Graph#1

2020.08.14

CONTENTS

- Graph

0. [C] Pointer & Struct
1. Search
   - ✓ DFS
   - ✓ BFS
2. Shortest Path Algorithm
   - ✓ Dijkstra
   - ✓ Bellman-Ford
   - ✓ Floyd-Warshall
   - ✓ SPFA(Shortest Path Faster Algorithm)
3. Graph Modeling
4. Topological Sort

#Pointer & Struct

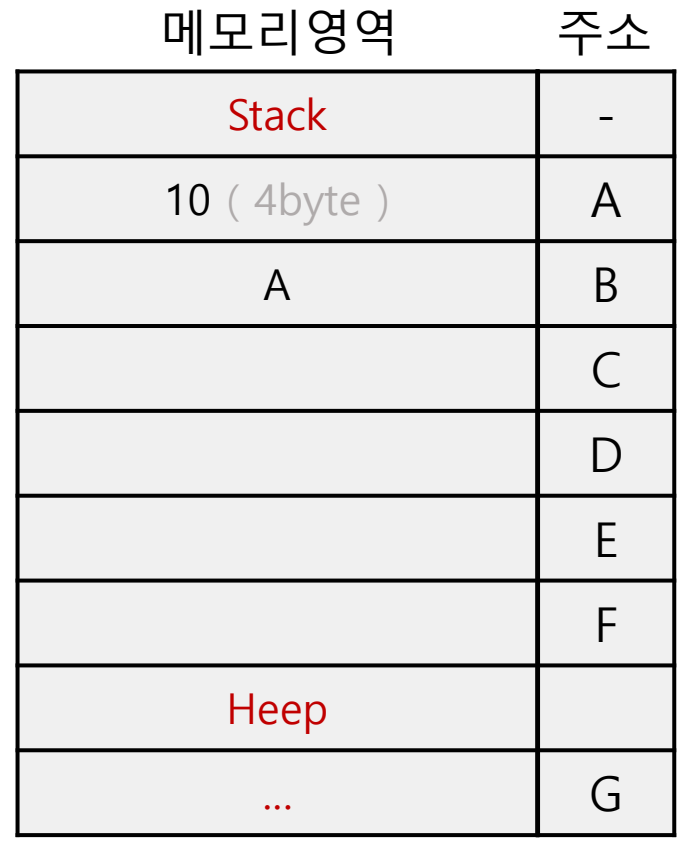
✓ 포인터 또는 포인터 변수란 데이터의 메모리 주소를 저장하는 변수.

포인터를 활용해 다른 지역의 해당 데이터를 빠르게 접근할 수 있다.

데이터형에 *(star)를 붙여서 포인터 변수를 선언한다. [*을 "가리키는 곳"라고 보자]

[Main.c]

```
int num = 10; // num에 메모리 할당
int* p_num = num; //num의 주소를 저장
printf("주소 : %p , 값 : %d", p_num, *p_num);
// 주소 : A , 값 : 10



*p_num = 50;
//역 참조 : 메모리 공간에 있는 데이터를 수정하는 것
printf("주소 : %p , 값 : %d", p_num, *p_num);
// 주소 : A , 값 : 50
printf("num : %d", num);
// num : 50
```



Stack : 변수가 지역을 벗어날 시 사라지는 메모리 영역

Heep : 프로그램 실행동안 사라지지 않고 수동으로 해제해줘야 하는 메모리 영역

메모리 누수주의

#Pointer & Struct

왜 포인터를 사용하는가?

1. 빠른 데이터 접근
   - 주소를 이용할 시 메모리에 있는 변수의 주소를 통해 저장공간에 바로 접근할 수 있다. 포인터를 사용하지 않는 다면 해당 데이터의 주소를 메모리 영역에서 찾은 다음 데이터를 수정한다. 이때, 일치하는 주소를 탐색하는 시간이 발생한다.

2. Call by Reference
   - 함수의 매개변수 전달방식은 값에 의한 호출, 참조에 의한 호출이 있다.
   - 값에 의한 호출은 함수가 호출될 때 인자에 저장된 데이터가 복사되어 함수 매개변수에 전달된다. 복사된 데이터는 함수 안의 새로운 지역 변수가 된다. 데이터 변경을 시도할 시 지역 내에서만 이루어지고 지역 변수이므로 함수가 종료되면 사라진다.

   - 참조에 의한 호출은 인자의 주소를 전달받는다. 주소를 전달받았기 때문에 전달받은 주소의 데이터가 변경될 시 인자의 데이터가 변경되고 함수가 종료되어도 사라지지 않는다.

