

3. Data Structure and Algorithms 2

Graph

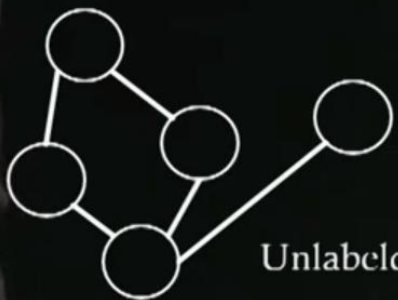
목차

- ▶ Graph
- ▶ Representation of graph
- ▶ Traversing Problem - DFS and BFS
- ▶ Shortest Path Problem - Dijkstra's Algorithm 1
- ▶ Shortest Path Problem - Dijkstra's Algorithm 2
- ▶ Minimum Spanning Tree Problem Prim's Algorithm

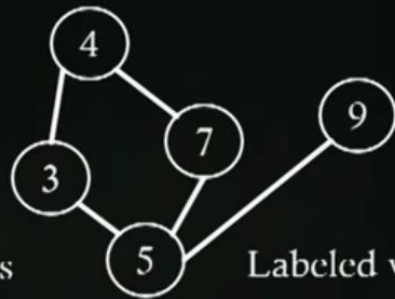
1. Graph

Graphs

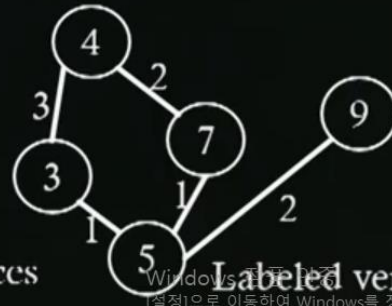
- ◇ Examples of ordered collections, where each item may have successors and predecessors :
 - ◇ List : one predecessor, one successor at most
 - ◇ Tree : one predecessor (parent), several successors (children)
 - ◇ Graph : several predecessors and successors
- ◇ Graph $G = (V, E)$
 - ◇ $V = \{ v_i \}$: a finite non-empty set of vertices (or nodes)
 - ◇ $E = \{ e_i \}$: a finite (possibly empty) set of edges (or arcs)
 - ◇ e_i connects two vertices in V



Unlabeled vertices



Labeled vertices



Labeled vertices
And Weighted edges

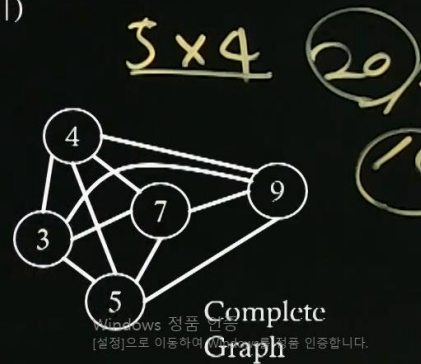
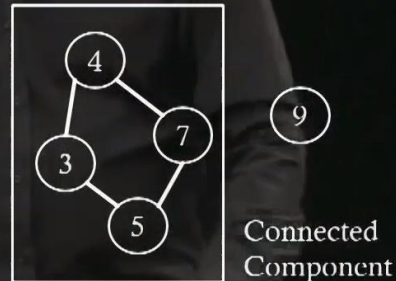
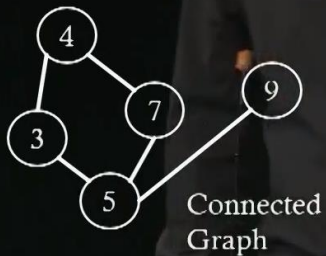
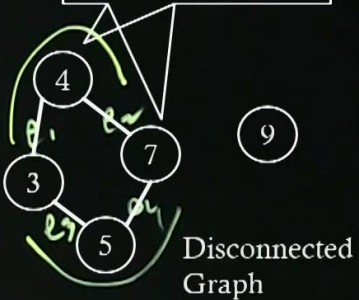
- Graph : 마음대로 연결이 되어 있는 것
- Graph $G = (V, E)$
- V : vertices (or node)
- E : set of edges(or link, arcs)
- 세가지로
- Unlabeled, Labeled vertices, Labeled vertices And Weighted edges

1. Graph

Graph terminology

- ◇ **adjacent, neighbor**
- ◇ **path** between A and B : a sequence of edges connecting A and B
- ◇ **connected graph** : path from each to every other vertex
- ◇ **connected component** : graph subset containing the set of vertices reachable from a vertex and their edges
- ◇ **complete graph** : edge for every pair of vertices
 - ◇ dense graph : close to complete graph $|E| = O(|V|^2)$
 - ◇ sparse graph : far from complete graph $|E| = O(|V|)$

Adjacent vertex or Neighbor

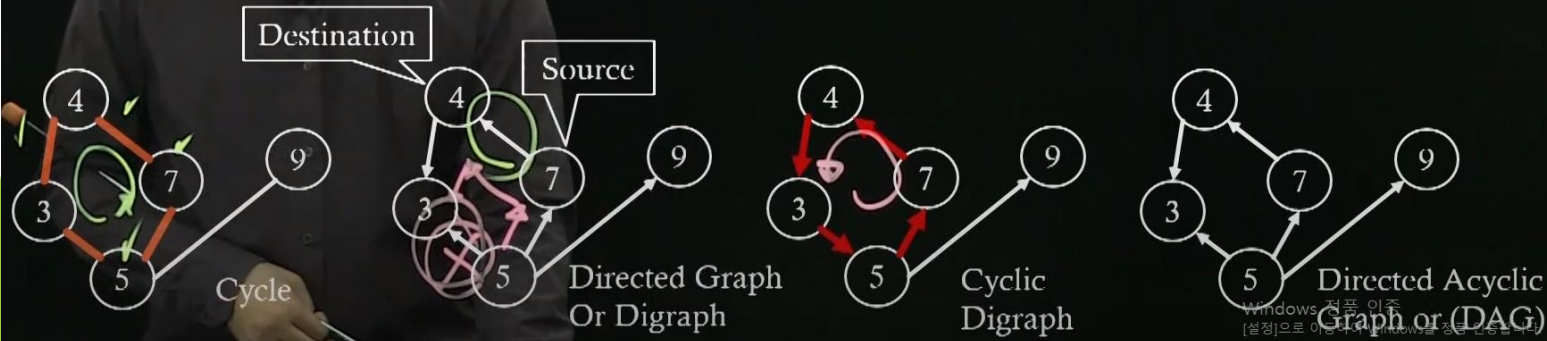


- Adjacent, neighbor : 서로 edge로 묶여 있는 것
- Path : 특정한 노드에서 다른 노드로 가는 길
- Connected graph : 모든 노드들이 연결된 Graph
- Connected component : path를 planning 할수 있는 subset graph
- Complete graph : 모든 노드들에 대해서 모든 path가 존재하는 graph

1. Graph

Graph terminology

- ◇ **Cycle** : a path starting from a node and ending the node itself
- ◇ **Directed edge** : an edge with direction (source and destination)
- ◇ **Digraph** : Directed graph.
 - ◇ Graph with directed edges
- ◇ **DAG** : Directed **Acyclic** Graph
 - ◇ Directed graph without cycle



- **Cycle** : 어떤 노드에서 시작해서 그것과 동일한 노드에서 끝나는 것.
- **Directed edge** : 방향성이 존재하는 edge
- **Digraph** : Cycle이 있는 directed graph
- **GAG** : Cycle이 없는 directed graph

2. Representation of graph

Data structure for graphs

◇ To store a graph

◇ Store a set of vertexes

◇ 0,1,2,3,4,5,6,7,8,9

◇ Store a set of edges

◇ (0,1), (1,3), (2,0), (5,3)...

◇ How to store?

◇ Storing vertexes

◇ Simple.

◇ Linked list, BST, Hash....

◇ Store edges

◇ Fundamentally, a pair of values

◇ Initially, a two-dimensional matrix

◇ Space: $O(V^2)$

◇ Time: $O(1)$

◇ However....

