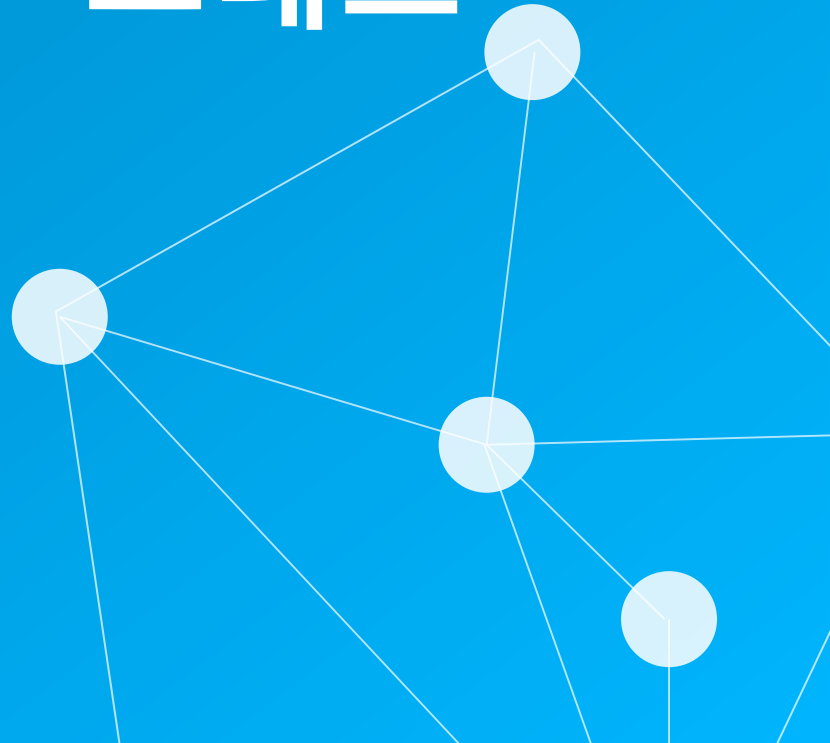


08 CHAPTER

그래프

SW알고리즘개발
12주차



8장. 그래프



08-1 그래프란?

08-2 그래프의 표현

08-3 그래프 순회

08-4 신장 트리

Greedy 알고리즘 설계(9장 탐욕적 설계)

08-5 최소비용 신장 트리: 프림의 알고리즘

최소 힙 (Mini-Heap) 자료구조

최단 경로 문제: Dijkstra 알고리즘

-

- 정점의 집합: V

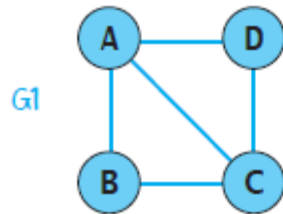
간선의 집합: E

그래프: $G=(V,E)$

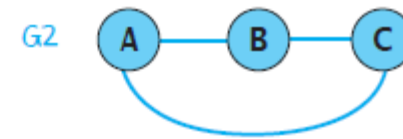
그래프의 종류



- 무방향 그래프(undirected graph)

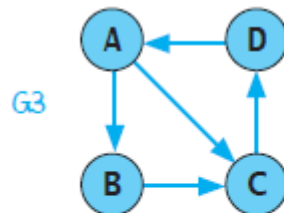


$V(G1) = \{A, B, C, D\}$
 $E(G1) = \{(A,B), (A,C), (A,D), (B,C), (C,D)\}$

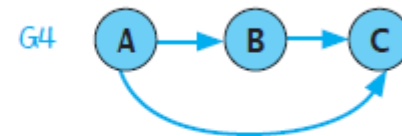


$V(G2) = \{A, B, C\}$
 $E(G2) = \{(A,B), (A,C), (B,C)\}$

- 방향 그래프(directed graph)



$V(G3) = \{A, B, C, D\}$
 $E(G3) = \{\langle A,B \rangle, \langle A,C \rangle, \langle B,C \rangle, \langle C,D \rangle, \langle D,A \rangle\}$

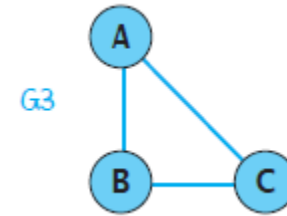
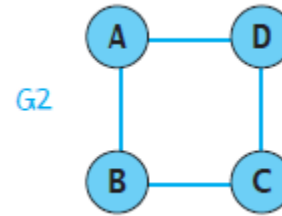
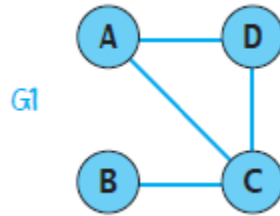
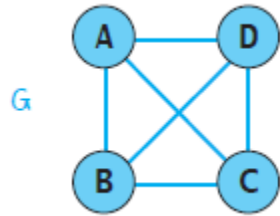


$V(G4) = \{A, B, C\}$
 $E(G4) = \{\langle A,B \rangle, \langle A,C \rangle, \langle B,C \rangle\}$