[Main.c]

```
int a=10,b=20;
add_value(a,b);
add_reference(&a,&b);
// [&]엠퍼센트는 주소를 반환하는 기호
```

```
void add_value(int a, int b)
{
        a= a+5;
        b= b+5;
}
// 함수 a, b는 이름만 같지 다른 메모리 주소를 가진 변수, 지역성을 가지므로 삭제됨
```

```
void add_reference(int* a, int* b)
{
        *a= *a+5;
        *b= *b+5;
}
// 주소를 받았기 때문에 a=15,b=25가 됨
```

#Pointer & Struct

구조체란 여러 변수들을 저장하는 문법
여러 변수들을 묶어 한꺼번에 선언할 수 있다.

```
string name1;
int age1;
int Phone_Number1;

string name2;
int age2;
int Phone_Number2;

string name3;
int age3;
int Phone_Number3;
        ....
```

```
struct Student
{
        string name;
        int age;
        int Phone_Number;
};
```

```
Student student[20];
```

20개의 Student 구조체 한꺼번에 생성

Graph

[정의] 정점들의 집합 V, 간선들의 집합 E가 있고 그래프가 G라고 할 때, G = (V , E)이다.

- 데이터와 데이터의 관계, 순서를 표현할 때 유용한 자료구조

[그래프의 구성]

- 정점 (vertex) : 위치
- 간선 (edge) : 정점과 정점을 잇는 선

```
typedef struct tagVertex
{
   Type Data; // 정점에 저장할 데이터
   int Visited; // 탐색 시 정점의 방문여부
   int Index; // 그래프 내에 정점 순서번호

   struct *tagVertex Next; // 그래프 내 다음 정점
   struct *tagEdge Adj_List // 인접 정점의 간선 리스트
}Vertext; //구조체 형태
```

```
typedef struct tagGraph
{
   Vertex* Verticies; // 첫 정점 노드
   int VertextCount; // 정점의 갯수
}Graph; //구조체 형태
```

```
typedef struct tagEdge
{
   int weight; // 가중치
   struct tagEdge* Next; // 다음 간선[같은 정점 간선 리스트]
   *Vertex From; // 출발 정점
   *Vertex Target; // 도착 정점
}Edge; //구조체 형태
```

DFS (Deep-Frist Search)

그래프의 마지막 깊이까지 들어가서 탐색하는 방식

[알고리즘 과정]

1. 시작 정점을 선택하고 [방문함]을 표시한다.

2. 시작 정점과 이웃한 정점 중에서 방문하지 않은 곳을 선택해서 이 정점을 다시 1을 수행한다. 이때, 정점에 방문하지 않은 곳이 없을 때 이전 정점으로 돌아간다.

3. 이전 정점으로 돌아가도 더 이상 방문할 이웃이 없다면 모두 방문했다는 것으로 종료한다.

```
typedef struct tagVertex
{
    Type Data; // 정점에 저장할 데이터
    int Visited; // 탐색 시 정점의 방문여부
    int Index; // 그래프 내에 정점 순서번호

    struct *tagVertex Next; // 그래프 내 다음 정점
    struct *tagEdge Adj_List // 인접 정점의 간선 리스트
}Vertext; //구조체 형태
```

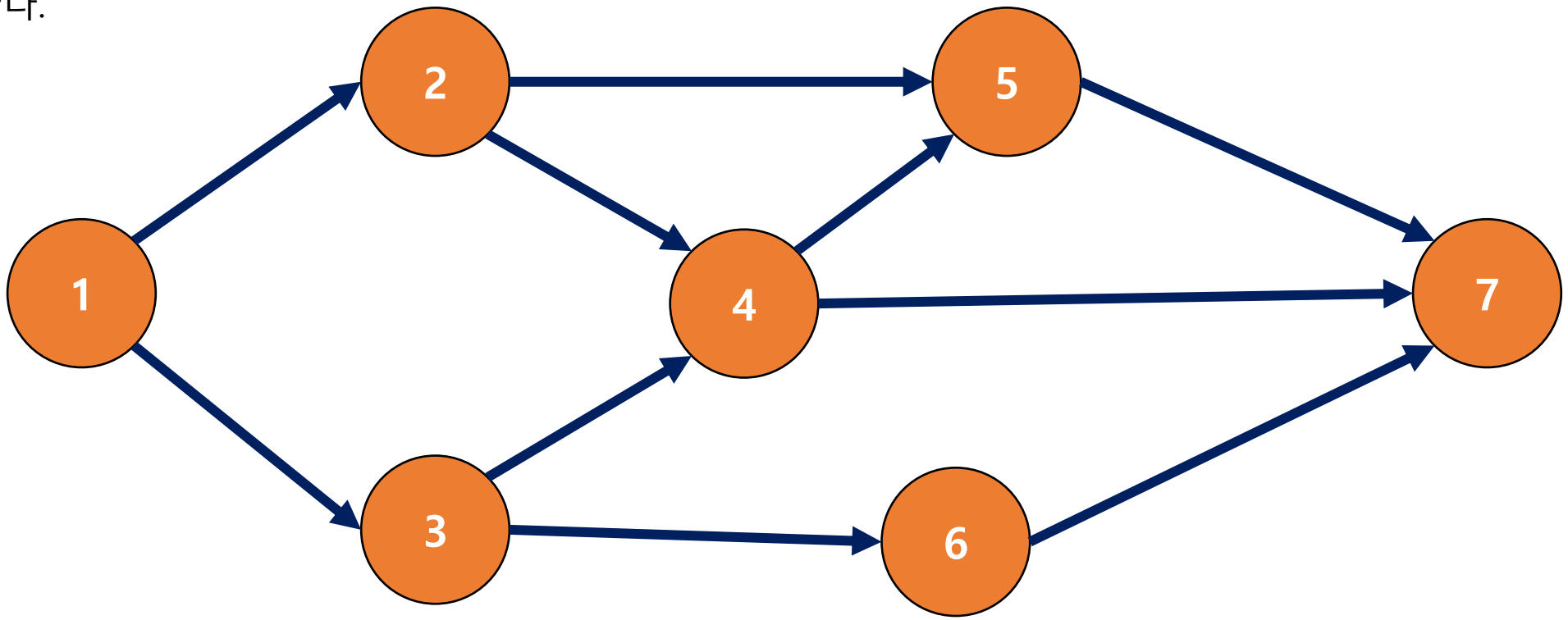
DFS (Deep-Frist Search)