◇ Graph density becomes problem

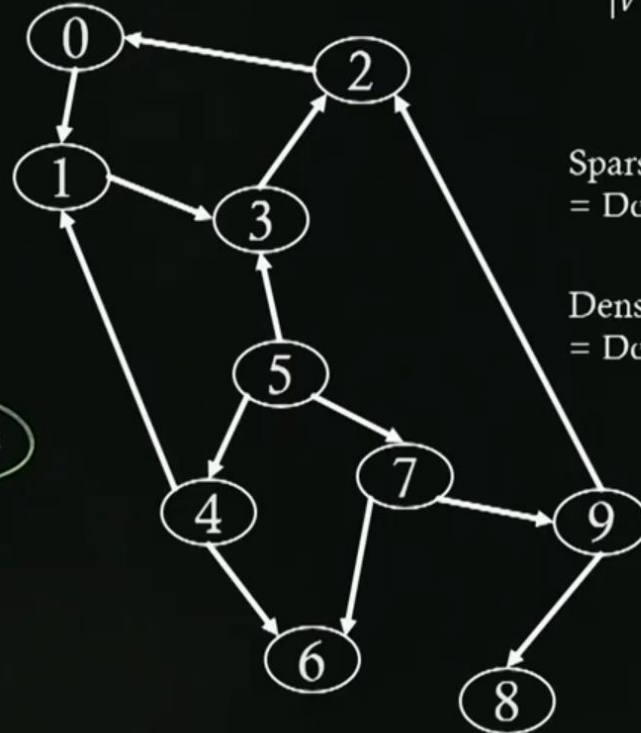
◇ So, adjacency list

◇ Space: $O(E)$

◇ Time: $O(E)$

Data structure for graphs

$$\text{Density} = \frac{|E|}{|V|(|V| - 1)}$$



Sparse graph
= Density $\rightarrow 0$

Dense graph
= Density $\rightarrow 1$

- Vertexes 저장하고
- Edges들을 저장한다.
- Vertexes를 저장하는 방법은 linked list, BST, Hash의 방법으로 저장하면 된다.
- Edges를 저장하는 것은 a pair of values를 저장
- Value 다음에 value가 있으면 1, 없으면 0으로 저장하는 two-dimensional matrix가 있다.
- 문제는 dense graph일수록 vertexes간 다양한 저장이 생긴다.
- 아주 많은 space가 필요하고 지금까지도 감당할 수 없다고 한다.
- 그래서 adjacency list를 활용해서 저장

2. Representation of graph

Adjacency List Representation for Sparse Graph

◇ Array Representation :
Adjacency Matrix

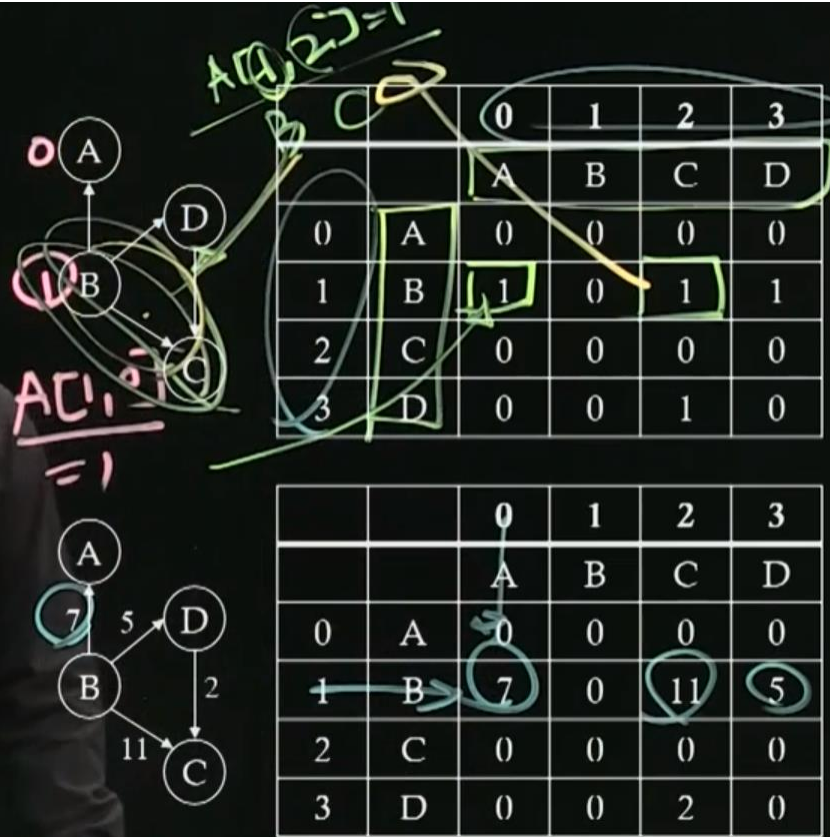
◇ Linked Representation :
Adjacency List

◇ Adjacency Matrix

◇ $A[i][j] = 1$ if $(v_i, v_j) \in E$
0 otherwise

◇ Adjacency Matrix for
weighted graph

◇ edge weight value
instead of 0/1



- Dense graph의 경우 Matrix를 활용하여 저장
- 저장할 edge가 많기 때문에 Matrix를 활용
- Adjacency Matrix : Array를 활용하는 것
- Adjacency List : linked list를 활용하는 것
- $A[i][j] = 1$ 이면 vertex가 존재
- $A[i][j] = 0$ 이면 vertex가 존재 하지 않음
- Adjacency Matrix를 weighted graph에 사용하면 0 과1 대신에 edge weight value를 사용
- A, B, C, D에 index번호를 0, 1, 2, 3으로 부여
- Weighted graph는 0,1대신 weight로.
- Matrix는 0이 많음 - 매우 sparse하게 값이 저장

2. Representation of graph

Matrix Representation for Dense Graph

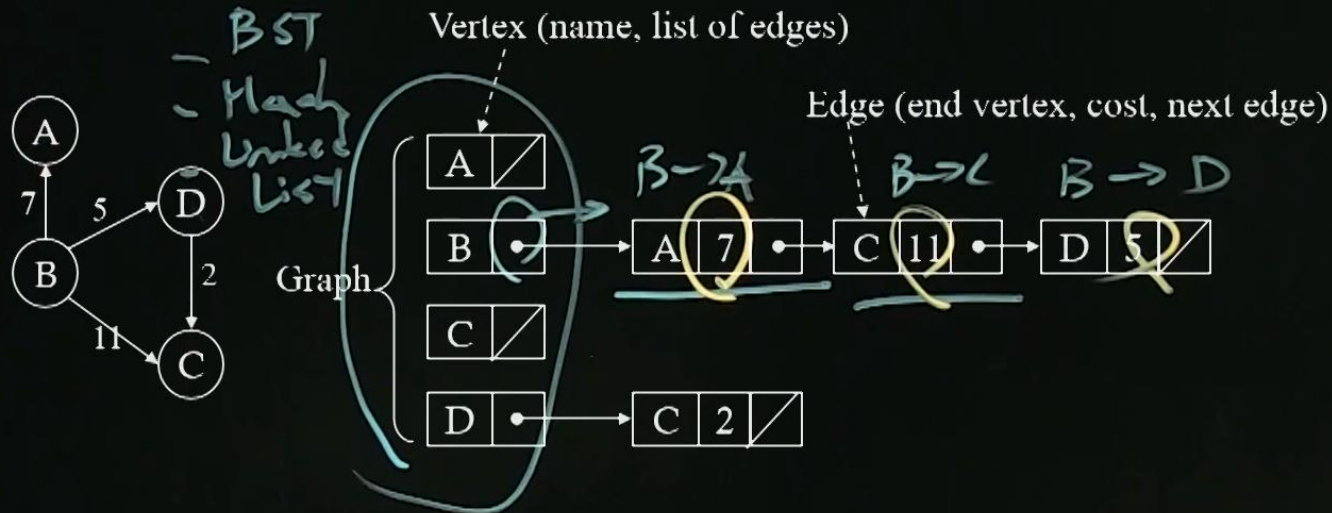
◇ Adjacency matrix : storage waste for sparse graph

◇ Adjacency list

◇ For each vertex, make a linked list of edges starting from the vertex

◇ Edge weight can be stored in 'Edge'

◇ Storage efficient



- Matrix는 space공간이 낭비가 많아서 Adjacency list를 활용한다.
- 각 vertex에 대해서 linked list로 가는 레퍼런스를 하나 만든다.
- Edge weight는 edge의 value로 저장할 수 있음
- B에서 출발하는 edge를 기록한다.
- B-A, B-C, B-D
- Weight도 같이 저장한다.
- 그래서 먼저 Vertex를 찾아가고 거기의 Linked list에 들어가 값을 찾으면 됨
- 상위 구조나 하위 구조는 BST, Hash, Linked List를 선택하여 사용할 수 있다.