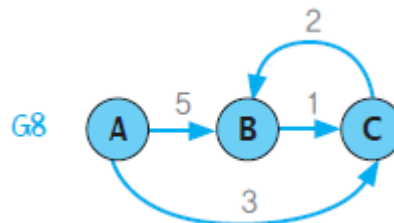
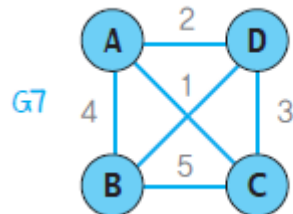
그래프의 종류



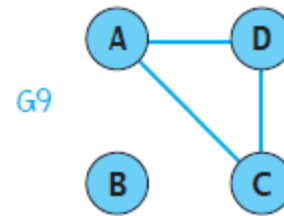
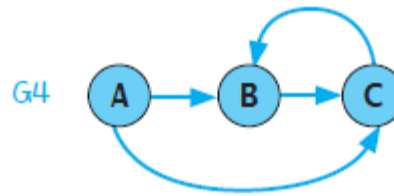
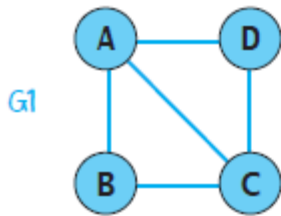
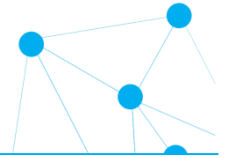
- 부분 그래프(subgraph)



- 가중치 그래프(weighted graph)



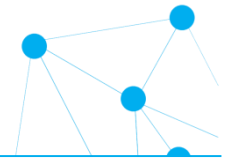
그래프의 용어



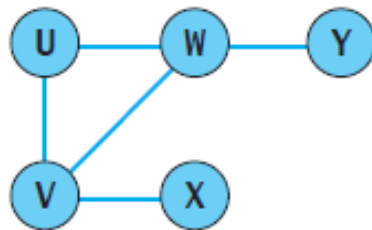
- 용어들

- 인접(adjacent) : 간선으로 연결된 두 정점은 '인접'
- 정점의 차수(degree) : 정점에 연결된 간선의 수
- 경로(path) : 간선을 따라갈 수 있는 길
- 단순(simple) 경로: 반복되는 간선이 없는 경로
- 사이클(cycle) : 시작 정점과 종료 정점이 같은 단순 경로
- 연결(connected) 그래프 : 모든 정점 사이에 경로가 존재
- 트리(tree) : 사이클을 가지지 않는 연결 그래프

8.2 그래프의 표현



- **인접 리스트:** 희소 그래프
- **인접 행렬:** 완전 그래프, 조밀 그래프 또는 정점 간의 인접 여부를 빨리 알아내야 하는 경우



(a) 무방향 그래프

	U	V	W	X	Y
U	0	1	1	0	0
V	1	0	1	1	0
W	1	1	0	0	1
X	0	1	0	0	0
Y	0	0	1	0	0

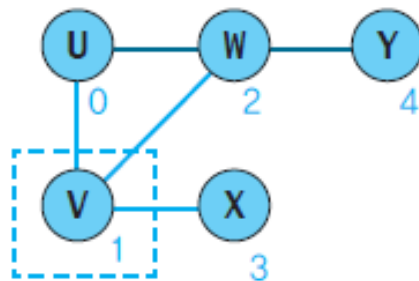
(b) 간선의 인접 행렬 표현

vtx = ['U', 'V', 'W', 'X', 'Y']

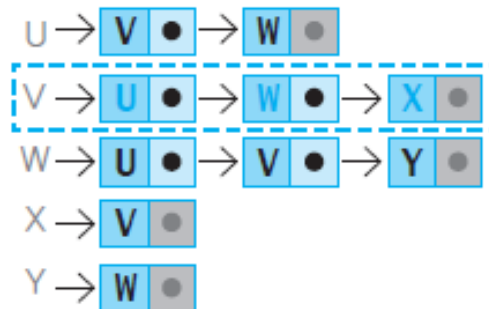
```
edge= [[0, 1, 1, 0, 0],
        [1, 0, 1, 1, 0],
        [1, 1, 0, 0, 1],
        [0, 1, 0, 0, 0],
        [0, 0, 1, 0, 0]]
```

(c) 파이썬 표현

항상 대칭



(a) 무방향 그래프



(b) 간선의 인접 리스트 표현

vtx = ['U', 'V', 'W', 'X', 'Y']

```
aList=[[1, 2],
        [0, 2, 3],
        [0, 1, 4],
        [1],
        [2]]
```

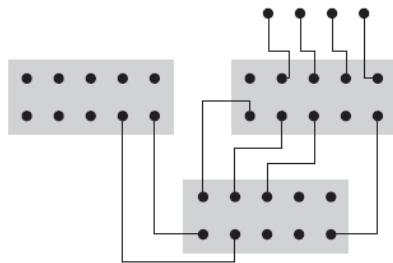
1번 정점(V)의
이웃은 0,2,3번
정점

(c) 파이썬 표현(파이썬 리스트 이용)

