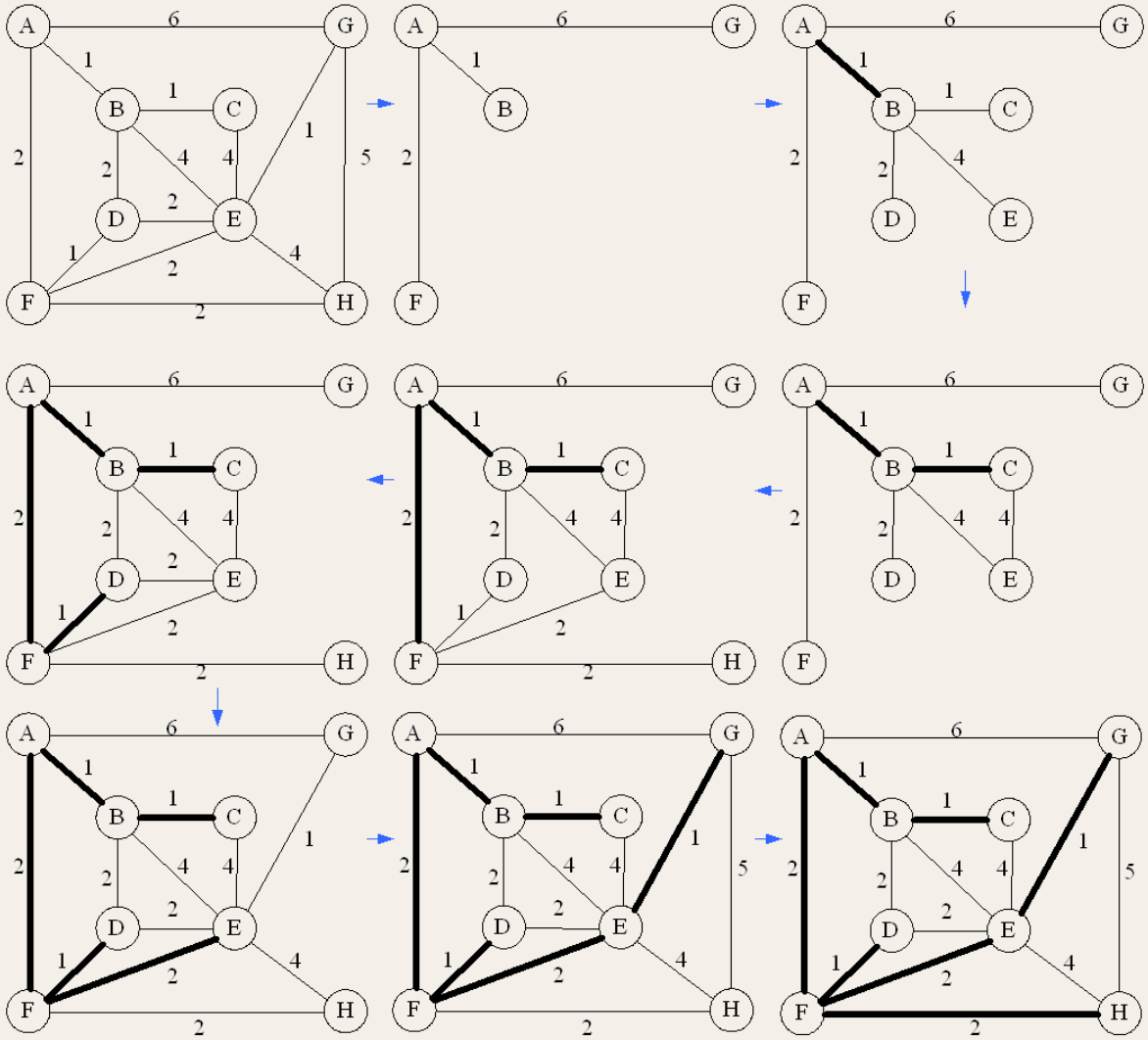
- 우선순위 우선탐색(PFS: Priority First Search)
- 가중치가 작을수록 우선순위가 높다고 정의

적용예

- 어떤 노드에서 출발해도 무관. A에서 출발하는 예
- A와 인접한 세 개의 노드 G, B, F 중 가장 가중치가 작은 것이 B
- 간선 A-B를 트리에 포함.
- 이 트리의 정점 A와 B에 인접한 모든 노드를 대상으로 가중치를 비교.
- B-C, A-G, A-F 중 가장 작은 것은 B-C
- 간선 B-C를 트리에 포함
- 현재 연결된 트리 내부의 모든 정점에 인접한 간선 중 최소 가중치 간선을 선택함. 단, 사이클을 유발하는 간선은 피함.
- 우선순위 큐를 사용하여 구현
  - A를 제거하는 대신 A에 인접한 G, B, F를 우선순위 큐에 삽입.
  - 우선순위가 높은 B를 삭제하는 대신, B에 인접한 C, E, D를 삽입.

프림 알고리즘

- 프림 알고리즘



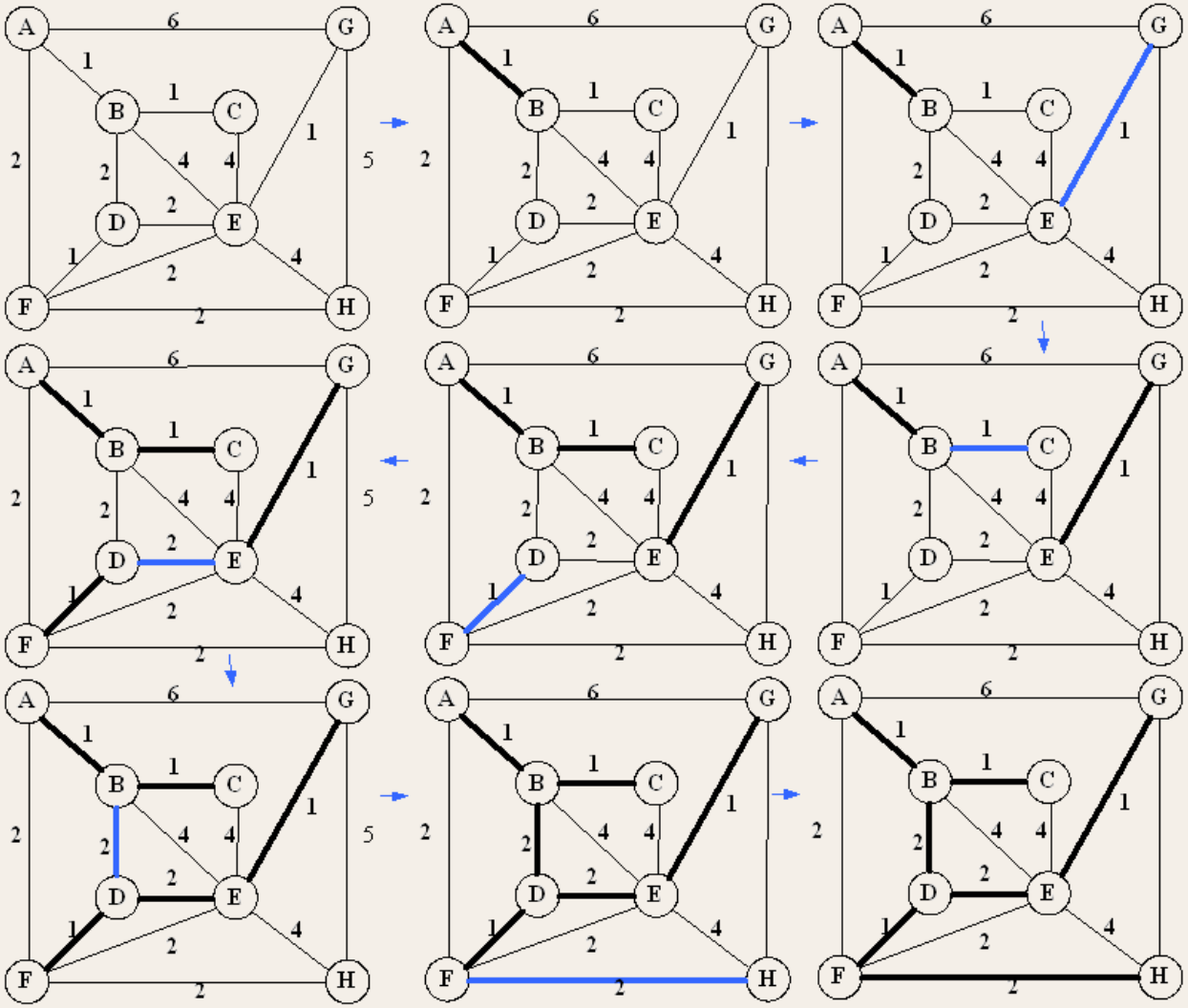
크루스칼 알고리즘

❑ 크루스칼 방법(Kruskal's Method)

- 가장 작은 가중치 1을 지닌 간선 네 개 중 임의로 A-B를 선택
- 나머지 그래프에서 가장 작은 가중치 역시 1이므로 이번에는 G-E를 선택
- 분리된 두 개의 트리(포리스트)가 생성
- 매 단계마다 그 상태에서 가장 작은 가중치를 가진 간선을 선택해서 두 개의 정점 또는 트리 사이를 잇는 방식. 단 사이클을 유발하는 간선은 피함.
- 탐욕 알고리즘(Greedy Algorithm)
  - “일단 가설비용이 제일 싼 것부터 먼저 건설하고 보자”
  - “당장 눈앞에 보이는 이득을 추구하는 것이 나중에 전체적으로도 크게 봐도 이득이 된다”는 접근 방식
  - 최소 신장트리에는 이러한 접근 방법이 해결책이 됨.