3. Traversing Problem - DFS and BFS

Operations of graph data structure

◇ Operations on graphs

◇ Operation of retrieving vertexes

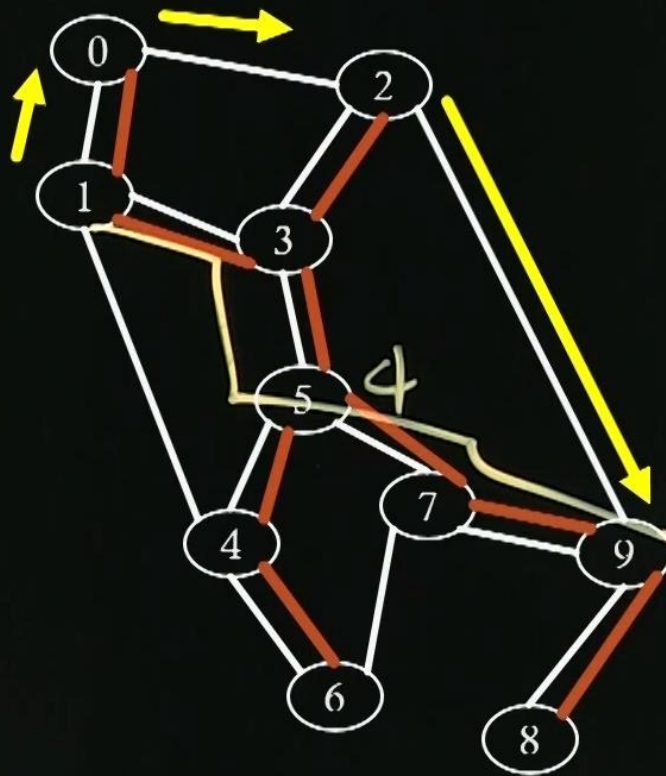
- ◇ BFS traverse
- ◇ DFS traverse

◇ Operation of finding shortest paths

- ◇ The shortest path
 - ◇ From vertex 1
 - ◇ To vertex 9

◇ Operation of finding a set of path to control whole vertexes

- ◇ The minimum spanning tree



- Traversing : graph의 모든 vertex를 꺼내 오는 것
- BFS traverse 와 DFS traverse 의 방법이 있다
- 목적지에 가장 빠르게 갈 수 있는 path를 구하는것도 있다.
- 전체 vertexes를 통제하기 위한 어떤 edge를 활용 할 수 있는지 알아보는 subset을 찾아내는 것도 있다.

3. Traversing Problem - DFS and BFS

Detour: Tree traversing

◆ Tree

- ◆ Complicated than a list
- ◆ Multiple ways to show the entire dataset

◆ If it were a list

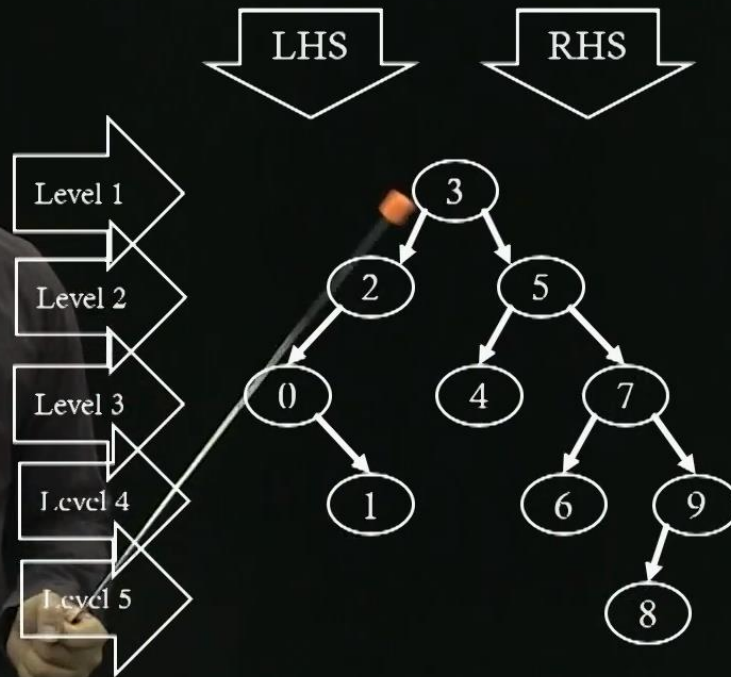
- ◆ Just show the values from the beginning to the end

◆ Since this is a BST

- ◆ You have to choose what to show at a time
 - ◆ The value in LHS
 - ◆ The value in RHS
 - ◆ The value that you have

- ◆ Hence there are multiple traversing approaches

Inserting 3, 2, 0, 5, 7, 4, 6, 1, 9, 8



- Tree Traversing :
- Level traversing
- In Order traversing
- Pre Order traversing
- Post Order traversing

3. Traversing Problem - DFS and BFS

DFS vs. BFS traverse on graphs

DFS utilizes

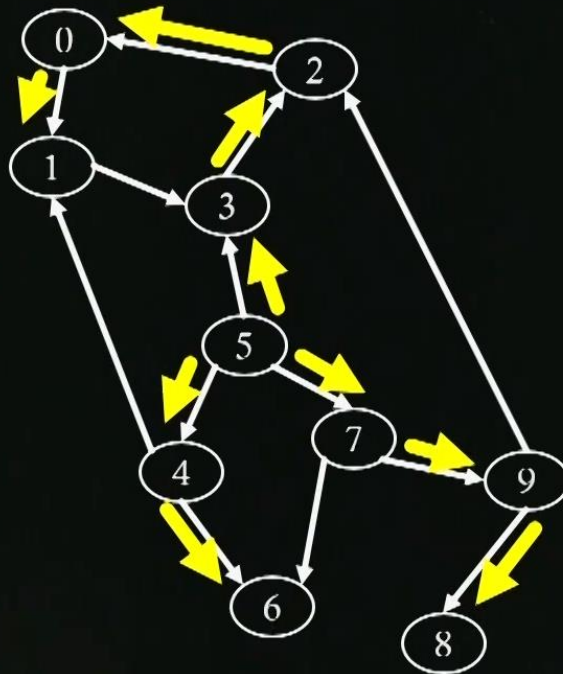
- Stacks or recursions that imitates the stack operations
- Pre-order traverse
- In-order traverse
- Post-order traverse
- In graphs, often only pre-order traverse is used

BFS utilizes

- Queues
- Level-order traverse

Having said this,

- Tree is a directed acyclic graph.
- Graph may not be a DAG
- Then....
 - You have to check the repeated visits to avoid falling into a cycle



- DFS :
- Graph에서는 In-order traverse가 불가능
- Pre와 Post가 될 수 있는데 Post를 하면 너무 복잡해 질 수 있어서 Pre를 선호한다.
- Traveling할때 지나온 것을 기억해서 한번 travel한 것에는 다시 못가게 한다.
- (5-3-2-0-1-4-6-7-9-8)
- BFS :
- Tree에서 level-order-traverse 와 같음
- Enqueue :
- 5 - 3,4,7 - 2 - 6 - 9 - 0 - 8 - 1
- Dequeue :
- 5 - 3 - 4 - 7 - 2 - 6 - 9 - 0 - 8 - 1