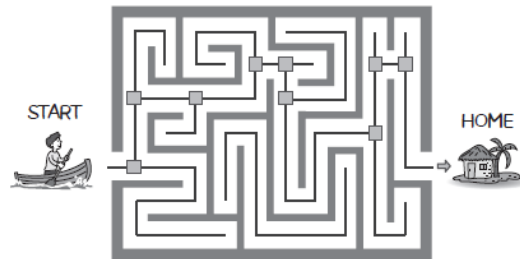
8.3 그래프 순회



- 그래프의 모든 정점을 한 번씩 방문하는 작업

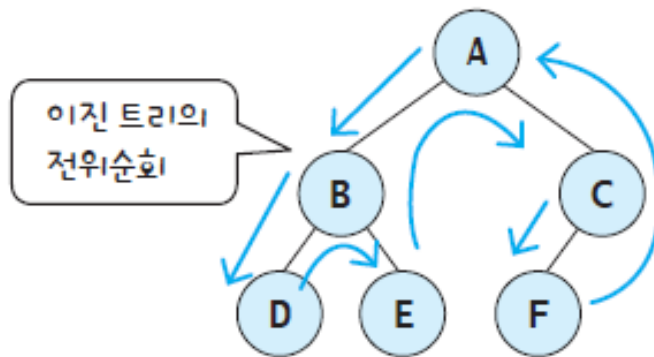


단자들 간의 연결성 검사

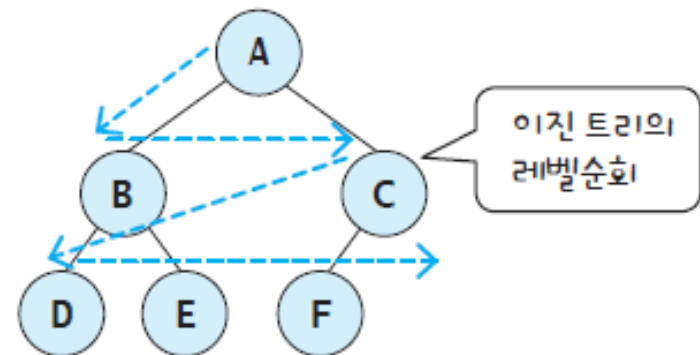


미로 탐색 문제

- 깊이 우선 탐색 / 너비 우선 탐색



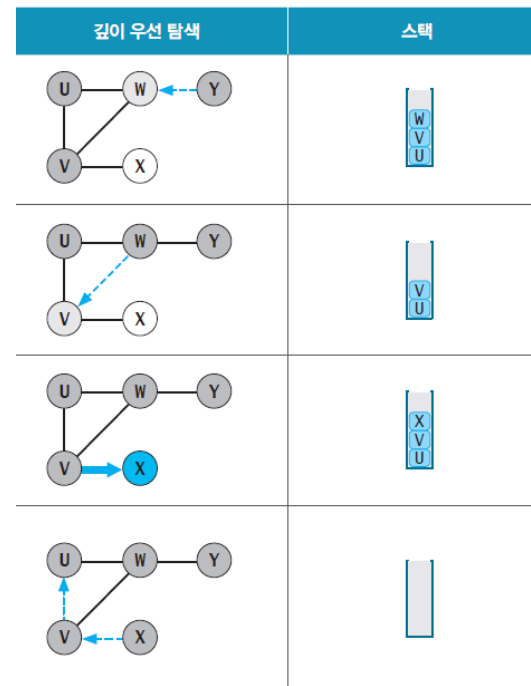
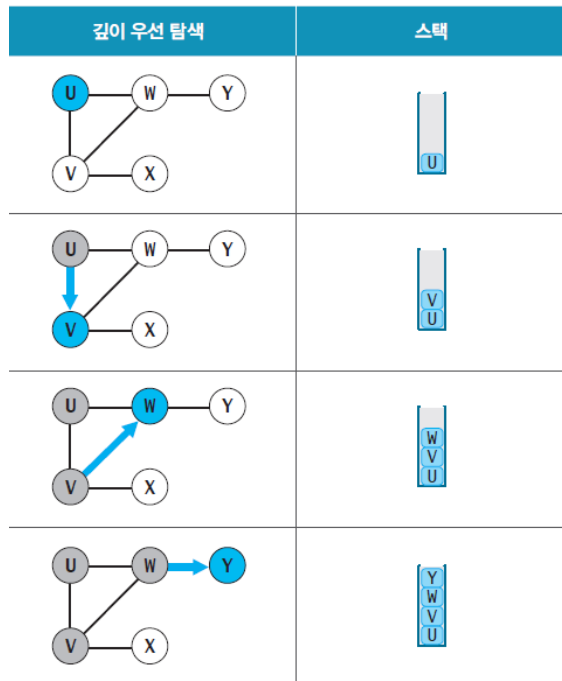
(a) 깊이 우선 탐색



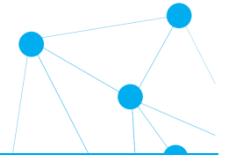
(b) 너비 우선 탐색

깊이 우선 탐색(DFS: Depth First Search): 스택

- 시작 정점에서 한 방향으로 갈 수 있는 곳까지 깊이 탐색을 진행하다가 더 이상 갈 곳이 없으면 가장 최근에 만났던 갈림길 정점으로 되돌아옴
- 갈림길로 돌아와서는 가 보지 않은 다른 방향의 간선으로 탐색을 다시 진행하고, 이 과정을 반복해 결국 모든 정점을 방문함
- 탐색 과정에서 여러 갈림길을 만나지만 **그 중에서 가장 최근에 만났던 갈림길로 되돌아와야 하므로** 이들을 후입선출의 구조의 **스택에 저장**