크루스칼 알고리즘

- 크루스칼 알고리즘



솔린 알고리즘

❑ 솔린 알고리즘(Sollin's Algorithm)

- 트리에 인접한 최소 가중치 간선을 사용
- 두 개의 서로 다른 트리를 연결

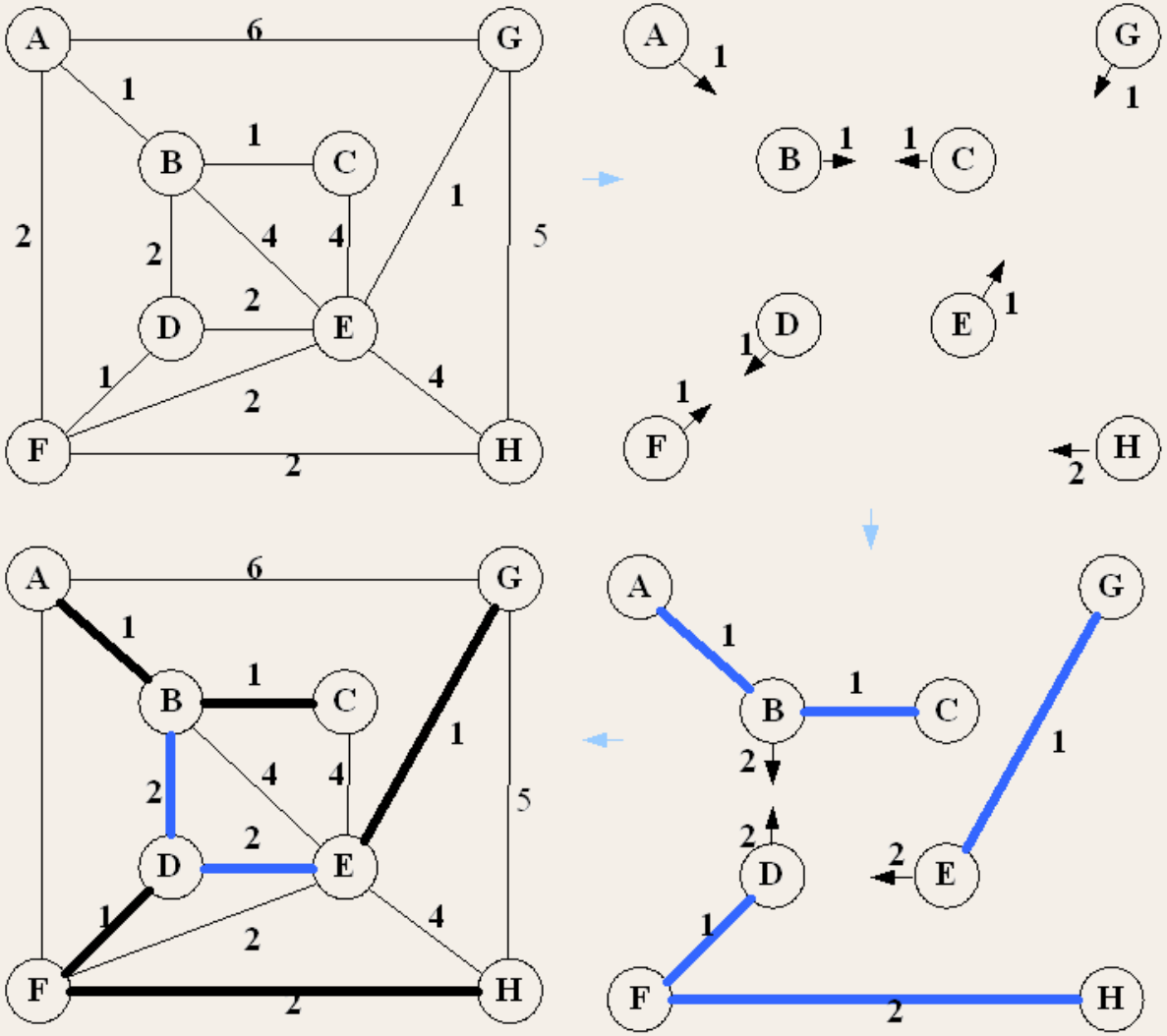
- 1 단계에서 그래프의 모든 정점이 각각 트리를 형성
- 트리에 인접한 간선 중, 가장 가중치가 작은 것을 선택하여 화살표로 표시
- 트리별로 간선을 따라 다른 트리와 연결

- 다시 각각의 트리에 인접한 간선들 중 가장 가중치가 작은 것을 선택
- 단계별로 각각의 트리에서 뻗어나가는 최소 가중치 간선을 선택하고, 이를 선택하여 서로 다른 트리를 이어감. 단 사이클을 유발하는 간선은 피함.

- 트리에 인접한 간선 중에서 최소 가중치를 지닌 간선을 선택한다는 점에서 프림 알고리즘과 유사. 포리스트를 놓고 작업한다는 점에서는 크루스칼 알고리즘과 유사.

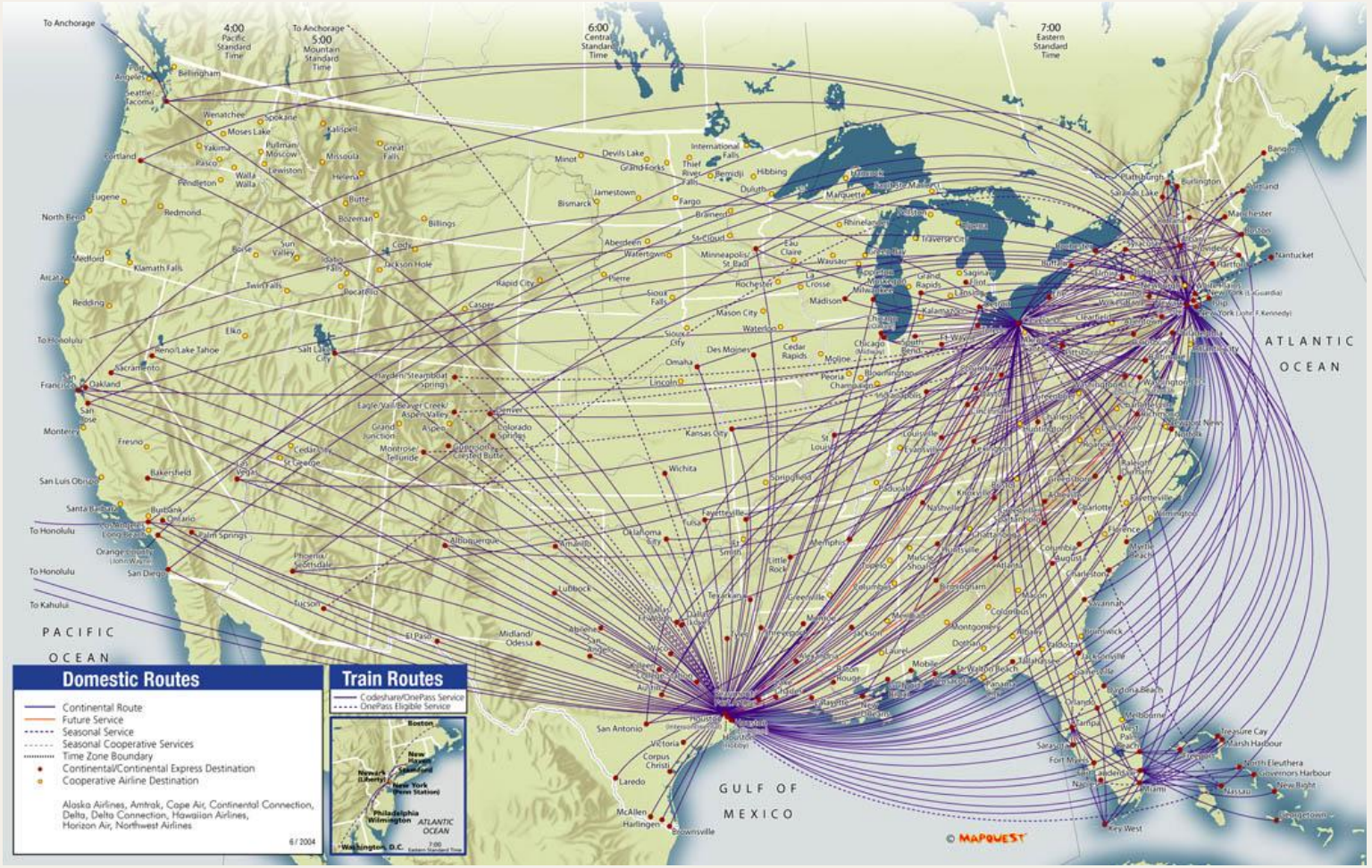
솔린 알고리즘

- 솔린 알고리즘



최단경로

- 주어진 도시에서 다른 도시로 가기 위해서 어떤 경로로 가는 것이 가장 적은 비용이 되는가.