3. Traversing Problem - DFS and BFS

DFS vs. BFS traverse on graphs

DFS vs. BFS traverse on graphs

DFS utilizes

- Stacks of recursions that imitates the stack operations

- Pre-order traverse

- In-order traverse

- Post-order traverse

- In graphs, often only pre-order traverse is used

BFS utilizes

- Queues

- Level-order traverse

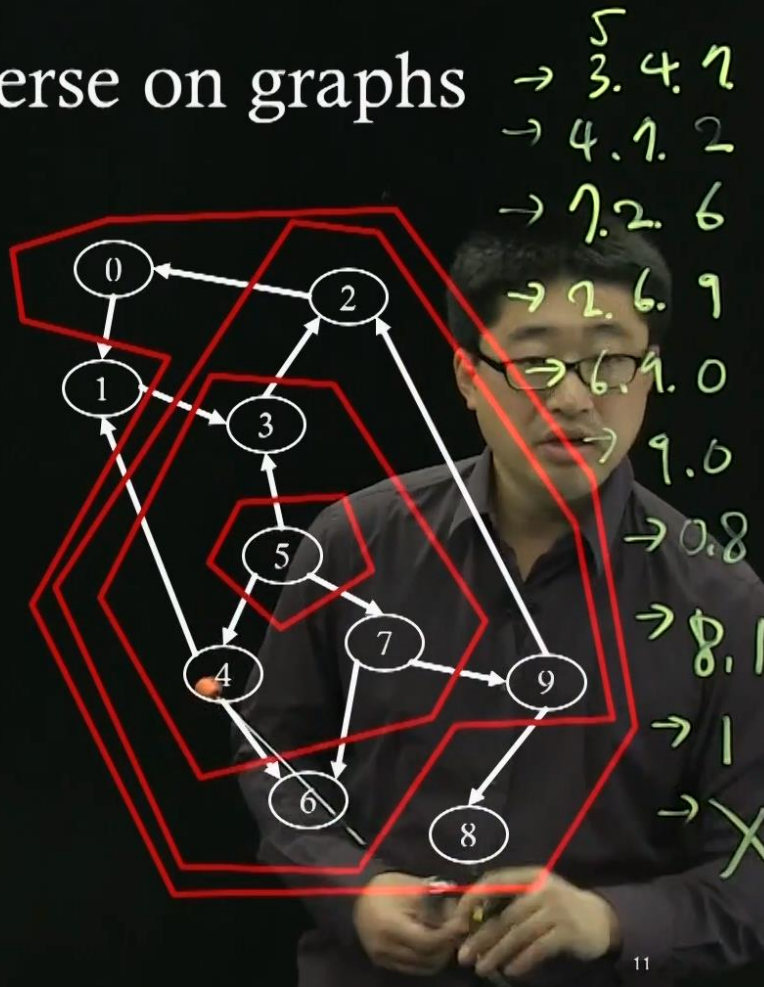
Having said this,

- Tree is a directed acyclic graph.

- Graph may not be a DAG

- Then....

- You have to check the repeated visits to avoid falling into a cycle



- DFS :
- Graph에서는 In-order traverse가 불가능
- Pre와 Post가 될 수 있는데 Post를 하면 너무 복잡해 질 수 있어서 Pre를 선호한다.
- Traveling할때 지나온 것을 기억해서 한번 travel한 것에는 다시 못가게 한다.
- (5-3-2-0-1-4-6-7-9-8)
- BFS :
- Tree에서 level-order-traverse 와 같음
- Enqueue :
- 5 - 3,4,7 - 2 - 6 - 9 - 0 - 8 - 1
- Dequeue :
- 5 - 3 - 4 - 7 - 2 - 6 - 9 - 0 - 8 - 1

4. Shortest Path Problem - Dijkstra's Algorithm 1

Single-Source Shortest Path Problem

◇ One recurring problem in graph

◇ Happens in

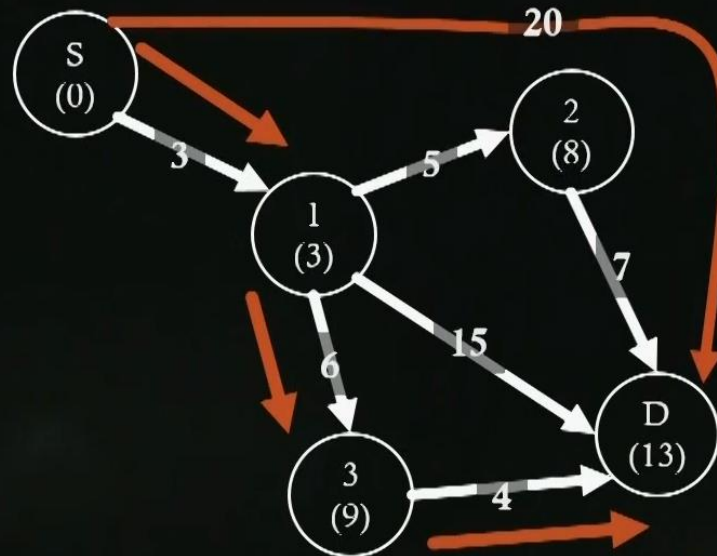
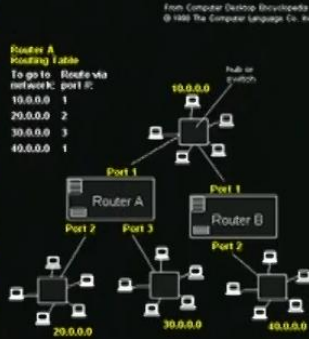
- ◇ Path finding
- ◇ Routing on comm. networks
- ◇ Social networks

◇ We know where we are

◇ We want to know how long to travel to our destination

◇ Terminology

- ◇ Source = where we start
- ◇ Destination = where we arrive



- 네비게이션에서 경로찾기와 유사
- 친구찾기 등 소셜네트워크에서도 많이 사용
- 어디서 시작할지(**source**)에 대해서 알 수 있다.
- 목적지까지 얼마나 걸릴것인지 알기 위함
- **Destination** : 어디로 가야하는지

5. Shortest Path Problem - Dijkstra's Algorithm 2

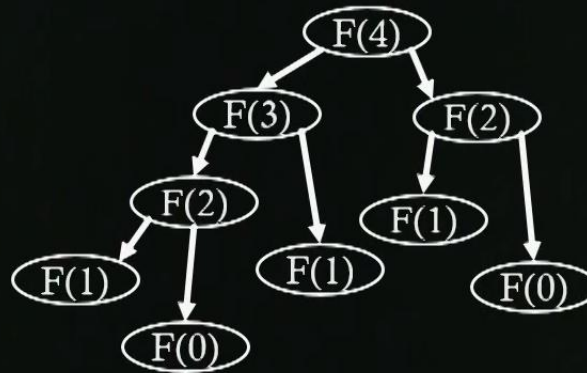
Detour: Dynamic Programming

Dynamic programming:

- ◇ A general algorithm design technique for solving problems defined by or formulated as *recurrences with overlapping sub-instances*
- ◇ In this context, Programming == Planning

Main storyline

- ◇ Setting up a recurrence
 - ◇ Relating a solution of a larger instance to solutions of some smaller instances
 - ◇ Solve small instances once
 - ◇ Record solutions in a table
 - ◇ Extract a solution of a larger instance from the table