[알고리즘 과정]

1. 시작 정점을 선택하고 [방문함]을 표시한다.

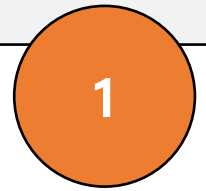
2. 시작 정점과 이웃한 정점 중에서 방문하지 않은 곳을 찾고 선택해서 이 정점을 다시 1과 같이 수행한다. 이때, 정점에 방문하지 않은 곳이 없을 때 이전 정점으로 돌아간다.

3. 이전 정점으로 돌아가도 더 이상 방문할 이웃이 없다면 모두 방문했다는 것으로 종료한다.



DFS (Deep-Frist Search)

```
void DFS( Vertex* V ) // 노드를 전달 받음( ex. 1 노드 )
{
  Edge* E=NULL;                    // 초기화

  V->Visited = Visited;            // 방문표시

  E = V->Adj_List; // 전달받은 노드의 인접리스트를 E에 저장

    while( E != NULL)
    {
        // 인접 정점이 존재여부와 방문한 정점이 아닐 시 다음 정점 탐색
        if( E->Target !=NULL && E->Target->Visited == NotVisited )
                DFS( E->Target );        // 재귀 호출

        // 인접 정점이 존재하지않고, 방문만 남았을 시 다른 간선으로 탐색
        E = E->Next;
    }
}
```




```
typedef struct tagVertex
{
   int Data = 1;
   int Visited = 1; // 0 , 1
   int Index = 1// 1

   struct *tagVertex Next; 1->2->3->4->5->6 [ 리스트 형태 ]
   struct *tagEdge Adj_List; //2, 3 [ 2의 Next가 3임 리스트형식 연
결 ]
} ex. No.1 Vertex
```

```
typedef struct tagVertex
{
   Type Data; // 정점에 저장할 데이터
   int Visited; // 탐색 시 정점의 방문여부
   int Index; // 그래프 내에 정점 순서번호

   struct *tagVertex Next; // 그래프 내 다음 정점
   struct *tagEdge Adj_List // 인접정점의 간선 리스트
}Vertext; //구조체 형태
```

```
typedef struct tagEdge
{
   int weight; // 가중치
   struct tagEdge* Next; // 다음 간선[같은 정점 간선 리스트]
   *Vertex From; // 출발 정점
   *Vertex Target; // 도착 정점
}Edge; //구조체 형태
```

BFS (Breadth-Frist Search)

그래프의 같은 깊이들의 노드들부터 탐색하는 방식

[알고리즘 과정]

0. 우선 큐를 만든다. (First-In First-out 자료구조)

1. 시작 정점을 방문표시하고 큐에 삽입한다.

2. 큐에서 Dequeue를 수행하고, 삭제했던 노드의 이웃한 인접 정점 중 방문을 하지않은 노드들을 방문표시를 하고 큐에 모두 삽입한다. 이를 반복한다.

3. 이때, 큐가 비게 되면 모두 방문한 것으로 종료하고 탐색을 마친다.

```
typedef struct tagVertex
{
    Type Data; // 정점에 저장할 데이터
    int Visited; // 탐색 시 정점의 방문여부
    int Index; // 그래프 내에 정점 순서번호

    struct *tagVertex Next; // 그래프 내 다음 정점
    struct *tagEdge Adj_List // 인접 정점의 간선 리스트
}Vertext; //구조체 형태
```

BFS (Breadth-Frist Search)