최단경로

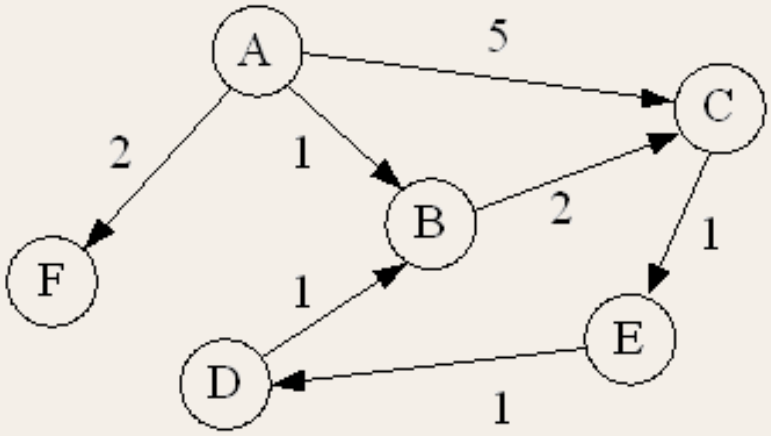
- 최단경로


열번을 갈아타도 좋으니
가장 싸게 가는 방법이
뭡니까?
여행 약
NEW York
Tokyo . .
FiRankFrut . .
Paris . .
London . . .

다이익스트라 알고리즘

❑ 최단 경로의 문제(Shortest Path Problem)

- 여러 곳을 거쳐서 간다면 전체 비용은 구간별 요금을 합산
- 최소비용다이익스트라 알고리즘(Dijkstra Algorithm)
  - 출발 노드가 주어졌을 때 나머지 모든 노드로 가는 최소비용 및 경로.



i / j	A	B	C	D	E	F
A	0	1	5	∞	∞	2
B	∞	0	2	∞	∞	∞
C	∞	∞	0	∞	1	∞
D	∞	1	∞	0	∞	∞
E	∞	∞	∞	1	0	∞
F	∞	∞	∞	∞	∞	0

다이익스트라 알고리즘

- 도표의 1 단계 행(Row)을 채우기 위해서는 인접행렬 정보를 사용
- 정점 중 하나를 선택하여 선택정점 칼럼에 추가
- 선택 정점은 현재의 비용 중 가장 작은 것, 즉 비용 1을 지닌 B. 정점이 선택된다는 것은 해당 정점으로 가는 최소비용이 확정된다는 것을 말함
- 만약 A에서 다른 곳을 거쳐서 B로 간다면 C, D, E, F 중 하나를 거침. 그런데 그곳까지 가는 비용이 이미 1보다 크기 때문에 그곳을 거쳐 온들 이 비용보다 작을 수 없음.

단계 수	선택정점	비용[B]	비용[C]	비용[D]	비용[E]	비용[F]
1	B	1	5	∞	∞	2
2	F		3 B	∞	∞	2
3	C		3 B	∞	∞	
4	E			∞	4 C	
5	D			5 E		

다이익스트라 알고리즘

- 2단계에서는 나머지 C, D, E, F를 처리
- B로 가는 최소비용이 확정되었으니, B를 거쳐서 해당 정점으로 가는 비용이 현재 비용보다 더 작지 않은가를 확인
- 비용[C]: 1단계에서 5원에 갈 수 있음. 그런데 A-B 1원에 간다고 확정되었고, 인접행렬을 참조하면 B-C는 2원에 갈 수 있으니 3원이면 A-B-C로 갈 수 있음. 이 값은 현재 5원보다 작으니 3으로 바꿈.
- 인접행렬을 참조하면 나머지 D, E, F는 B에서 가는 길이 없으니 1단계 값들이 그대로 내려옴. 최소 값인 정점 F를 선택
- 3단계에서는 F를 거쳐서 C, D, E가는 비용을 확인. 인접행렬을 보면 F에서 C, D, E로 가는 것은 없으므로 2단계 값들이 그대로 내려오고, 그들 중 최소값 3을 지닌 정점 C를 선택.

단계 수	선택정점	비용[B]	비용[C]	비용[D]	비용[E]	비용[F]
1	B	1	5	∞	∞	2
2	F		3 B	∞	∞	2
3	C		3 B	∞	∞	
4	E			∞	4 C	
5	D			5 E		

다이익스트라 알고리즘

- 4단계에서는 방금 선택된 C를 거쳐서 가는 길을 확인. 3단계에서 A-C를 최소 비용 3원에 가고, 다시 C-E를 1원에 가면 4원. 이 값은 현재 A에서 C가는데 드는 최소 비용인 무한대보다 작음. 따라서 해당 값을 4원으로 변경. 정점 E를 선택. 5단계에서는 정점 D로 가는 최소비용이 확정됨.

- 경로를 찾아내기 위해서 비용이 갱신될 때마다 거쳐 온 직전 노드를 표시. 5단계에서 비용[D]가 5원으로 갱신된 이유는 A-E + E-D = 4 + 1 = 5 이었기 때문. D 직전에 E를 거쳐 온 것이므로 도표의 해당 엔트리에 E를 표시하여 역추적

단계 수	선택정점	비용 [B]	비용 [C]	비용 [D]	비용 [E]	비용 [F]
1	B	1	5	∞	∞	2
2	F		3 B	∞	∞	2
3	C		3 B	∞	∞	
4	E			∞	4 C	
5	D			5 E		

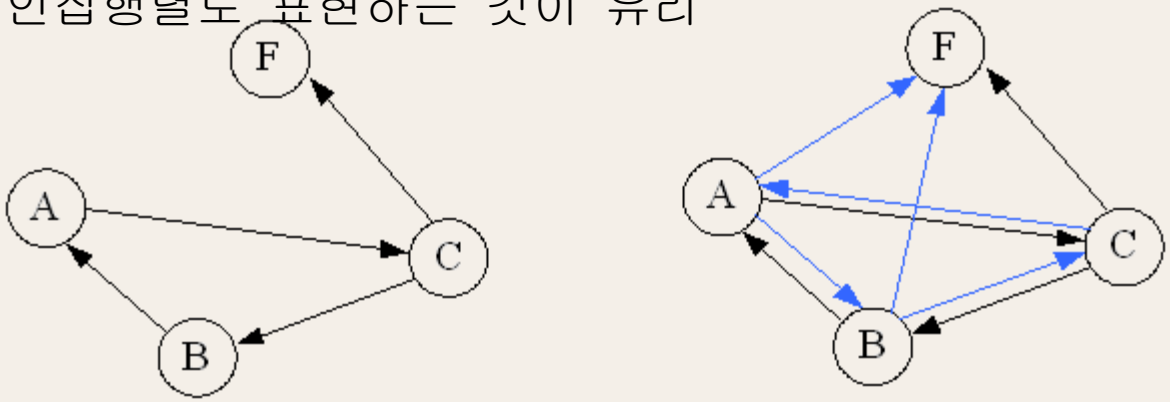
다이익스트라 알고리즘

- 도표의 행은 해당단계에서 최소비용
  - 3단계에서는 D까지 가는 비용은 현재까지 계산된 바로는 무한대 가장 작은 비용
  - 단계가 거듭될 수록 점차 최적화
- 동적 프로그램 기법(Dynamic Programming)
  - 문제가 너무 커서 해결할 수 없을 때
  - 간단한 사실 하나를 알아내는 데 주력
  - 알려진 사실을 바탕으로 해서 하나씩 새로운 사실을 추가하여 알려진 사실의 범위를 점차로 확대