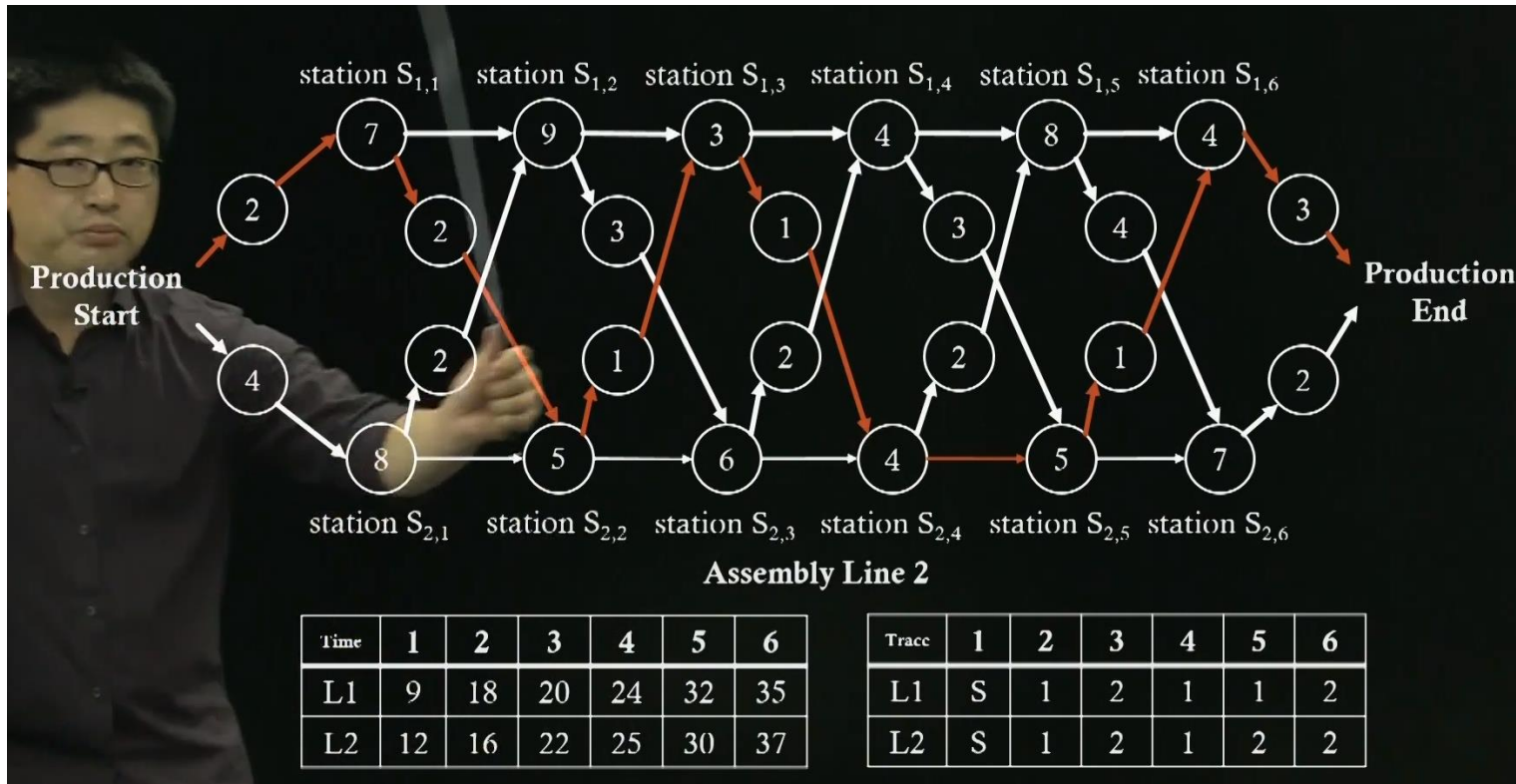


Instance	Solution
F(0)	0
F(1)	1
F(2)	1
F(3)	2
F(4)	?

- 피보나치4를 먼저 구하지 말고 밑에서 부터 구하자
- 이 방법을 이용해서 **shortest path problem**을 해결한다.

5. Shortest Path Problem - Dijkstra's Algorithm 2

Process of Assembly Line Scheduling



- 어떤 라인을 선택해야 가장 빠르게 productio을 생산할 수 있는지.
- 이것은 목적지에 가장 빠른시간안게 갈 수 있는 path를 구하는 것과 같다.
- 이 공정이 graph structure 이다.

5. Shortest Path Problem - Dijkstra's Algorithm 2

Process of Assembly Line Scheduling

Graph
single source
problem

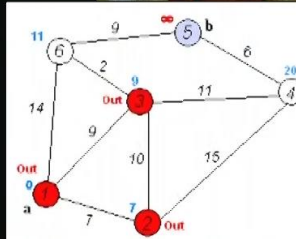
- ◇ V = the set of vertexes
- ◇ W = the set of weights on edges
- ◇ s = the source vertex
- ◇ Dijkstra's algorithm(V, W, s)
- ◇ $\text{dist} = \{\}$
- ◇ For itr in V
- ◇ $\text{dist}[v] = 99999$
- ◇ $\text{dist}[s] = 0$
- ◇ While $\text{size}(V) \neq 0$
- ◇ $u = \text{getVertexWithMinDistance}(V, \text{dist})$
- ◇ $V.\text{remove}(u)$
- ◇ For neighbor in $\text{getNeighbors}(u)$
- ◇ If $\text{dist}[\text{neighbor}] > \text{dist}[u] + w(u, \text{neighbor})$
- ◇ $\text{dist}[\text{neighbor}] = \text{dist}[u] + w(u, \text{neighbor})$
- ◇ Return dist

Memoization Table

Retrieving minimum distance from a list. Does that ring a bell?

Somchow, I know it takes 9 to reach 3

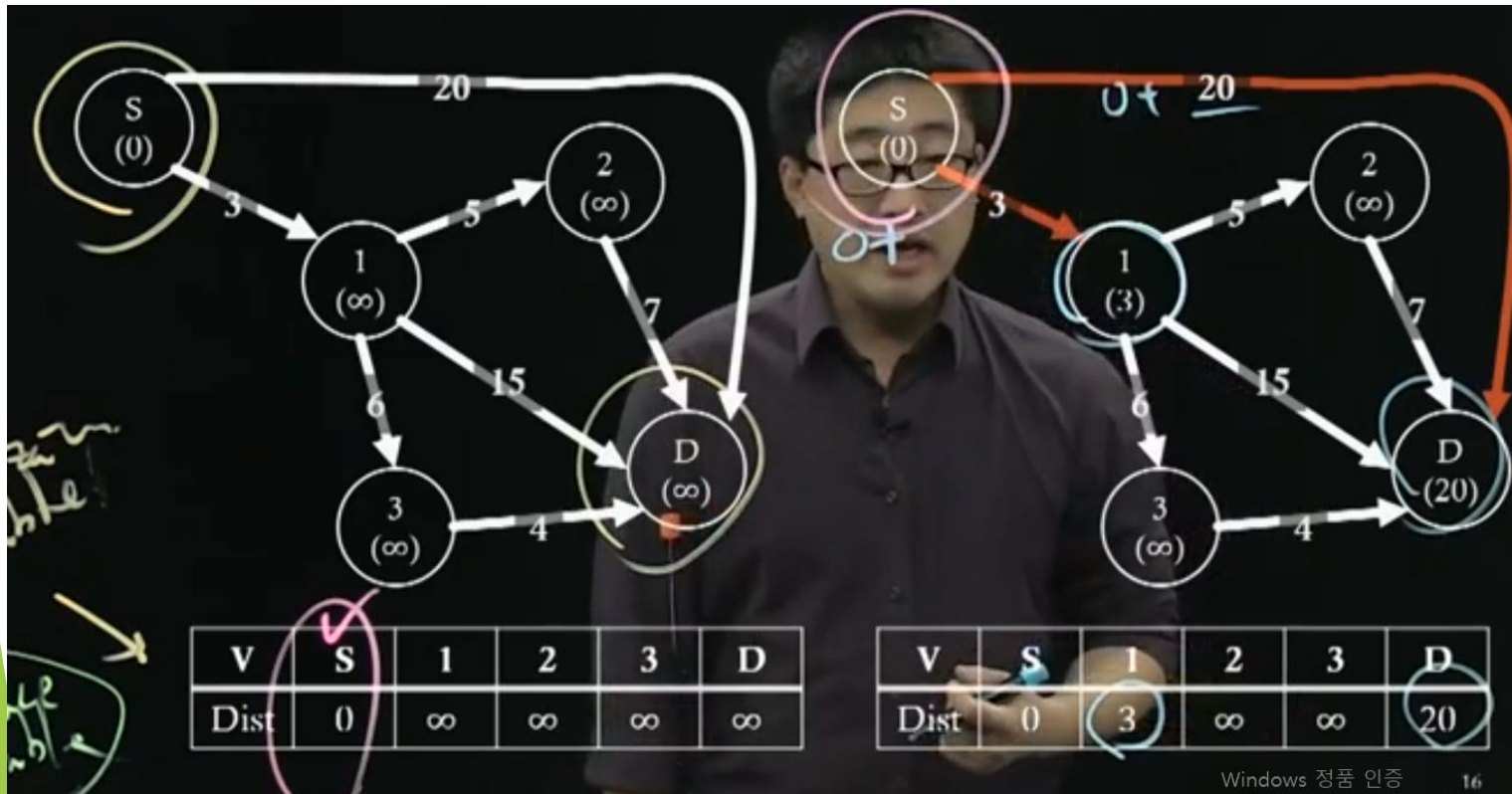
Will I update the time 20 with time $9+4$?