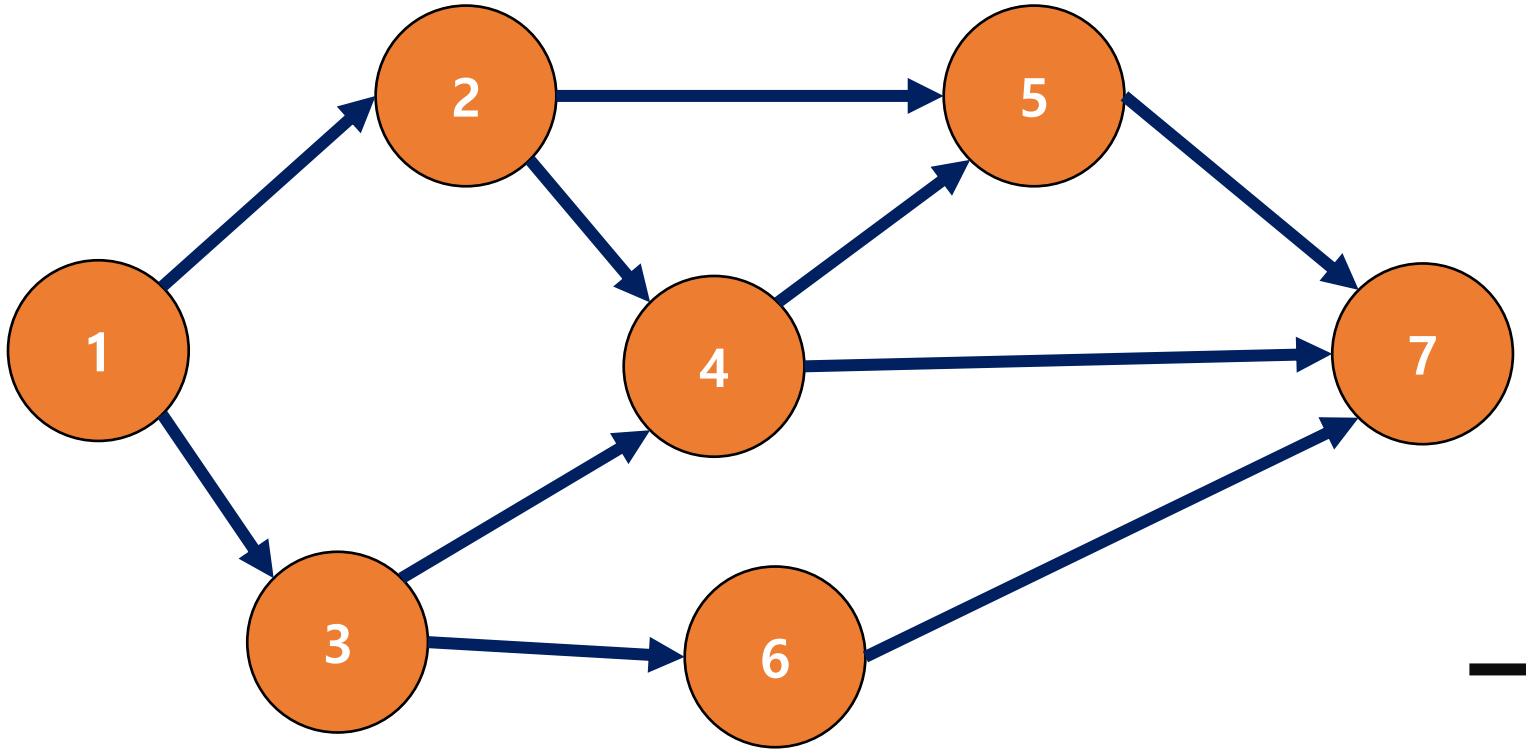
[알고리즘 과정]

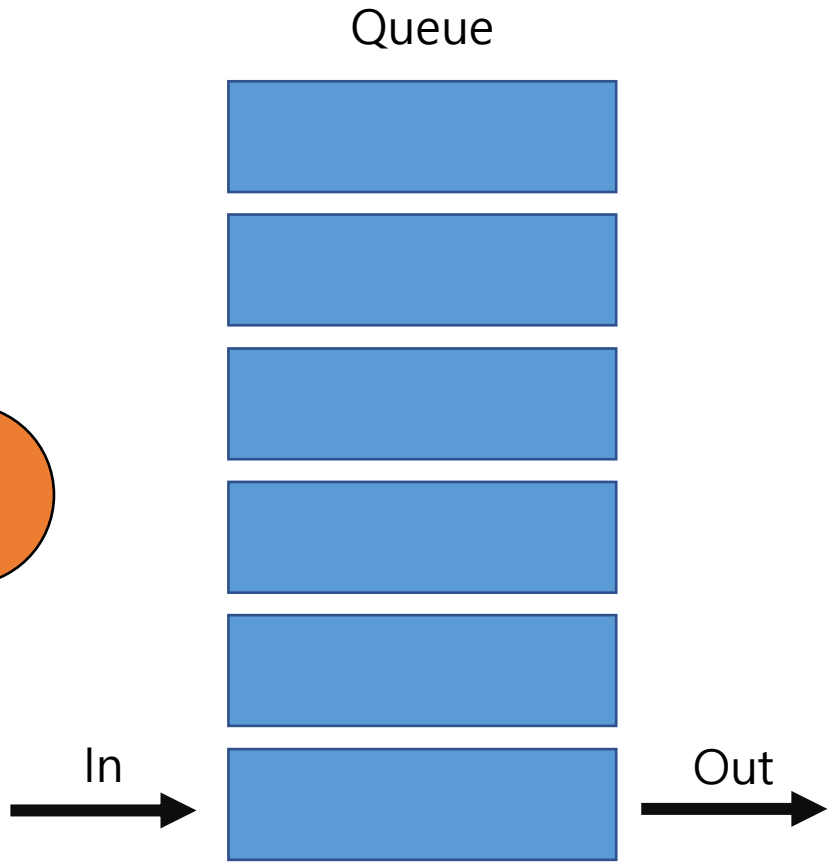
0. 우선 큐를 만든다. (First-In First-out 자료구조)