깊이 우선 탐색(인접 행렬 방식)



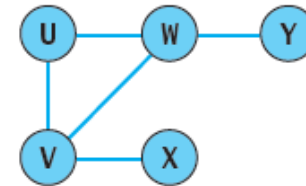
```
01: def DFS(vtx, adj, s, visited) :
02:     print(vtx[s], end=' ')
03:     visited[s] = True
04:
05:     for v in range(len(vtx)) :
06:         if adj[s][v] != 0 :
07:             if visited[v]==False:
08:                 DFS(vtx, adj, v, visited)
```

← 현재 정점 s는 방문 했으므로, 화면에 출력하고, visited를 True로 갱신.

← 방문하지 않은 이웃 정점 v가 있으면 그 정점을 시작으로 다시 DFS 호출

```
01: vtx = ['U', 'V', 'W', 'X', 'Y'] # 그림 8.9의 정점 리스트
02: edge= [[0, 1, 1, 0, 0], ... ] # 그림 8.9의 인접 행렬
03:
04: print('DFS(출발:U) : ', end='')
05: DFS(vtx, edge, 0, [False]*len(vtx))
06: print()
```

visited를 위해 길이가 정점 수와 같은 False 리스트를 만들

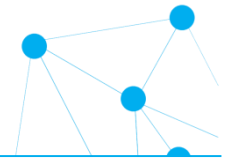


실행 결과

DFS(출발:U) : U V W Y X

시작 정점이 U인 경우와 DFS 정점 방문 순서

Note: DFS의 시간 복잡도

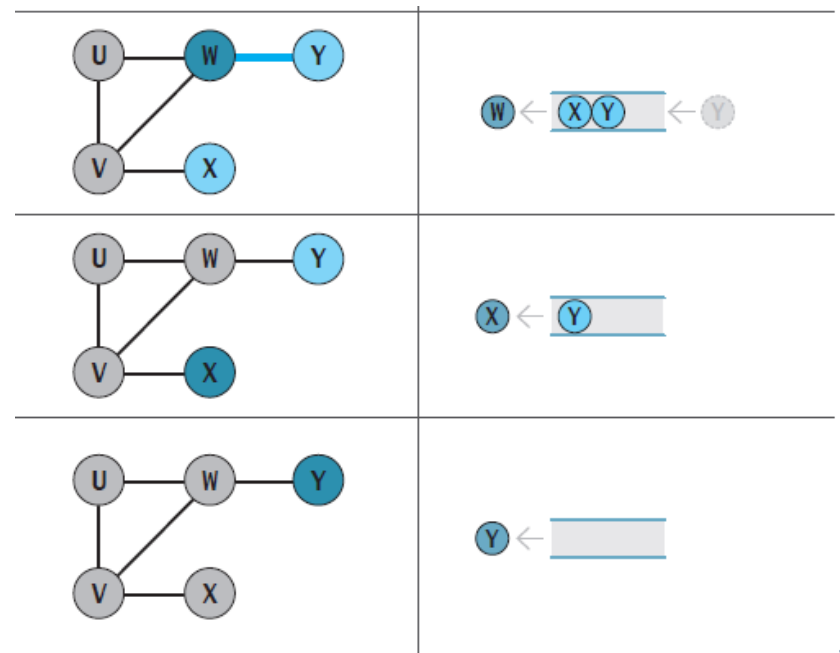
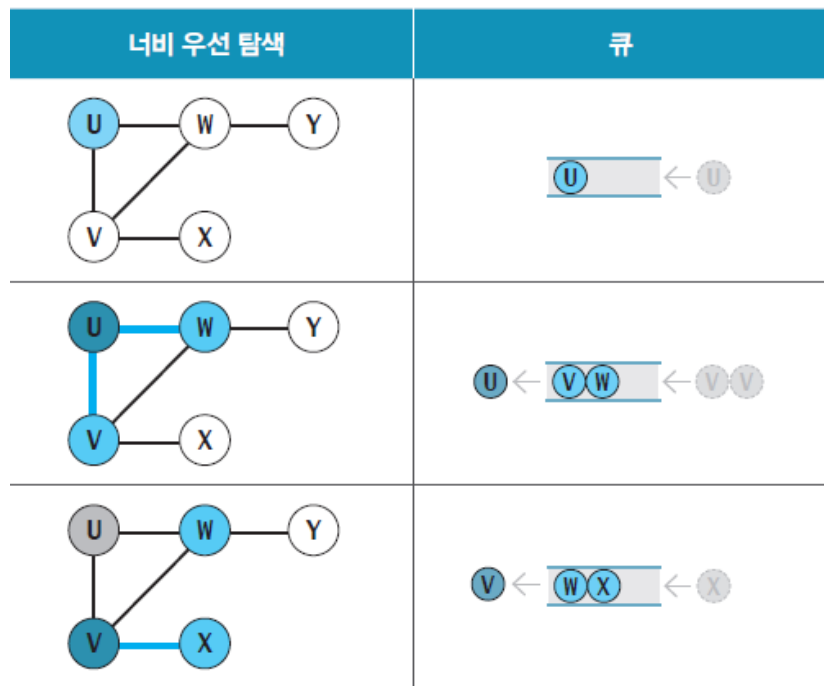