- V 와 W : graph의 조건
- s : single source parameter
- Dijkstra's algorithm(V, W, s)
- Dist : 과거를 기록하는 table
- $\text{Dist}[s]$: 시작하는 위치니 0
- Vertex가 0이 될때까지 while loop를 돌림
- Traverse 와 유사하다
- 남아 있는 V 중에서 dist 가 가장 작으거 가져옴
- If문 무한대보다 neighbor가 작다
- (0) dist에서 온 값, 20 W(edge)에서 온 값
- 3(9) new addef covered
- S-3-D로 가는 길이 13으로 더 짧아 20을 13으로 업데이트한다.

5. Shortest Path Problem - Dijkstra's Algorithm 2

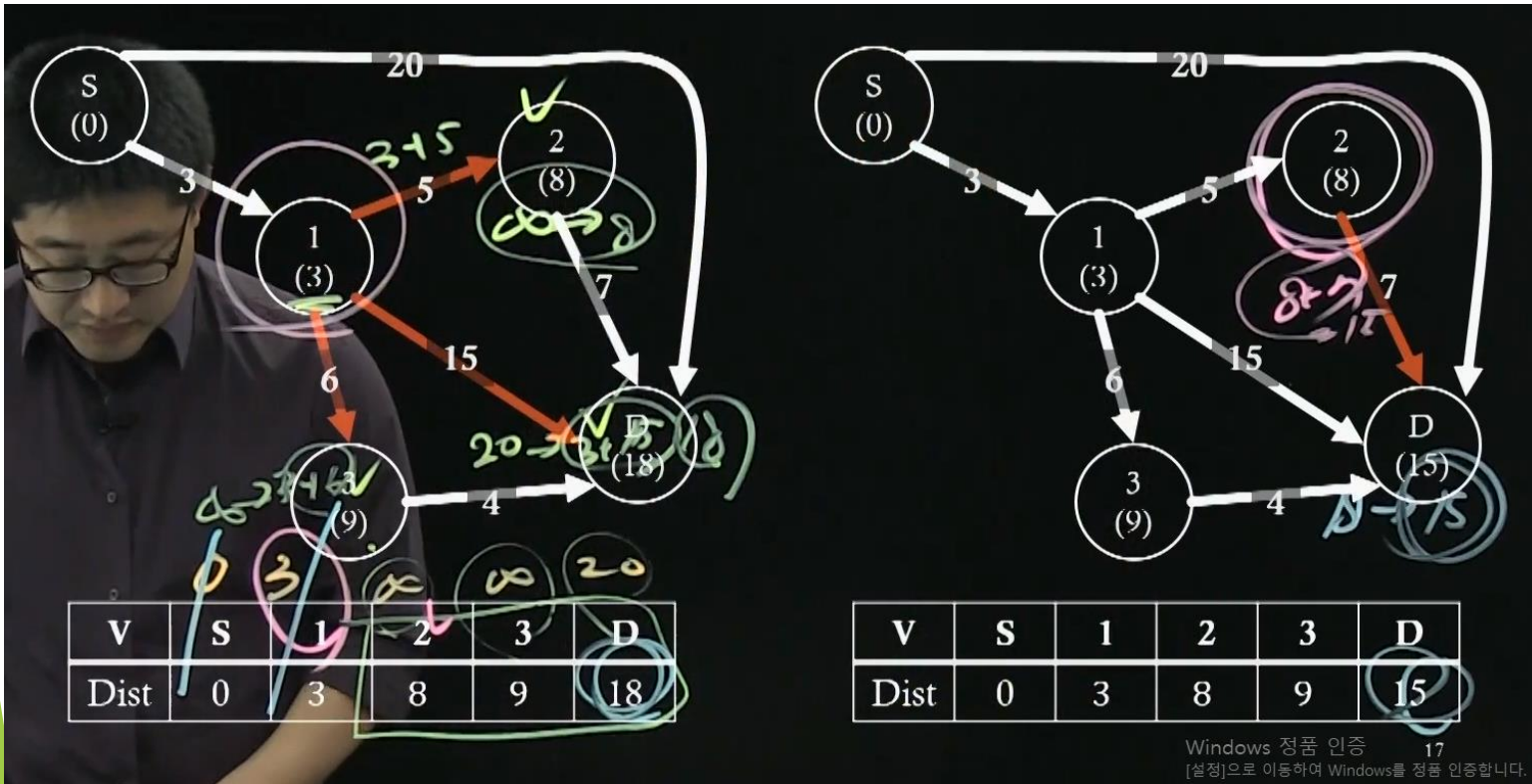
Progress of Dijkstra's algorithm (1)



- Source에서 destination까지 가는 것
- Source만 0이고 나머지는 다 무한대
- 가장 작은 vertex인 0을 선택, s선택
- S의 네이버를 찾는다.
- 1로 가는 path는 3으로 업데이트
- D로 가는 path는 20으로 업데이트

5. Shortest Path Problem - Dijkstra's Algorithm 2

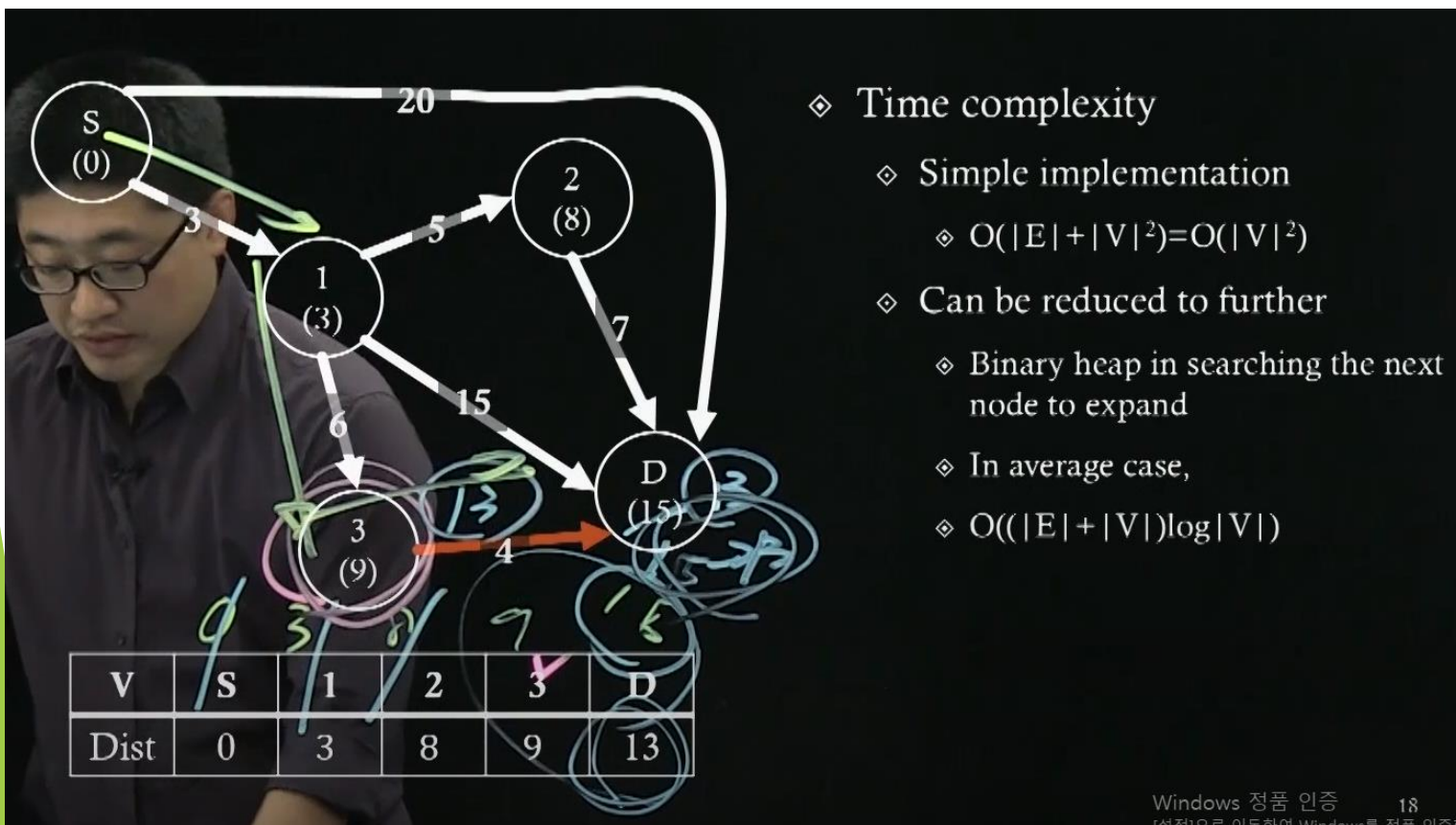
Progress of Dijkstra's algorithm (2)