1. 시작 정점을 방문표시하고 큐에 삽입한다.

2. 큐에서 Dequeue를 수행하고, 삭제했던 노드의 이웃한 인접 정점 중 방문을 하지않은 노드들을 방문표시를 하고 큐에 모두 삽입한다. 이를 반복한다.

3. 이때, 큐가 비게 되면 모두 방문한 것으로 종료하고 탐색을 마친다.





BFS (Breadth-Frist Search)

```
void BFS( Vertex* V, Queue* Queue ) // 노드와 큐를 전달 받음)
{
  Edge* E=NULL; // 초기화

  V->Visited = Visited; // 방문표시
  Queue_Enqueue( &Queue, Q_CreateNode ( V ) ); // 큐에 정점을 큐의 노드로 추가하는 명령문

    while( !LQ_IsEmpty( Queue ))                  // 큐가 비어 있으면 false 탐색종료
    {
        Node* popped = Q_Dequeue( &Queue ); // 큐에 저장된 노드를 반환
        V = popped->Data;                         // 반환한 정점을 V에 저장
        E = V->Adj_List;                          // 정점의 인접 정점을 가리키는 첫 번째 간선 저장

              while( E !=NULL)                    // 간선이 없을 시(NULL) 종료
               {
                    V = E->Target;                 // 간선이 가리키는 정점을 저장
                    if( V != NULL && V->Visited == NotVisited )
                     {
                            V->Visited = Visited;
                            LQ_Enqueue( &Queue, Q_CreateNode ( V ) );
                     }
                    E = E -> Next;                 // 다음 간선을 저장
               }
    }
}
```

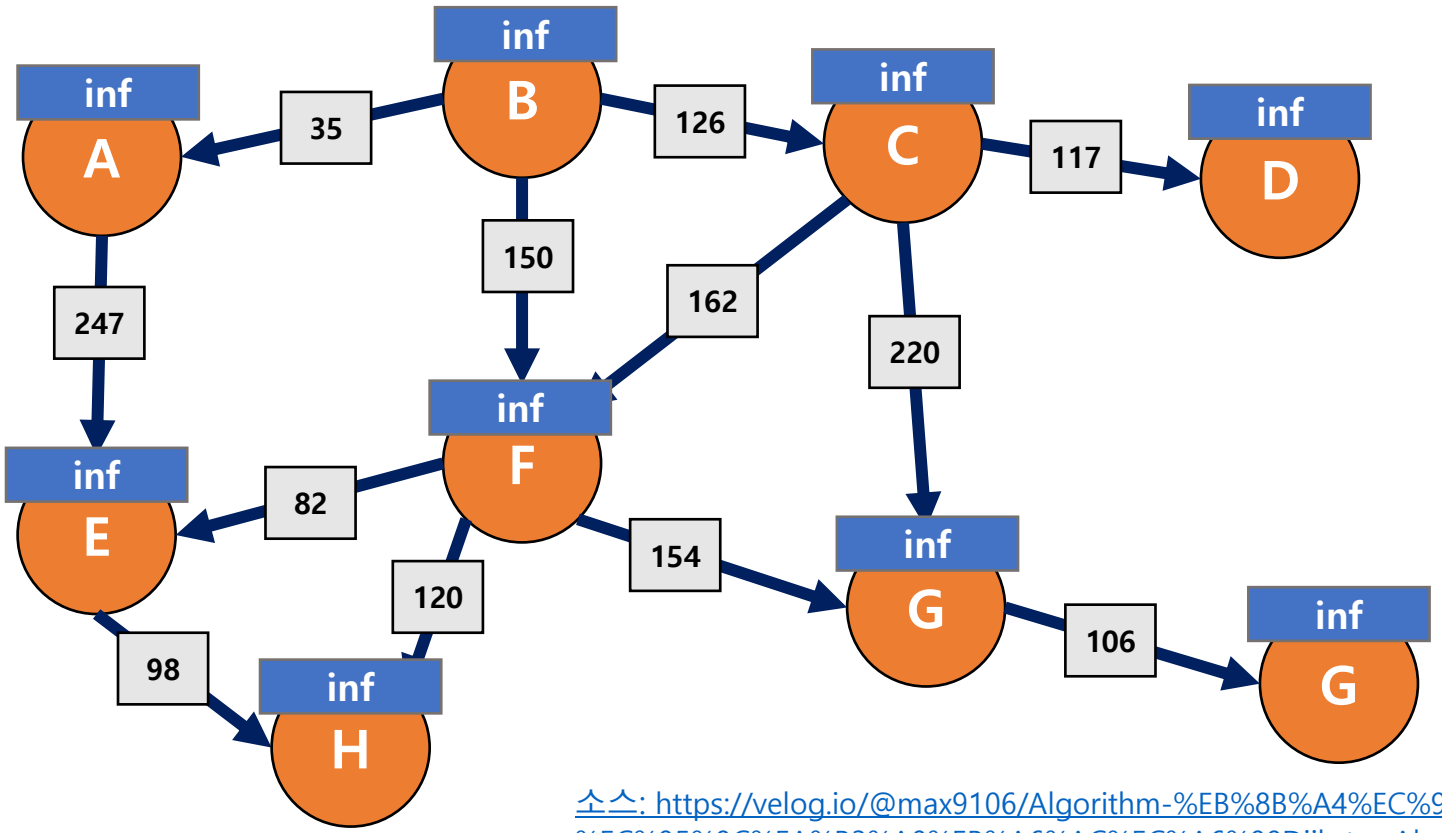
```
typedef struct tagVertex
{
    Type Data; // 정점에 저장할 데이터
    int Visited; // 탐색 시 정점의 방문여부
    int Index; // 그래프 내에 정점 순서번호

    struct *tagVertex Next; // 그래프 내 다음 정점
    struct *tagEdge Adj_List // 정점의 간선 리스트
}Vertext; //구조체 형태
```