이행폐쇄

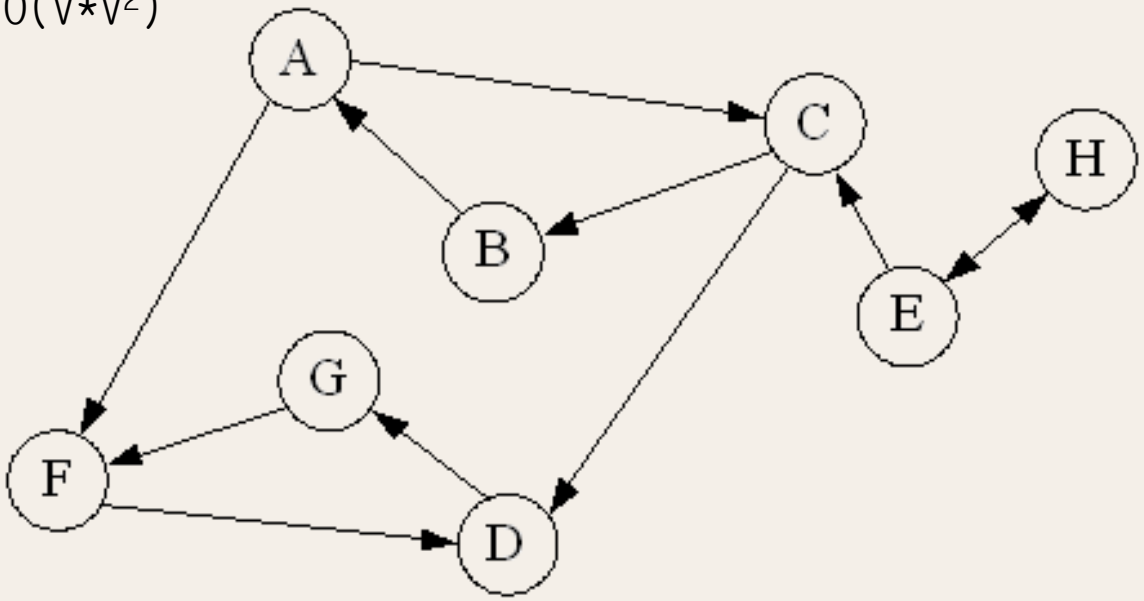
- 방향 그래프의 정점 X에서 정점 Y로 가는 길이 있는가

  - Y가 X에 연결되어 있는가 하는 연결성의 문제
  - 가는 길이 있다면 정점 X에서 정점 Y로 직접 가는 간선을 추가
  - 이후 질문에 즉답이 가능
  - 이행 폐쇄(Transitive Closure)
    - 거쳐서 갈 수 있는 모든 곳을 직접 가는 간선으로 연결한 그래프
    - 간선의 추가에 의해서, 대부분의 노드들이 인접노드로 바뀜.
    - 인접행렬로 표현하는 것이 유리

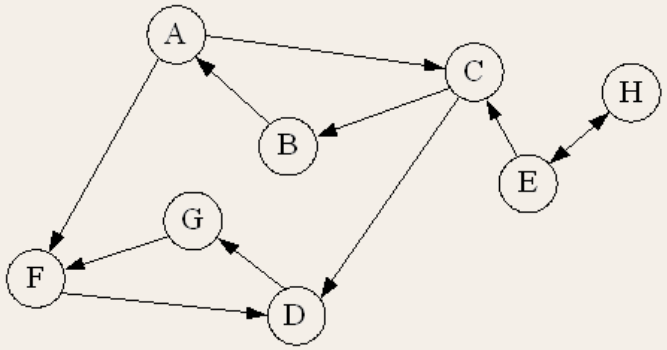


이행폐쇄

- 깊이우선 탐색
  - A로부터 시작하는 ACBDGF 라는 경로
  - A와 CBDGF가 연결되어 있음을 의미.
  - H에서 출발하면 A로 갈 수 있으나 이 연결 정보는 위 경로에 포함 안 됨.
  - 모든 노드에서 출발하는 깊이우선 탐색을 별도로 실행
  - 모든 정점에서 출발하는 깊이우선 탐색
    - 인접 리스트로 구현하면 O(V*(V+E)), 인접행렬로 구현하면 $O(V*V^2)$



와샬 알고리즘



	A	B	C	D	E	F	G	H
A	0	0	1	0	0	1	0	0
B	1	0	0→1	0	0	0→1	0	0
C	0	1	0	0	0	0→1	0	0
D	0	0	0	0	0	0	1	0
E	0	0	1	0	0	0	0	1
F	0	0	0	1	0	0	0	0
G	0	0	0	0	0	1	0	0
H	0	0	0	0	1	0	0	0

와샬 알고리즘

- 왼쪽에서 오른쪽, 그리고 위에서 아래 순서로
  - A[B][A] = 1은 B에서 A가는 길이 있다는 의미
  - 이는 A에서 갈 수 있는 모든 곳을 B에서 갈 수 있다는 의미. 왜냐하면 B에서 A를 거쳐 가면 되기 때문. A행을 보면 현재 C, F가 A에서 갈 수 있는 곳. 따라서 표의 A[B][C], A[B][F] 값을 1로 바꿈.
  - 스캔 할 때마다 0이 아닌 엔트리에 주목하되, 이는 이전에 0이었다가 1로 변한 것도 포함. 다시 말해 알고리즘 실행 결과는 누적됨.
  - 효율 면에서 와샬 알고리즘은 $O(V^3)$. 인접행렬의 모든 엔트리에 대해서 스캔을 가하여야 하므로 V^2 . 각각의 엔트리에 대해서 하나의 행을 탐색하기 위해 V 시간이 소요. B에서 C로 갈 수 있으면 C에서 갈 수 있는 모든 곳을 찾기 위해 C 행을 모두 탐색

플로이드 알고리즘

- 플로이드 알고리즘(Floyd's Algorithm)
  - 모든 정점으로부터 다른 모든 곳으로 가는 최단경로. 모든 쌍의 최단경로
  - 다이익스트라: 주어진 정점으로부터 다른 모든 곳으로 가는 최단경로
  - 와샬 알고리즘의 변형

플로이드 알고리즘

- A[A][B] = 1
  - A에서 B로 갈 수 있으므로 B에서 갈 수 있는 모든 곳을 A에서 갈 수 있음
  - B에서 갈 수 있는 모든 곳은 도표의 B 행에 표시
  - 비용 2에 C로 갈 수 있음. 따라서 A-B + B-C = 1 + 2 = 3. 이 비용은 현재 A에서 C로 가는 값 A[A][C] 인 5보다 작으므로 변경.

- A[D][B] = 1
  - D에서 B를 거쳐 C로 가는 D-B-C의 비용은 D-B + B-C = 1 + 2 = 3
  - 이 값은 현재 도표의 A[D][C] 인 무한대보다 작으므로 값을 3으로 변경
  - 도표에는 항상 지금까지 알아낸 최소값이 기록됨.