- Source에서 destination까지 가는 것
- Source만 0이고 나머지는 다 무한대
- 가장 작은 vertex인 0을 선택, s선택
- S의 네이버를 찾는다.
- 1로 가는 path는 3으로 업데이트
- D로 가는 path는 20으로 업데이트
- 0은 한번 covered되서 지워준다.
- 1이 선택되고, 네이버를 찾는다
- 2로 가는 path가 3+5 거리니 업데이트
- 3으로 가는 path가 3+6 거리니 업데이트
- D로 가는 path가 3+15 거리니 업데이트
- 1번은 지워주고 나머지중 가장 작은 2를 선택
- 2의 네이버를 찾는다
- D로가는 path가 15 거리니 18보다 작아서 업데이트

5. Shortest Path Problem - Dijkstra's Algorithm 2

Progress of Dijkstra's algorithm (3)



Time complexity

Simple implementation

$$O(|E| + |V|^2) = O(|V|^2)$$

Can be reduced to further

Binary heap in searching the next node to expand

In average case,

$$O((|E| + |V|) \log |V|)$$

- Source에서 destination까지 가는 것
- Source만 0이고 나머지는 다 무한대
- 가장 작은 vertex인 0을 선택, s선택
- S의 네이버를 찾는다.
- 1로 가는 path는 3으로 업데이트
- D로 가는 path는 20으로 업데이트
- 0은 한번 covered되서 지워준다.
- 1이 선택되고, 네이버를 찾는다
- 2로 가는 path가 3+5 거리니 업데이트
- 3으로 가는 path가 3+6 거리니 업데이트
- D로 가는 path가 3+15 거리니 업데이트
- 1번은 지워주고 나머지중 가장 작은 2를 선택
- 2의 네이버를 찾는다
- D로가는 path가 15 거리니 18보다 작아서 업데이트
- 2도 지우고 나머지중 가장 작은 3을 선택
- D로 가는 path가 13 거리니 다시 업데이트
- 따라서 D까지 13거리게 제일 빠르다

6. Minimum Spanning Tree Problem Prim's Algorithm

Minimum Spanning Tree Problem

Shortest-path problem

- ◇ Path planning from a selected source
- ◇ Many times, algorithm for planning

Minimum spanning tree

- ◇ Network control problem
- ◇ All vertex coverage with minimum cost
- ◇ Algorithm from network design
 - ◇ Telephone network
 - ◇ Electricity grid network
 - ◇ TV cable network
 - ◇ Computer network
 - ◇ Road network
- ◇ Evolving to the influence propagation tree
 - ◇ Social network influence
 - ◇ From one politician to all tweeter accounts

- Minimum spanning tree - coverage가 핵심
- 네트워크 컨트롤 문제(얼마나 제어를 할수 있느냐)
- 모든 vertex를 minimum cost로 coverage하는것을 찾는것
- 여러 네트워크 디자인에 많이 사용됨
- 하얀점에서 모든 vertex를 cover할 수 있는지
- 저 점을 위로 쪽 올리면 Tree처럼 됨
- 사이클이 생기지 않기 때문에 가능

6. Minimum Spanning Tree Problem Prim's Algorithm

Prim's algorithm

◆ V = the set of vertexes

◆ U = the covered set of vertexes

◆ W = the set of weights on edges

◆ E = the selected set of edges

◆ s = the source vertex

◆ Prim's algorithm(V, W, s)

- ◆ $U = \{s\}, E = \{\}$
- ◆ While $(V - U) \neq \emptyset$
 - ◆ $edges = \text{Find edges of } (src, dst)$
s.t. $src \in U, dst \in V - U$
 - ◆ $e = \text{getEdgeWithMinimumWeight}(edges)$
 - ◆ $E = E \cup \{e\}$
 - ◆ $U = U \cup \{e.dst\}$
- ◆ Return E and U

Edges of don't care vertex

Covered Nodes

Uncovered Nodes

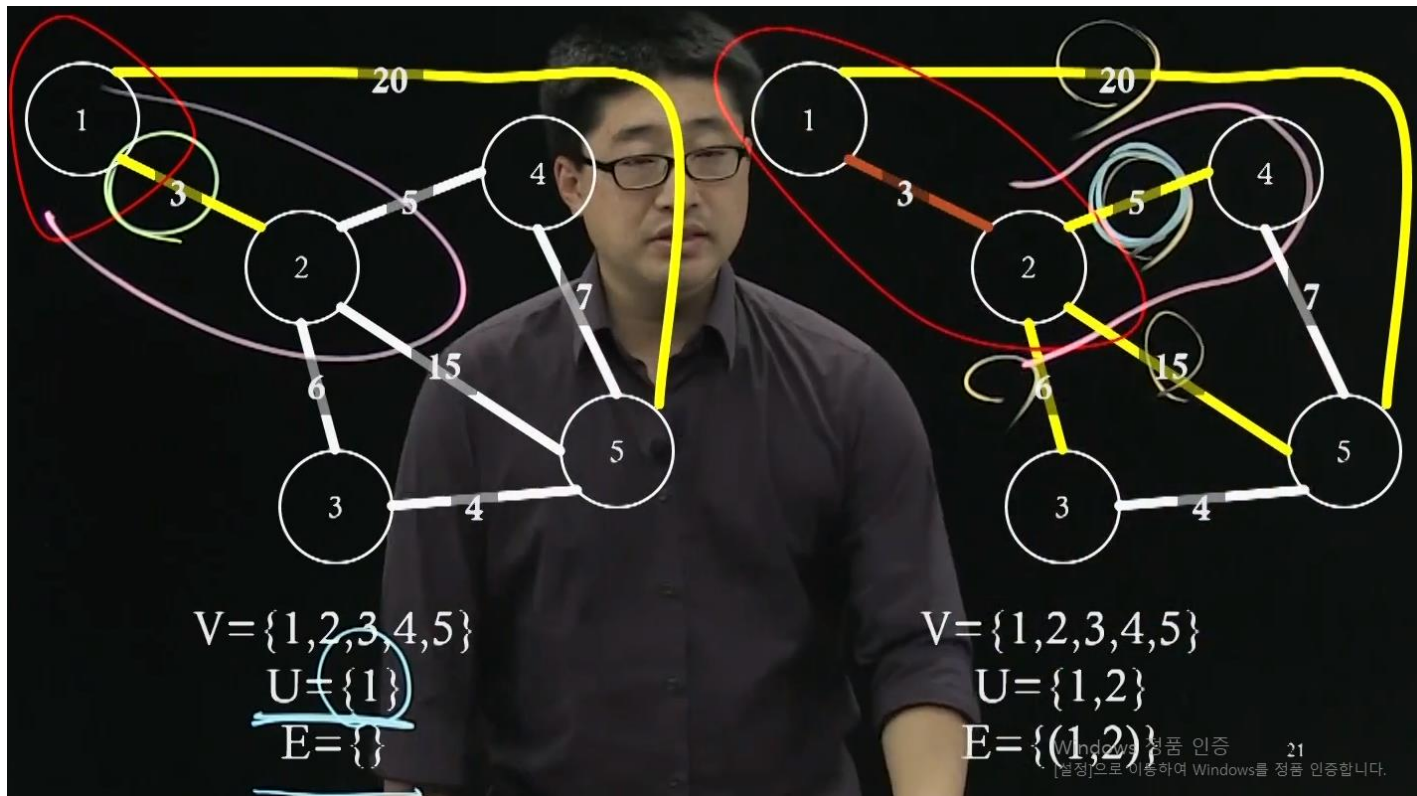
Edges of don't care

Windows 정품 인증
[설정]으로 이동하여 Windows를 정품 인증합니다.