- DFS의 시간 복잡도는 각 정점을 한 번 방문하고, 각 간선을 한 번씩 확인하는 데 기반
- **인접 리스트**: 각 정점에 연결된 간선 만을 저장하여, 모든 정점과 간선을 탐색
- **인접 행렬**: 모든 정점 쌍에 대해 연결 여부를 확인

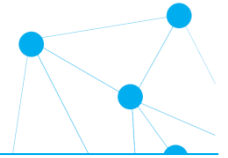
그래프 표현 방식	정점 수 (V)	간선 수 (E)	DFS의 시간 복잡도
인접 리스트	V	E	$O(V + E)$
인접 행렬	V	-	$O(V^2)$

너비 우선 탐색(BFS: Breadth First Search): 큐

- 거리가 0인 시작 정점으로부터 거리가 1인 모든 정점을 방문하고, 거리가 2인 모든 정점들, 거리가 3인 정점들의 순서로 방문을 진행
- 가까운 거리에 있는 정점들을 차례로 저장하고, 들어간 순서대로 꺼낼 수 있는 자료구조 큐 사용
- 큐에서 정점을 꺼낼 때마다 아직 방문하지 않은 모든 정점들을 큐에 삽입
- 탐색 과정은 큐가 공백 상태가 될 때까지 진행



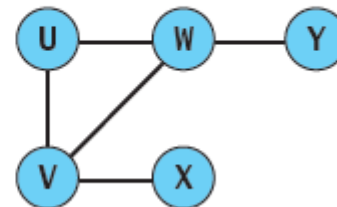
너비 우선 탐색(인접 리스트 방식)



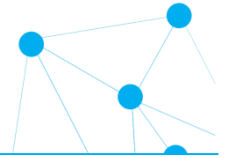
```
01: from queue import Queue          # queue 모듈의 Queue 사용
02: def BFS_AL(vtx, aList, s):
03:     n = len(vtx)                  # 그래프의 정점 수
04:     visited = [False]*n           # 방문 확인을 위한 리스트
05:     Q = Queue()
06:     Q.put(s)                       ← 큐를 만들고, 맨 처음에 시작 정점 s만 큐에 넣음.
07:     visited[s] = True              s는 "방문"했다고 표시.
08:     while not Q.empty():
09:         s = Q.get()
10:         print(vtx[s], end=' ')
11:         for v in aList[s]:
12:             if visited[v]==False:  ← 큐에서 정점 s를 꺼내고,
13:                 Q.put(v)           s의 인접 정점들 중에서
14:                 visited[v] = True  아직 방문하지 않은 정점들을 모두
                                     큐에 삽입하고, "방문"했다고 표시.
```

```
01: vtx = ['U', 'V', 'W', 'X', 'Y']
02: aList=[ [1, 2], ... ]
03: print('BFS_AL(출발:U): ', end='')
04: BFS_AL(vtx, aList, 0)
05: print()
```

그림 8.11(c)의 정점 리스트
그림 8.11(c)의 인접 리스트



Note: BFS의 시간 복잡도



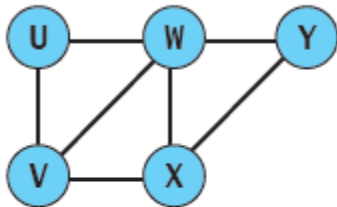
- BFS의 시간 복잡도는 각 정점을 한 번 방문하고, 각 간선을 한 번씩 확인하는 데 기반
- **인접 리스트**: 각 정점에 연결된 간선만을 저장하는 방식으로, BFS는 모든 정점과 그에 연결된 모든 간선을 탐색
- **인접 행렬**: 모든 정점 쌍에 대해 연결 여부를 저장하는 방식으로, BFS는 모든 정점에 대해 모든 정점을 탐색

그래프 표현 방식	정점 수 (V)	간선 수 (E)	BFS의 시간 복잡도
인접 리스트	V	E	$O(V + E)$
인접 행렬	V	-	$O(V^2)$

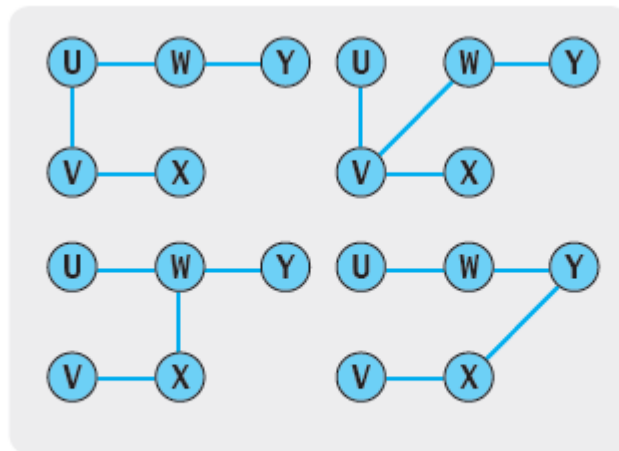
8.4 신장 트리(spanning tree)