Break-Time

Dijkstra

한 정점에서 모든 정점의 최단경로를 구하는 알고리즘

[알고리즘 조건]
사이클이 없는 방향성 그래프
Minimum Spanning Tree



[알고리즘 과정]
0. 시작지점으로 부터 목적지까지의 경로의 길이들을 저장할 [최단경로 자료구조]를 만들고 모든 공간에 무한으로 초기화 한다.

1. 시작 정점을 선택한 다음 그 정점의 경로 길이는 0으로 초기화하고 최단경로에 추가한다.[시작에서 시작은 0]

2. 최단경로에 추가된 정점의 인접 정점들을 통해 경로 길이를 갱신하고 최단경로에 추가한다. 갱신은 목적지까지의 가중치를 합한 것이다. 이때, 이미 최단 경로 안에 목적지가 존재한다면 새 경로와 이전 경로 중 가중치 합이 작은 것으로 경로를 수정한다.

3. 그래프 내의 모든 정점이 최단 경로에 소속될 때까지 2를 반복한다.

소스: https://velog.io/@max9106/Algorithm-%EB%8B%A4%EC%9D%B5%EC%8A%A4%ED%8A%B8%EB%9D%BC-%EC%95%8C%EA%B3%A0%EB%A6%AC%EC%A6%98Dijkstra-Algorithm [Junseo Kim]

Bellman-Ford

간선이 음의 가중치를 가질 때의 최단경로 알고리즘
#하나의 시작점에서 모든 도착점으로 가는 최단경로

[알고리즘 과정]
1. 우선 정점들의 인접행렬을 구한다.

2. 다음 모든 정점에 대해 반복하여 모든 경로를 구한다. 이때 초기화된 인접행렬과 비교를 하는데 [1에 인접행렬에 저장된 도착지까지의 가중치]와 [1의 인접행렬 가중치 + 해당 정점에서 도착지까지의 가중치]을 비교한다. 후자가 더 작으면 가중치 위치에 저장한다.

3. 성능 : O(VE)

소스: https://engkimbs.tistory.com/363 [새로비]

Floyd-Warshall

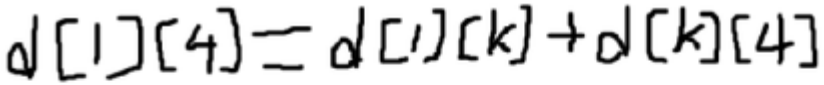
모든 정점에서 모든 정점으로 가는 최단 경로를 구하는 알고리즘

[알고리즘 과정]

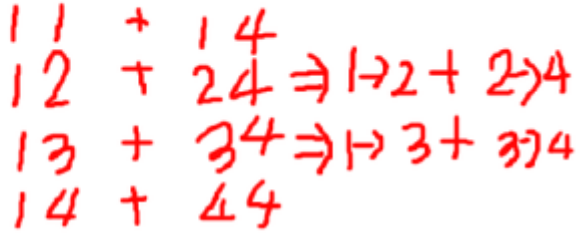
1. 모든 정점의 인접행렬을 구한다.