	A	B	C	D	E	F
A	0	1	5→3	∞	∞→4	2
B	∞	0	2	∞	∞→3	∞
C	∞	∞	0	∞	1	∞
D	∞	1	∞→3	0	∞	∞
E	∞	∞	∞	1	0	∞
F	∞	∞	∞	∞	∞	0

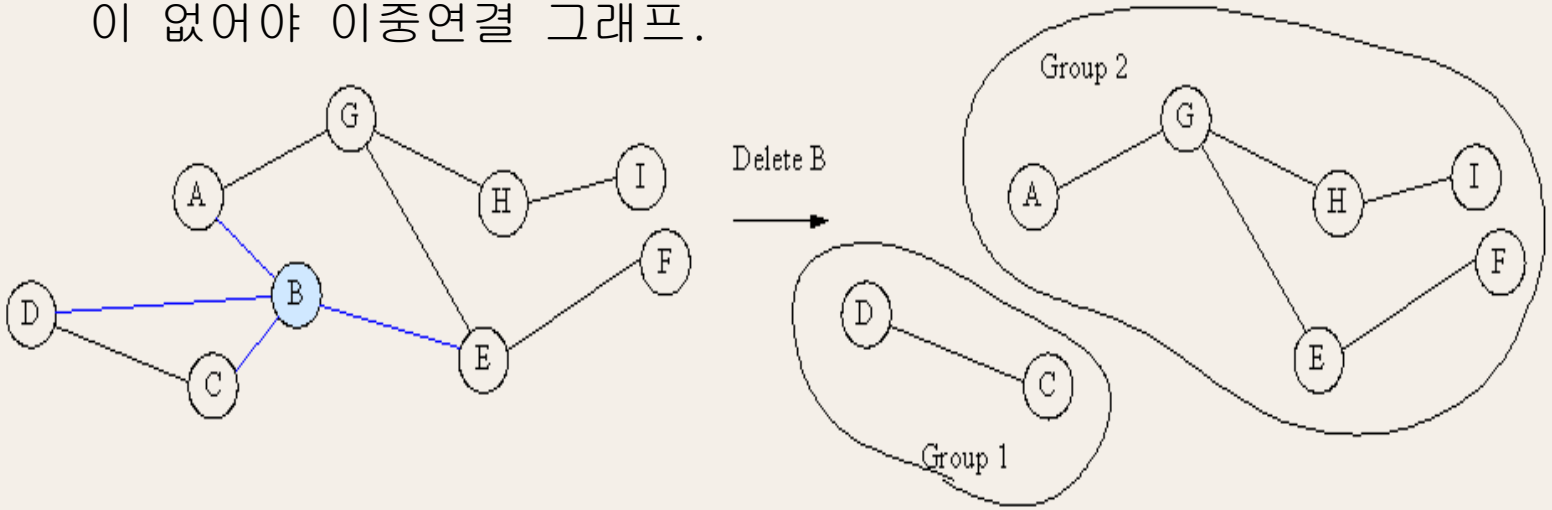
이중연결 그래프

이중연결 그래프(Biconnected Graph)

- 서울-경주-포항 이라는 철로. 경주 역에 장애가 발생. 서울에서 다른 곳을 거쳐서 포항을 갈 수 있도록 설계해야 함.
- 모든 정점 쌍을 연결하는 경로가 두 개 이상인 그래프.

관절점(AP: Articulation Point)

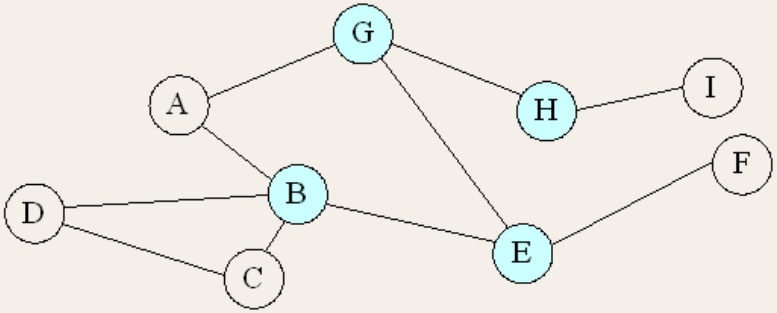
- 만약 그것이 사라지면 그래프가 두 개 이상의 그룹으로 분리되는 그러한 정점. 만약 이런 정점이 존재한다면 한 그룹에서 다른 그룹으로 건너가려면 반드시 그 정점을 통과해야 함. 관절점이 없어야 이중연결 그래프.

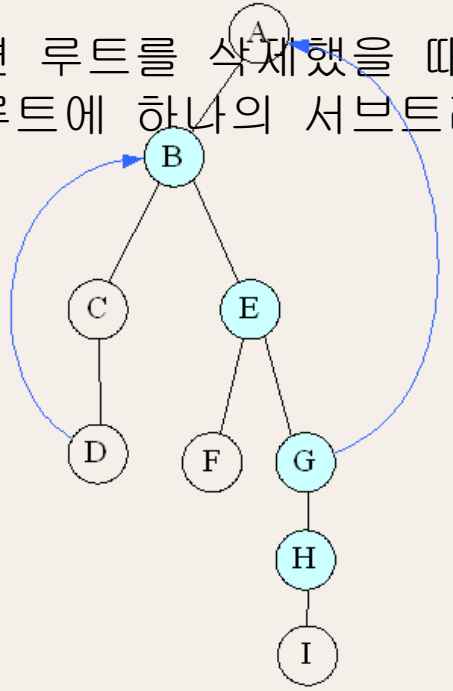


이중연결 그래프

❑ 깊이우선 탐색트리에 의한 관절점 판단

- B가 지워지면 두 개의 서브트리로 분리. 오른쪽 서브트리의 자식노드 중 G가 B의 상위노드를 가리킴. 왼쪽 서브트리는 어떤 자식노드도 B의 상위노드를 가리키는 것이 없음. B의 삭제는 왼쪽 서브트리 그룹의 분리로 이어지고, 결과적으로 B는 관절점
- C가 없어져도 자식인 D가 C의 상위노드를 가리키므로 C는 관절점이 아님. 서브트리가 하나일 경우에는 그 서브트리에 단 하나라도 삭제된 노드의 상위노드를 향하는 연결이 있으면 그 정점은 관절점이 아님.
- 루트가 2개 이상의 서브트리로 나뉜다면 루트를 삭제했을 때 서브트리가 분리되므로 루트가 관절점. 루트에 하나의 서브트리만 있다면 그 루트는 관절점이 아님.

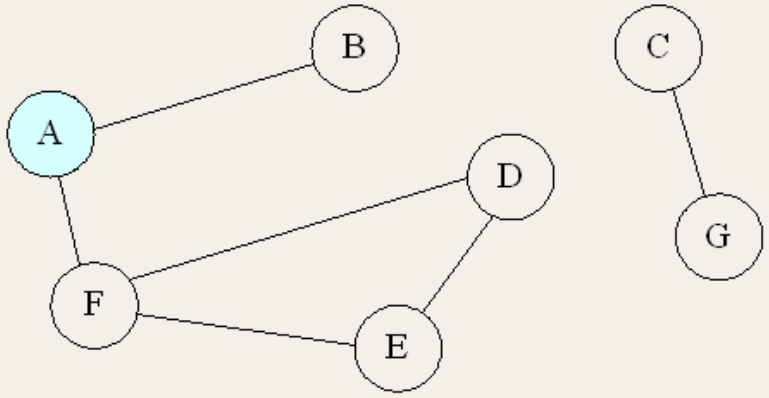




집합의 그래프 표현

집합의 그래프 표현

- 집합의 원소는 정점으로 나타냄
- 같은 집합에 속한다는 사실은 정점 사이의 연결로 나타냄
- 집합 {A, B, F, D, E}와 집합 {C, G}.
- 원소끼리 연결만 되어 있다면 동일한 집합을 나타냄



1	BFDE
2	AFDE
3	G
4	EFAB

...

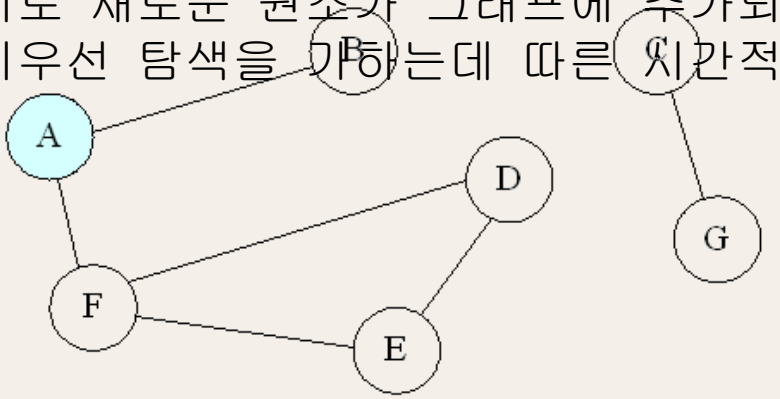
유니언 파인드

❑ 유니언 파인드

- 주어진 원소들이 서로 같은 집합에 속하는가를 알아내고(Find)
- 만약에 그렇지 않으면 같은 집합에 속하도록 추가하라(Union)

❑ 깊이우선 탐색

- 배열 첫 요소는 노드 A에서 출발한 것으로서 BFDE가 탐색결과
- D가 A와 같은 집합에 속하는가라고 질문하면 이 요소를 검색하여 답함.
- 이 방법은 집합의 원소가 고정된, 이른바 정적 상황에 알맞은 방법
- 수시로 새로운 원소가 그래프에 추가되는 동적상황에서는 매번 깊이우선 탐색을 가하는데 따른 시간적 비효율이 발생

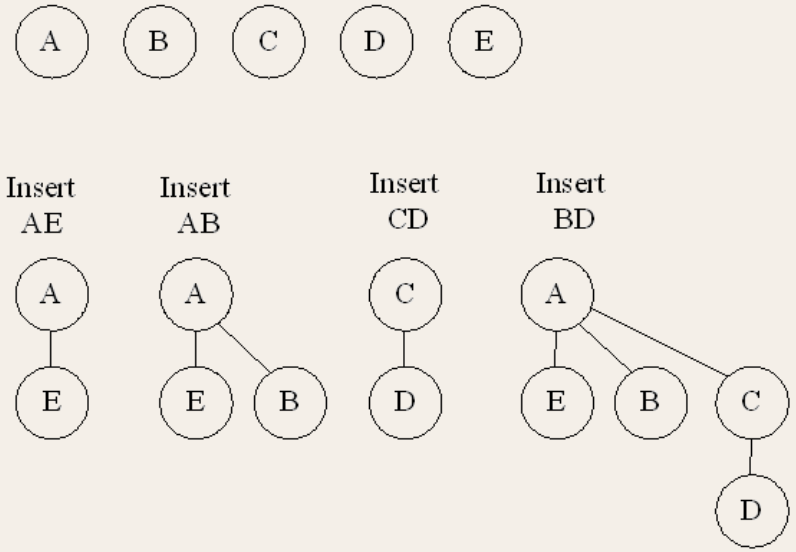


1	BFDE
2	AFDE
3	G
4	EFAB

...