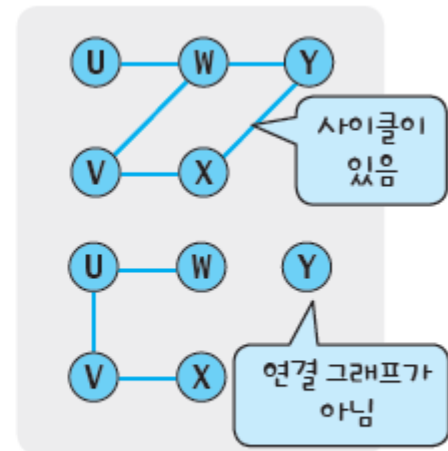
- 그래프 내 모든 정점을 포함하는 트리
 - 그래프의 모든 정점을 포함
 - 사이클이 없어야 함
 - 그래프의 모든 정점의 수가 n 일 경우 $n - 1$ 개의 간선
- 하나의 그래프에는 여러 개의 신장 트리가 존재
- 탐색 도중에 사용된 간선들만 모으면 신장 트리가 만들어짐
 - DFS, BFS



그래프 G

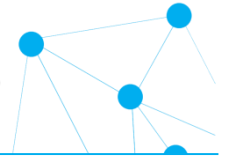


신장 트리의 예



신장 트리가 아닌 예

DFS를 이용한 신장트리(인접행렬 방식)



```
01: def ST_DFS(vtx, adj, s, visited) :
```

```
02:     visited[s] = True
```

```
03:     for v in range(len(vtx)) :
```

```
04:         if adj[s][v] != 0 :
```

```
05:             if visited[v]==False:
```

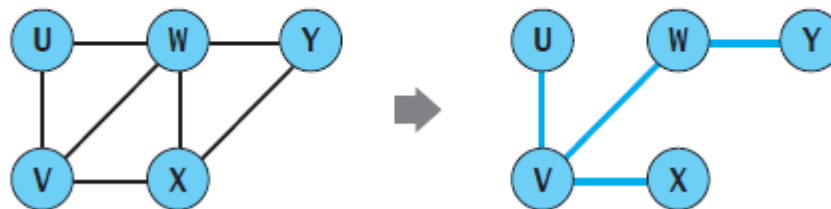
```
06:                 print("(" , vtx[s], vtx[v], ")", end=' ')
```

```
07:                 ST_DFS(vtx, adj, v, visited)
```

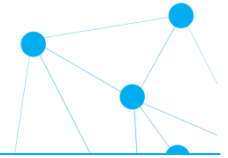
방문하지 않은 s의 이웃 정점 v가 있으면,
간선 (s,v)를 신장트리에 추가하고,
v를 시작으로 다시 깊이우선탐색 진행

실행 결과

ST_DFS_AM: (U V) (V W) (W Y) (V X)



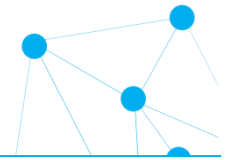
Note: 신장트리의 시간 복잡도



- DFS를 이용한 신장 트리 생성
- **인접 리스트**: 각 정점에 연결된 간선만을 저장하는 방식으로, 모든 정점과 그에 연결된 모든 간선을 탐색
- **인접 행렬**: 모든 정점 쌍에 대해 연결 여부를 저장하는 방식으로, 모든 정점을 순회하면서 연결 여부를 확인

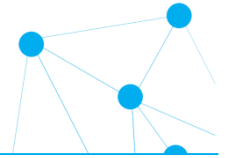
그래프 표현 방식	정점 수 (V)	간선 수 (E)	신장 트리의 시간 복잡도
인접 리스트	V	E	$O(V + E)$
인접 행렬	V	-	$O(V^2)$

Note: Greedy(탐욕) 알고리즘 설계



- Greedy 기법:
 - 문제를 해결할 때 **매 순간 가장 최선이라고** 생각되는 선택을 하여 전체 최적의 해를 찾는 전략
 - 각 단계에서의 선택이 이후의 선택에 영향을 미치지 않는 경우에 유효
 - 즉, 문제를 부분 문제로 나누고, 부분 문제를 해결함으로써 전체 문제의 최적 해를 구성할 수 있는 경우
- Greedy 기법의 최적 성능을 보장하는 조건
 - **Greedy 선택 속성 (Greedy Choice Property)**
 - 각 단계에서의 국소적 선택이 전체 최적 해의 일부로 구성
 - **최적 부분 구조 (Optimal Substructure)**
 - 문제의 최적 해가 부분 문제의 최적 해로 구성
- Greedy 기법 조건을 만족하지 경우에는 동적 계획법, 다른 최적화 기법을 사용

Greedy 기법과 그래프 자료구조



- Greedy 기법은 그래프 문제에서 자주 사용되며, **최소 신장 트리, 최단 경로, 네트워크 최대 플로우, 그래프 색칠** 등 문제에서 효과적
 - Greedy 선택 속성과 최적 부분 구조가 성립하므로, **최적 해나 근사 해**를 보장
- **최소 비용 문제** : 그래프의 모든 정점을 연결하면서 간선 가중치의 합이 최소가 되는 **최소 신장 트리**를 찾는 문제
 - Prim 알고리즘, Kruskal 알고리즘
- **최단 경로 문제**: 그래프에서 특정 정점에서 다른 정점까지의 최단 경로를 찾는 문제
 - Dijkstra 알고리즘