2. '거쳐가는 정점' 정점을 이용하여 모든 정점에 대하여 최단 경로를 구할 수 있다.

3. 성능 : O(n^3)



첫번째 반복문 - 거쳐가는 정점[k]
두번째 반복문 - 출발하는 정점[i]
세번째 반복문 - 도착하는 정점[j]



출처: https://ssungkang.tistory.com/entry/Algorithm-%ED%94%8C%EB%A1%9C%EC%9D%B4%EB%93%9C-%EC%99%80%EC%83%ACFloyd-Warshall-%EC%95%8C%EA%B3%A0%EB%A6%AC%EC%A6%98 [수학과의 좌충우돌 프로그래밍]

출처: https://mygumi.tistory.com/110 [마이구미의 HelloWorld]

SPFA(Shortest Path Faster Algorithm)

벨만포드의 알고리즘을 향상시킨 알고리즘

벨만포드의 시간복잡도와 같은 O(VE)이지만 평균시간복잡도는 O(E) 더 빠른 알고리즘.

SPFA는 **바뀐 정점과 연결된 간선에 대해서만 업데이트를 진행**한다는 것이다.

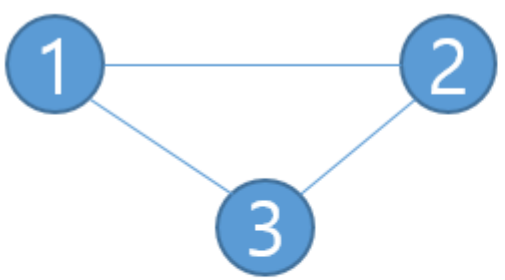
이를 위해 바뀐 정점은 큐를 이용해서 관리하고, 큐에 해당 정점이 있는지 없는지는 배열을 이용해서 체크한다.

출처: https://hongjun7.tistory.com/134 [Note. Hongjun Jang]
출처: https://www.crocus.co.kr/1089 [Crocus]

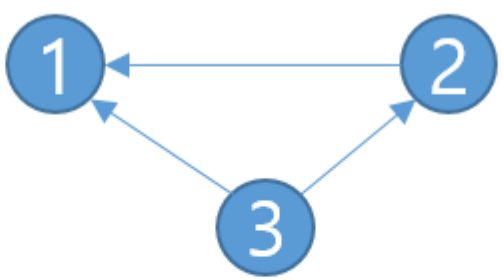
Graph Modeling

Graphical Model이란, 변수들간의 상호 의존 관계를 표현한 확률 모델.

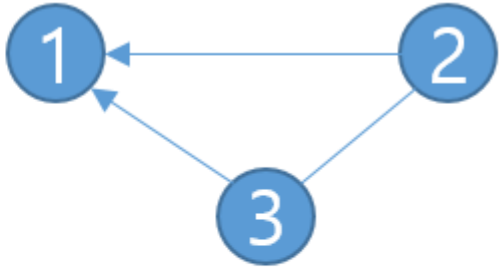
- Directed Graphical Model [방향성]
- Undirected Graphical Model [비방향성]
- Directed Graphical Model과 Undirected Graphical Model [혼합]



비방향성 그래프



방향성 그래프



혼합 그래프

출처: https://lypicfa.tistory.com/320 [스터디 자료실]

출처: https://medium.com/@chullino/graphical-model%EC%9D%B4%EB%9E%80-%EB%AC%B4%EC%97%87%EC%9D%B8%EA%B0%80%EC%9A%94-2d34980e6d1f [Chullin]

Topological Sort

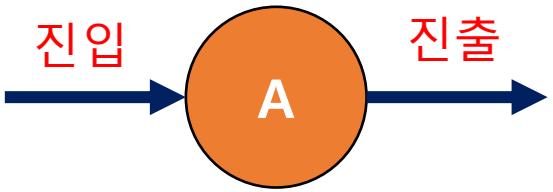
그래프 내에 순서를 가진 정점들을 정렬시키는 방법
[정렬 조건]
사이클이 없는 방향성 그래프

[알고리즘 과정]
0. 리스트를 준비한다.