- V, W : graph를 나타내는 값
- U : vertex 들의 covered 될수 있는 set
- E : 선택된 edge
- S : source vertex
- U 를 점차 키워 나간다.
- While loop를 통해서 키워나간다.
- $V - U$: non-covered vertex
- $V - U$ 가 empty가 아닐때까지 loop를 돌린다.
- Src는 U 에 cover, dst $V - U$ 에 non-cover
- Covered와 non-covered를 분리하고 그것들을 연결하는 edge를 찾는것
- E - minimum weight를 선택하고
- Edge를 selected set에 추가하고
- 이 edge의 dst는 non-cover에서 covered 상황으로 만들어준다.

6. Minimum Spanning Tree Problem Prim's Algorithm

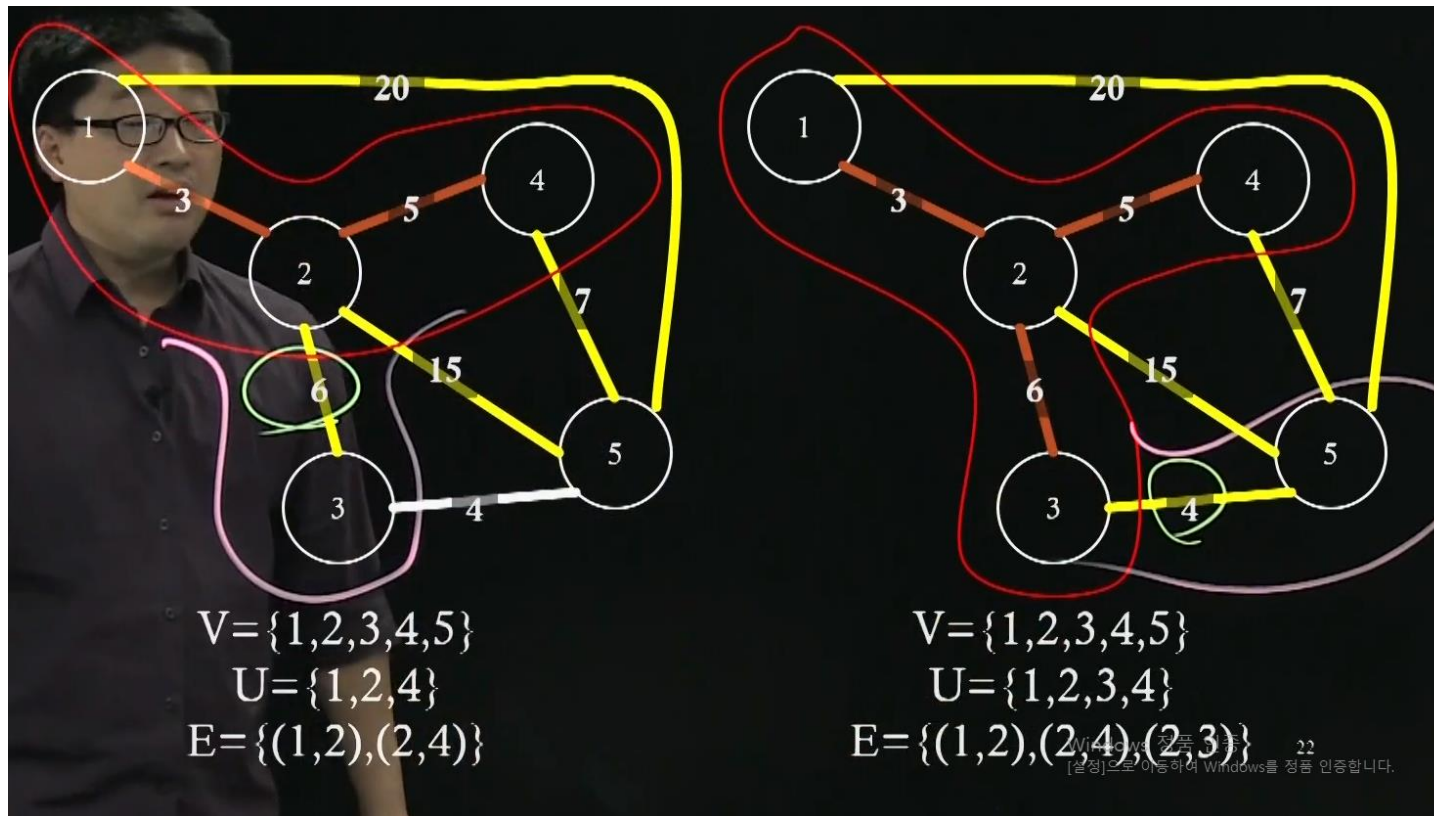
Progress of Prim's Algorithm (1)



- 1부터 시작
- 이 시기는 오로지 1만 covered 됨
- Covered와 non-covered edge인 3과 20중에 minimum 3을 선택하고
- 연결된 2는 covered 포함한다.
- 그다음 Covered와 non-covered edge에서 minimum 5를 선택하고 연결하는 4는 covered됨
- 다음 Covered와 non-covered edge에서 minimum 6을 선택하고 3이 covered set에 포함
- 다음 브릿지 edge중에 제일 작은값 4를 선택하고 5를 covered set에 포함

6. Minimum Spanning Tree Problem Prim's Algorithm

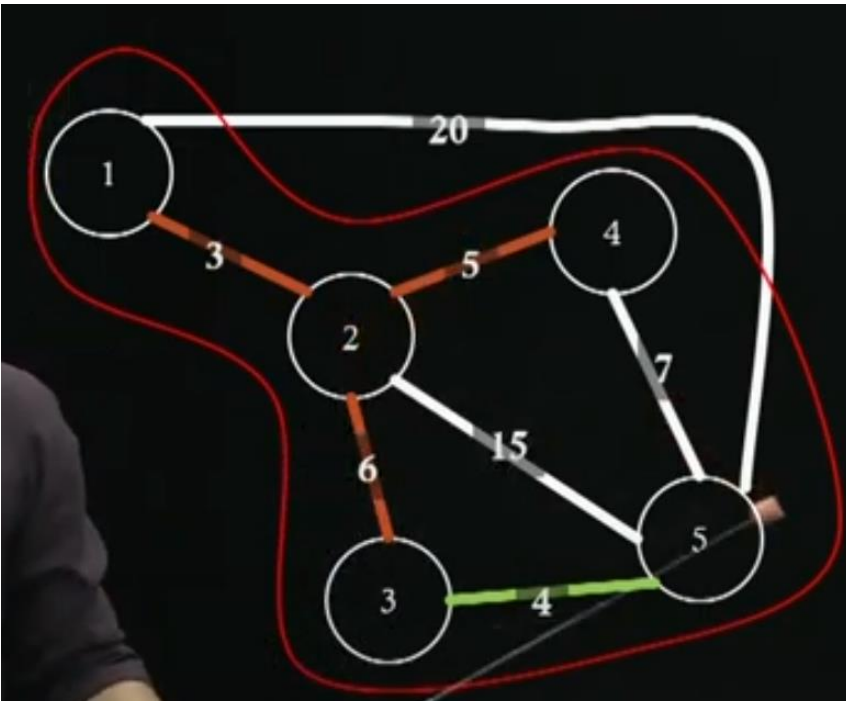
Progress of Prim's Algorithm (2)



- 1부터 시작
- 이 시기는 오로지 1만 covered 됨
- Covered와 non-covered edge인 3과 20중에 minimum 3을 선택하고
- 연결된 2는 covered 포함한다.
- 그다음 Covered와 non-covered edge에서 minimum 5를 선택하고 연결하는 4는 covered됨
- 다음 Covered와 non-covered edge에서 minimum 6을 선택하고 3이 covered set에 포함
- 다음 브릿지 edge중에 제일 작은값 4를 선택하고 5를 covered set에 포함

6. Minimum Spanning Tree Problem Prim's Algorithm

Progress of Prim's Algorithm (2)



$V = \{1, 2, 3, 4, 5\}$
 $U = \{1, 2, 3, 4, 5\}$
 $E = \{(1, 2), (2, 4), (2, 3), (3, 5)\}$

- ◇ Time complexity
 - ◇ $O((|E| + |V|) \log |V|)$
 - ◇ We will not prove this
 - ◇ Same time complexity to the Dijkstra's algorithm

Windows 정품 인증
[설정]으로 이동하여 Windows를 정품 인증합니다.

- 수고하셨습니다.