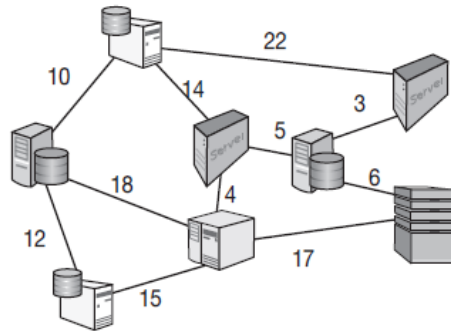
Dijkstra's Algorithm VS. Prim's Algorithm:



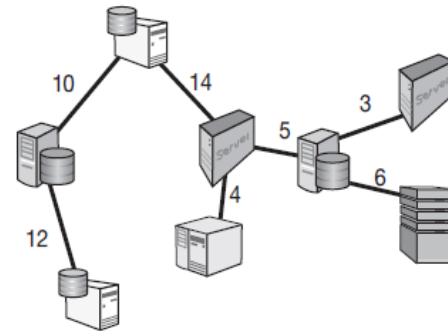
특징	다익스트라 알고리즘	프림 알고리즘
목적	특정 시작 노드에서 다른 모든 노드까지의 최단 경로를 찾음	연결된 그래프의 최소 비용 신장 트리(MST)를 찾음
문제 유형	단일 출발점 최단 경로 문제	최소 비용 신장 트리 문제
적용 분야	네트워크 라우팅, GPS 네비게이션, 경로 탐색 등	네트워크 설계, 회로 배치 등 최소 연결 비용이 필요한 분야
그래프 조건	비가중치 및 가중치가 음수가 아닌 그래프, 방향 그래프도 가능	무방향 연결 그래프로, 가중치가 양수이어야 함
알고리즘 접근 방식	탐욕적(Greedy) - 항상 현재 최단 경로를 선택하여 확장	탐욕적(Greedy) - MST에 연결된 가장 작은 비용의 간선을 선택
초기 설정	시작 노드의 거리를 0, 나머지 노드는 무한대로 초기화	임의의 시작 노드에서 MST를 확장하여 최소 간선을 추가
사용하는 자료 구조	최소 힙(우선순위 큐)을 사용하여 최소 거리를 가진 노드를 선택	최소 힙(우선순위 큐)을 사용하여 최소 비용의 간선을 선택
시간 복잡도 (최소 힙 사용 시)	$O(E \log V)$, V 는 정점 수, E 는 간선 수	$O(E \log V)$, V 는 정점 수, E 는 간선 수
출력 결과	시작 노드에서 모든 노드까지의 최단 경로	최소 비용 신장 트리를 구성하는 간선 집합
간선 완화(경신)	시작 노드로부터의 최단 경로에 따라 간선을 완화	MST에 추가할 최소 비용의 간선으로 간선 완화
제한 사항	가중치가 음수인 경우 사용 불가 (음수 가중치가 있을 경우 Bellman-Ford 알고리즘 사용)	무방향 그래프만 가능하며, 모든 노드가 연결되어 있어야 함

8.5 최소 비용 신장 트리 (Minimum Spanning Tree)

- 가중치 그래프의 여러 신장 트리 중에서 간선의 가중치 합이 최소인 트리
 - 통신망 구축: 모든 사이트가 연결되도록 하면서 비용을 최소화



(a) 사이트 사이의 연결 비용

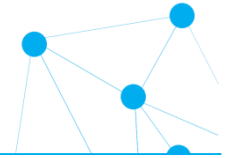


$$\text{가중치 합} = 12 + 10 + 14 + 4 + 5 + 3 + 6 = 54$$

(b) 최소 신장 트리의 예

- 도로망: 도시들을 모두 연결하면서 도로의 길이가 최소가 되도록 하는 문제
- 배관 작업, 전기 회로

프림(Prim) 알고리즘



- MST에 포함되지 않은 정점 v 에 대해, 현재까지 발견된 최소 비용 간선을 유지하고, 새로운 간선이 더 적은 비용일 경우 이를 업데이트
- 즉, 정점 u 와 정점 v 사이의 간선이 최소 비용이라면, 간선 완화 :
$$dist[v] = \min(dist[v], weight(u, v))$$
 - $dist[v]$: 정점 v 와 MST 사이의 현재 최소 비용 간선의 가중치
 - $weight(u, v)$: 정점 u 와 v 를 잇는 간선의 가중치
- 알고리즘 단계:
 - 모든 정점의 초기값을 무한대, 시작 정점의 거리를 0으로 설정
 - 아직 MST에 포함되지 않은 정점들 중에서 최소 비용 간선으로 연결된 정점을 선택하여 MST에 추가
 - 새로 추가된 정점 u 와 연결된 모든 인접 정점 v 에 대해, 위의 수식을 사용하여 **간선 완화 과정**을 수행
 - 이 과정을 반복하여, 모든 정점이 MST에 포함될 때까지 가장 낮은 비용의 간선을 차례로 추가