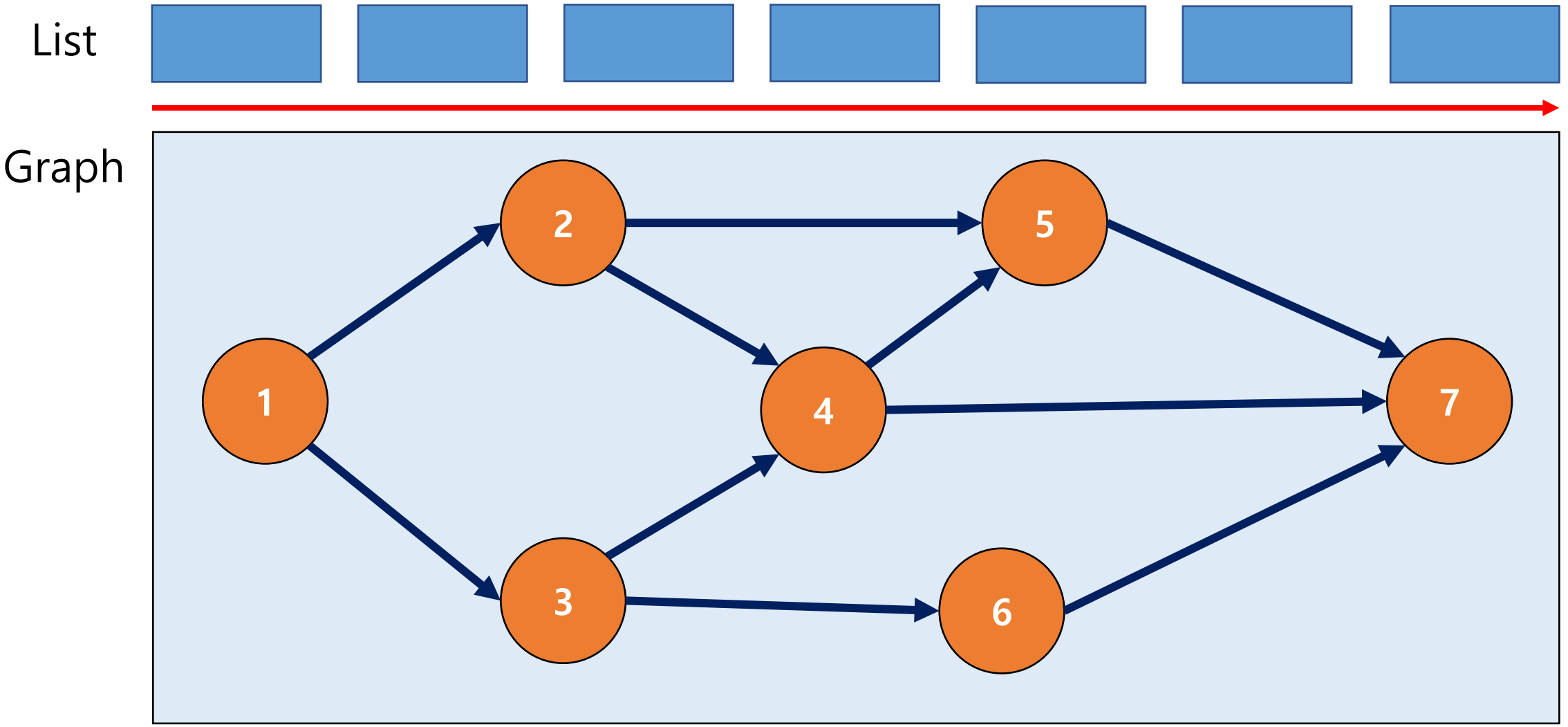
1. 그래프 내에 진입 간선이 없는 정점을 리스트에 넣고 정점과 정점의 인접간선[진출]을 그래프에서 제거한다.
   -> 그래프에서 제거 될때마다 정점 카운트를 낮춘다

2. 리스트에 있는 정점의 진출 간선을 보고 1의 과정을 반복

3. 그래프 내에 정점이 없다면 종료



Topological Sort



소스 : https://m.blog.naver.com/ndb796/221236874984 [안경잡이 개발자]

Reference

[Dijkstra]
출처: https://velog.io/@max9106/Algorithm-%EB%8B%A4%EC%9D%B5%EC%8A%A4%ED%8A%B8%EB%9D%BC-%EC%95%8C%EA%B3%A0%EB%A6%AC%EC%A6%98Dijkstra-Algorithm [Junseo Kim]

[Bellman-Ford]
출처: https://engkimbs.tistory.com/363 [새로비]

[Floyd-Warshall]
출처: https://ssungkang.tistory.com/entry/Algorithm-%ED%94%8C%EB%A1%9C%EC%9D%B4%EB%93%9C-%EC%99%80%EC%83%ACFloyd-Warshall-%EC%95%8C%EA%B3%A0%EB%A6%AC%EC%A6%98 [수학과의 좌충우돌 프로그래밍]
출처: https://mygumi.tistory.com/110 [마이구미의 HelloWorld]

[SPFA]
출처: https://hongjun7.tistory.com/134 [Note. Hongjun Jang]
출처: https://www.crocus.co.kr/1089 [Crocus]

[Graph Modeling]
출처: https://lypicfa.tistory.com/320 [스터디 자료실]
출처: https://medium.com/@chullino/graphical-model%EC%9D%B4%EB%9E%80-%EB%AC%B4%EC%97%87%EC%9D%B8%EA%B0%80%EC%9A%94-2d34980e6d1f
[Chullin]

[위상정렬]
출처: https://m.blog.naver.com/ndb796/221236874984 [안경잡이 개발자]