유니언 파인드

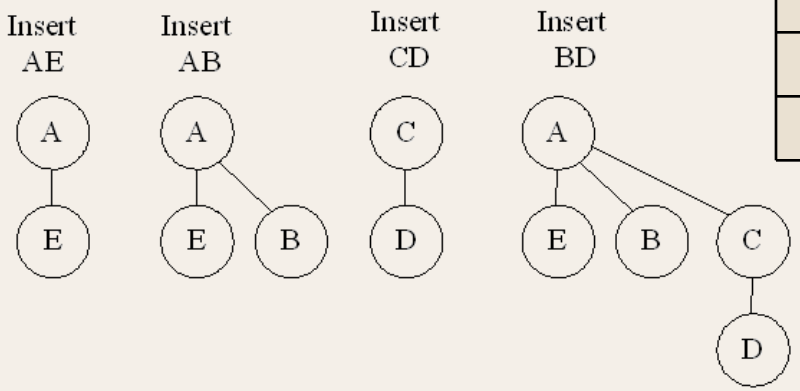
- 유니언 파인드 알고리즘
  - 동적 상황에 유리
  - 집합 A에 E를 추가하는 것은 간선 A-E를 연결함을 의미
  - A-E 삽입명령이 들어오면 A를 루트로 그 아래 E를 연결
  - A-B 삽입명령의 결과 A는 두개의 자식노드를 거느림
  - B-D 삽입은 직접 B에다 D를 연결하지 않고 B의 루트아래에 D의 루트를 연결



유니언 파인드

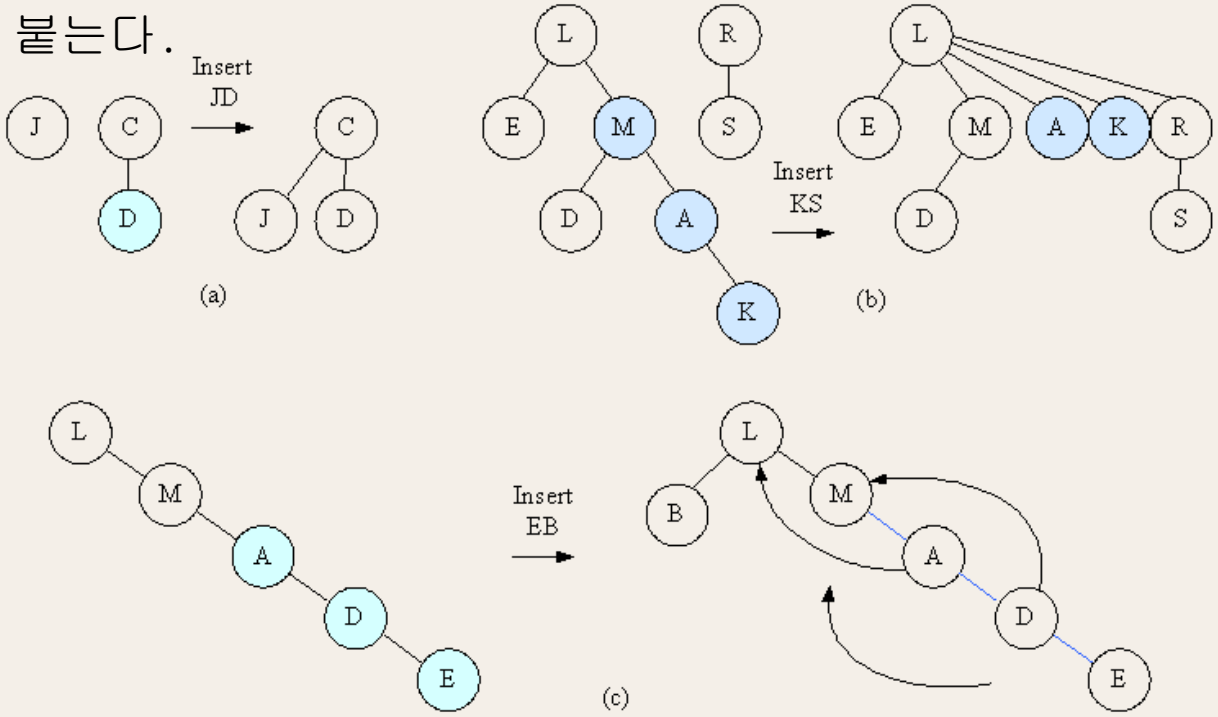
- 자료구조로 배열을 사용
  - A-E가 삽입되면 표의 * 표시한 곳에 기록. E의 루트가 A라는 의미
  - B-D의 삽입은 D의 루트를 B의 루트에 삽입. D의 루트는 C이고, B의 루트는 A이므로 C를 A에 삽입. C의 루트가 A가 되어야 하므로 C 칼럼에 A가 기록.
  - 노드의 루트를 발견하기 위해서는 while (A[i] ≠ Empty) i = A[i];
- M과 N이 같은 집합인가
  - M의 루트와 N의 루트가 일치하면 삽입

	A	B	C	D	E
A-E 삽입					*A
A-B 삽입		A			
C-D 삽입				C	
B-D 삽입			**A		



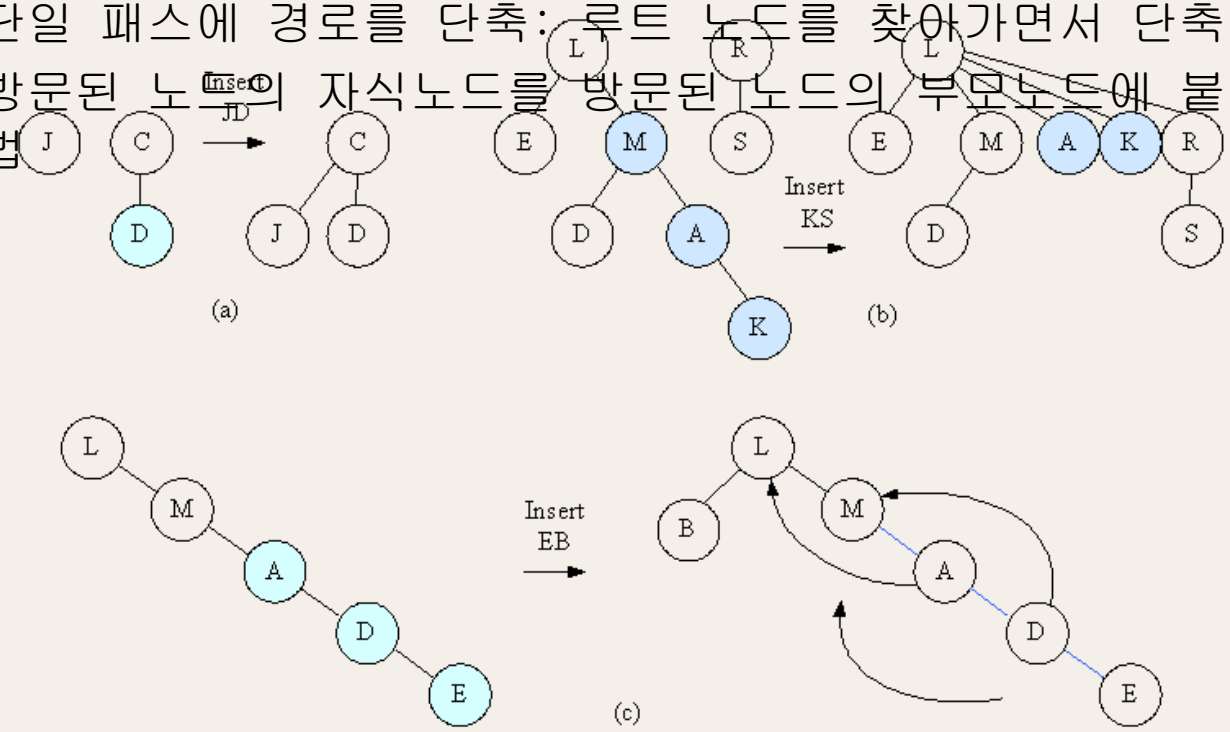
유니언 파인드

- **효율**
  - D-E 삽입, C-E 삽입, B-E 삽입
  - E의 루트를 찾기 위한 반복문 실행에 많은 시간. 최악의 경우 표의 모든 엔트리를 참조. 효율은 O(V).
- **가중치 유니언 (Union by Weighting)**
  - 자식노드가 더 많은 루트를 최종 루트로 하는 방법
  - (a)의 J-D를 추가하는 작업에서 J의 루트인 J가 D의 루트인 C에 붙는다.



유니언 파인드

- 경로압축(Path Compression)
  - 루트를 찾는 과정에서 나타나는 모든 노드를 직접 최종 루트 아래로. K의 루트인 L을 찾는 과정에서 만나는 것은 K, A, M
  - 2 단계의 작업: 루트를 찾으면서 나타나는 모든 노드에 표시. 나중에 루트를 찾으면 그 노드를 모두 찾아가서 부모노드를 변경.
- 분할(Halving)
  - 단일 패스에 경로를 단축: 루트 노드를 찾아가면서 단축작업
  - 방문된 노드의 자식노드를 방문된 노드의 부모노드에 붙이는 방법



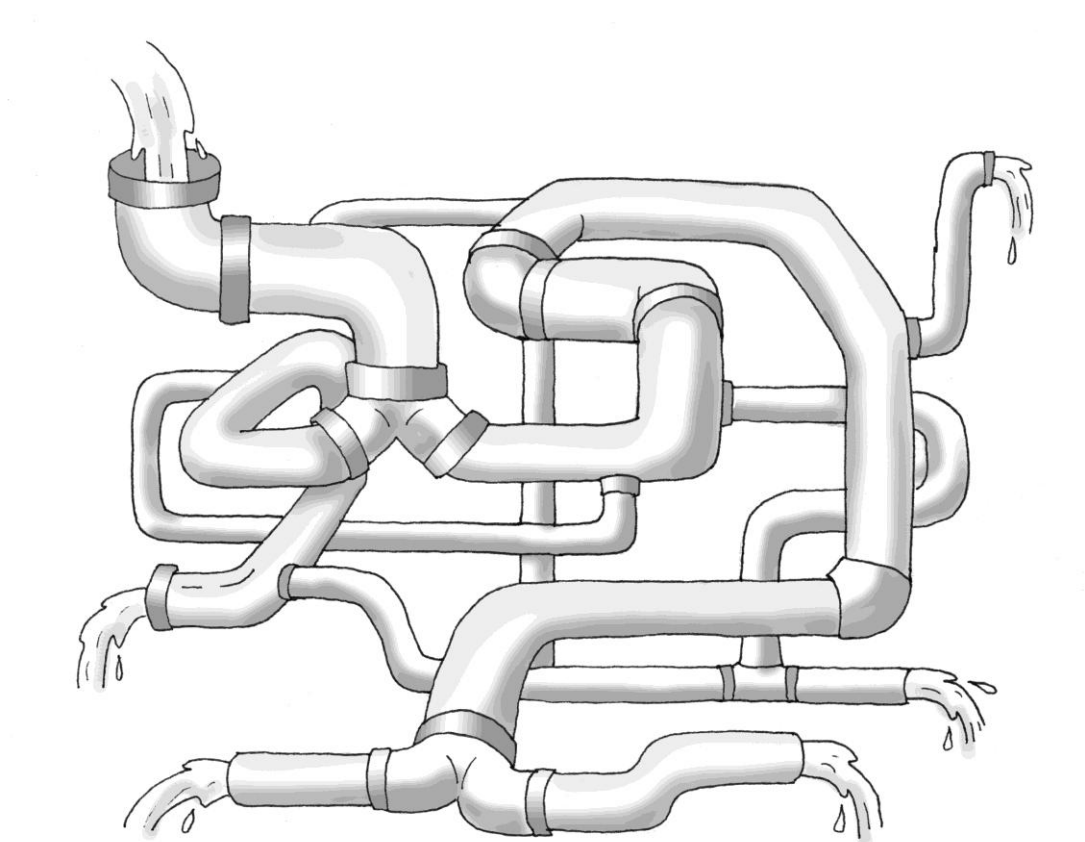
네트워크 플로우

- 방향성 그래프 G의 가중치에 유량(流量, Flow)을 할당
  - 유량을 최대화하는 문제
  - 상하수도 배관이 수용하는 범위 내에서 가능한 많은 유량
  - 통신 선로 대역폭이 수용할 수 있는 범위 내에서 가능한 많은 정보
  - 도로가 수용할 수 있는 범위 내에서 가능한 많은 교통량

- 네트워크 플로우(Network Flow)
  - 어떤 정점에서 다른 정점으로 갈 수 있는 흐름을 최대화
  - 어떤 간선이 수용할 수 있는 용량이 초과되면 역류가 발생
  - 역류를 일으키지 않으면서 유량을 최대화하라는 문제

네트워크 플로우

- 네트워크 플로우



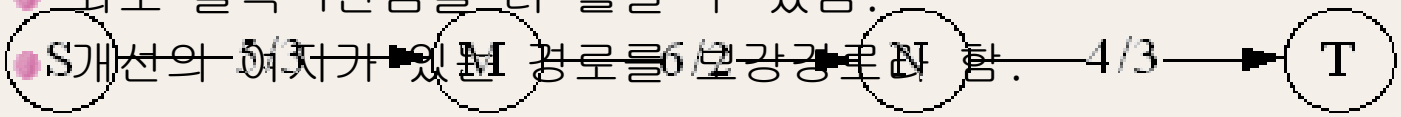
네트워크 플로우

❑ 네트워크 플로우

- 노드 S는 소스, 노드 T는 싱크를 나타냄.
- 가중치 5/3은 최대용량(Capacity) 5에 현재유량(Flow) 3을 흘림을 의미
- 개념적 간선 S->M->N->T를 잡으면 이 방향은 실제 간선의 방향과 일치. 이를 순방향 간선(Forward Edge)이라 부름.
- 최대용량에서 현재유량을 뺀 차이를 슬랙(Slack, 느슨한 정도)이라 함.
- S-M 사이의 슬랙은 5 - 3 = 2

❑ 보강경로(Augmenting Path, Alternating Path)

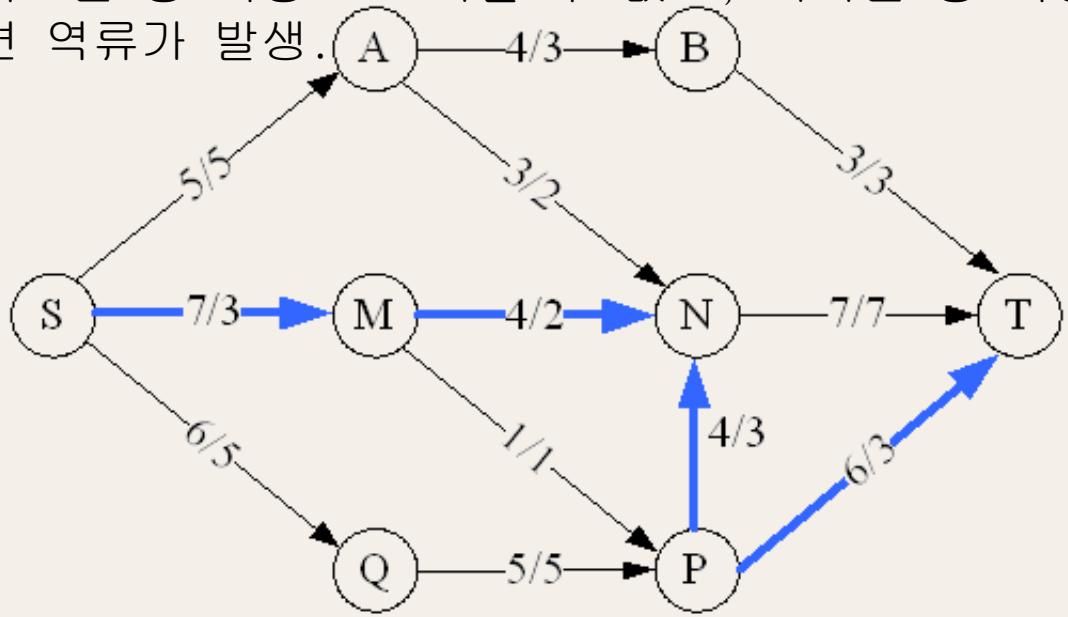
- 각 노드 사이의 슬랙은 2, 4, 1
- 최소 슬랙 1만큼을 더 흘릴 수 있음.
- S개선의 여지가 있는 경로를 보강경로라 함.



네트워크 플로우

❑ 순방향 흐름 분석

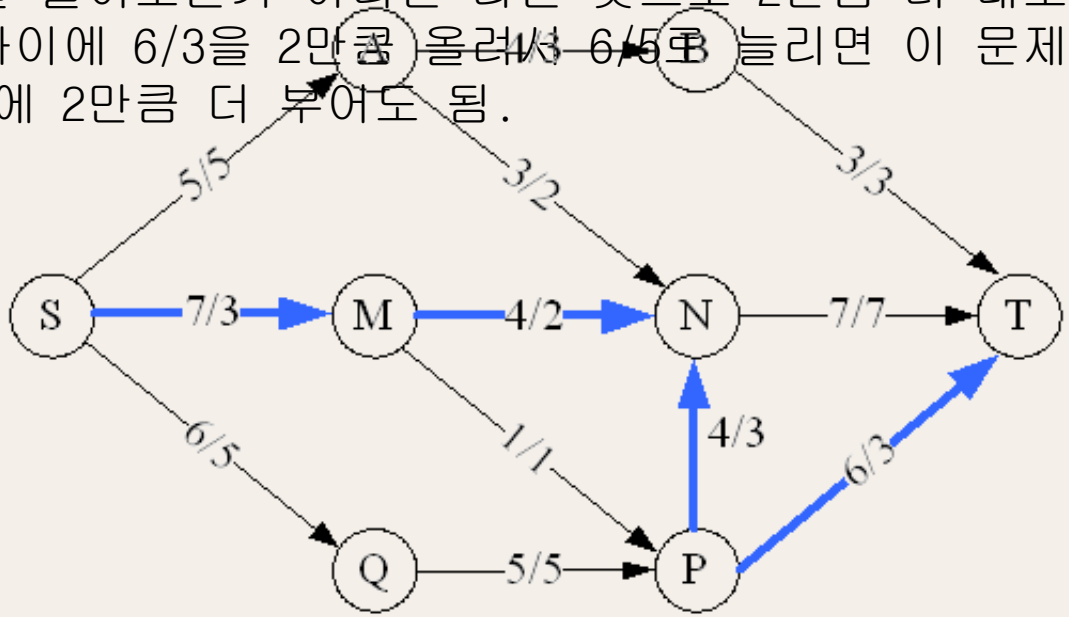
- S-M-N-T의 순방향 흐름은 모두 최대
- N-T 사이가 최대용량으로서 슬랙이 0
- S-Q-P-T 역시 최소 슬랙 0
- 소스에서 나가는 양은 5 + 3 + 5 = 13, 싱크로 들어오는 양은 3 + 7 + 3 = 13
- 어떤 노드로 들어오는 유량과 나가는 유량의 합은 같아야 한다. 들어오는 양 이상으로 나갈 수 없고, 나가는 양 이상으로 들어오면 역류가 발생.



네트워크 플로우

❑ 역방향 간선(Backward Edge)

- 개념적인 간선의 방향을 S-M-N-P-T로. N-P 사이의 개념적 흐름은 실제 흐름과 반대.
- S-M-N 사이의 최소 슬랙은 2. 소스에 2를 더 부르면 S-M이 7/5로, M-N이 4/4로 증가. 그런데 이렇게 되면 노드 N이 문제. 2만큼 더 들어오니 그만큼 더 나가거나, 아니면 다른 곳에서 2만큼 덜 들어오면 됨. P-N 사이를 보자. N으로 들어오는 양을 4/1로 줄이면 2만큼 덜 들어오니 N으로서는 무리가 없음. 이제 N의 제약이 P의 제약으로 바뀜. P로서는 2만큼 덜 내보내야 하니 그만큼 덜 들어오든가 아니면 다른 곳으로 2만큼 더 내보내면 됨. P-T사이에 6/3을 2만큼 올려서 6/5로 늘리면 이 문제는 해결됨. 소스에 2만큼 더 부어도 됨.

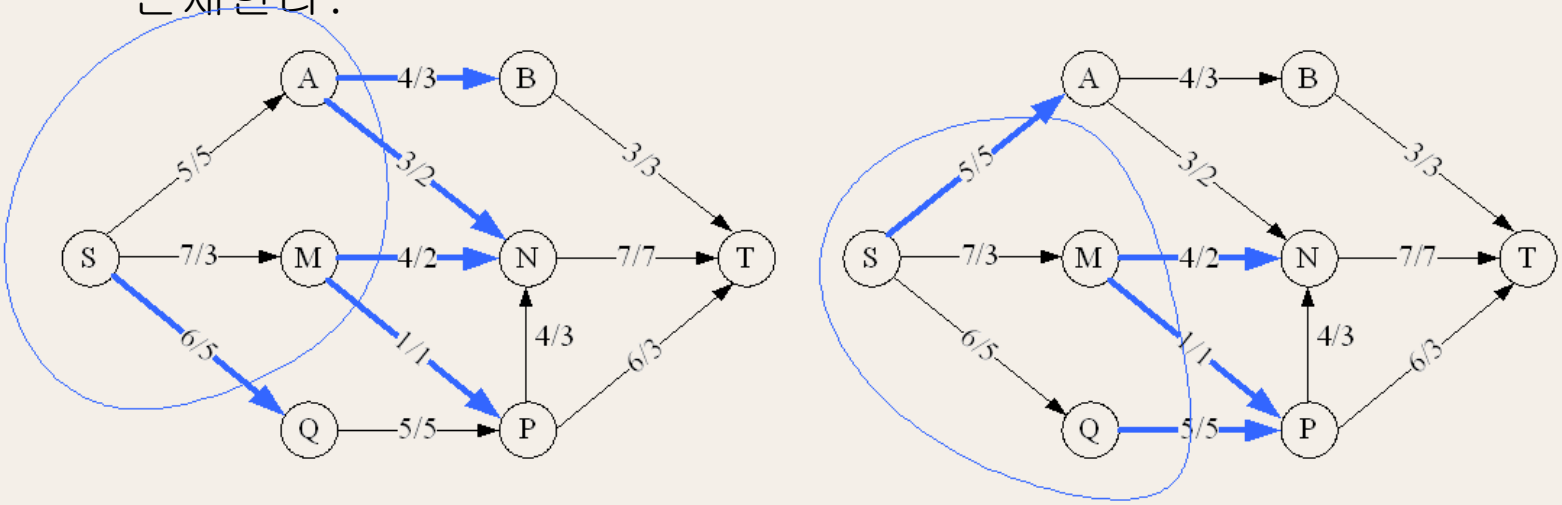


보강경로 이론

- “만약에 그래프 내에 어떠한 보강경로도 없다면 그 흐름은 최대이다.”
- 어떤 그래프에서 더 이상의 보강경로가 없는지를 어떻게 증명하는가?

커트

- 커트(Cut)는 소스와 싱크를 분리하는 간선의 집합
  - 첫째 그림은 S, A, M을 소스 그룹으로, 나머지 정점을 싱크 그룹으로 분리. 소스와 싱크를 분리하는 간선은 A-B, A-N, M-N, M-P, S-Q 등 다섯 개가 존재
  - 둘째 그림은 S, M, Q를 소스 그룹으로, 나머지를 싱크 그룹으로 분리. 그룹을 분리하는 간선은 S-A, M-N, M-P, Q-P 등 세 개가 존재한다. 자르는 방법에 따라서 그래프 내에는 수많은 커트가 존재한다.



맥스 플로우 민 커트 이론

- “최대 유량은 최소 커트와 같다 “
  - 각 그룹을 하나의 노드로 바라보면 그래프는 두 개의 노드와 그 노드를 잇는 커다란 배관. 두 노드를 잇는 배관의 크기는 각 커트의 최대용량(Capacity)의 합과 같음. 여러 가지 방법으로 커트를 만들 수 있으나 어떻게 자르던 최대용량은 커트를 초과할 수 없음.

- 커트 용량
  - 첫째 커트의 최대용량의 합은 (4 + 3 + 4 + 1 + 6) = 18. 둘째 커트는 (5 + 4 + 1 + 5) = 15. 이 두 가지를 종합하면 최대 유량은 일단 15를 초과할 수 없음.
  - 다른 커트 방법을 찾아해야 함. 최소 커트는 병목현상에 있어서 병목에 해당