프림 알고리즘(1/3)



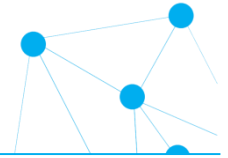
단계		그래프	최소신장트리(MST)																								
0	<p>맨 처음에 MST는 공백 트리이므로 selected는 모두 False입니다. dist[]는 시작 정점 A만 0이고 나머지는 ∞로 초기화됩니다.</p> <table> <tr><td></td><td>A</td><td>B</td><td>C</td><td>D</td><td>E</td><td>F</td><td>G</td></tr> <tr><td>selected</td><td>F</td><td>F</td><td>F</td><td>F</td><td>F</td><td>F</td><td>F</td></tr> <tr><td>dist</td><td>0</td><td>∞</td><td>∞</td><td>∞</td><td>∞</td><td>∞</td><td>∞</td></tr> </table>		A	B	C	D	E	F	G	selected	F	F	F	F	F	F	F	dist	0	∞	∞	∞	∞	∞	∞		
	A	B	C	D	E	F	G																				
selected	F	F	F	F	F	F	F																				
dist	0	∞	∞	∞	∞	∞	∞																				
1	<p>dist가 최소인 A를 MST에 넣습니다 (selected 갱신). 이제 A의 인접 정점들의 dist를 갱신해야 합니다. 만약, A와 인접 정점 사이의 간선의 가중치가 기존의 dist보다 작다면, 가중치를 갱신해야 합니다. B와 D가 25, 12로 변경됩니다.</p> <table> <tr><td></td><td>A</td><td>B</td><td>C</td><td>D</td><td>E</td><td>F</td><td>G</td></tr> <tr><td>selected</td><td>T</td><td>F</td><td>F</td><td>F</td><td>F</td><td>F</td><td>F</td></tr> <tr><td>dist</td><td>0</td><td>25</td><td>∞</td><td>12</td><td>∞</td><td>∞</td><td>∞</td></tr> </table>		A	B	C	D	E	F	G	selected	T	F	F	F	F	F	F	dist	0	25	∞	12	∞	∞	∞		
	A	B	C	D	E	F	G																				
selected	T	F	F	F	F	F	F																				
dist	0	25	∞	12	∞	∞	∞																				
2	<p>dist가 최소인 D를 선택하고, 정점 D와 간선 (A, D)를 MST에 넣습니다. E와 F가 아직 선택되지 않은 인접 정점이고, dist를 기존과 비교해 17과 37로 갱신합니다.</p> <table> <tr><td></td><td>A</td><td>B</td><td>C</td><td>D</td><td>E</td><td>F</td><td>G</td></tr> <tr><td>selected</td><td>T</td><td>F</td><td>F</td><td>T</td><td>F</td><td>F</td><td>F</td></tr> <tr><td>dist</td><td>0</td><td>25</td><td>∞</td><td>12</td><td>17</td><td>37</td><td>∞</td></tr> </table>		A	B	C	D	E	F	G	selected	T	F	F	T	F	F	F	dist	0	25	∞	12	17	37	∞		
	A	B	C	D	E	F	G																				
selected	T	F	F	T	F	F	F																				
dist	0	25	∞	12	17	37	∞																				

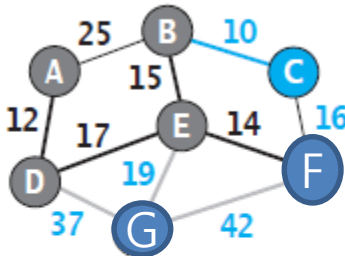
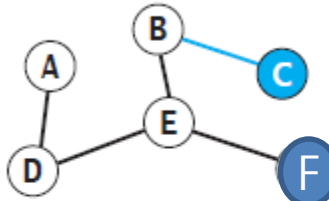
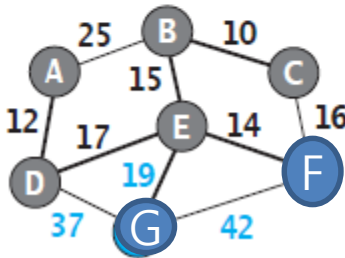
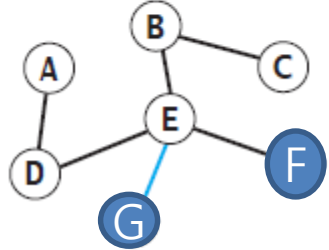
프림 알고리즘(2/3)



3	<p>dist가 최소인 E가 선택되고, E와 간선 (D, E)를 MST에 넣습니다. B, F, G가 아직 선택되지 않은 인접 정점인데, 기존 dist와 비교해 이들을 각각 15, 19, 14로 갱신합니다.</p> <table> <tr> <th></th><th>A</th><th>B</th><th>C</th><th>D</th><th>E</th><th>F</th><th>G</th></tr> <tr> <td>selected</td><td>T</td><td>F</td><td>F</td><td>T</td><td>T</td><td>F</td><td>F</td></tr> <tr> <td>dist</td><td>0</td><td>15</td><td>∞</td><td>12</td><td>17</td><td>19</td><td>14</td></tr> </table>		A	B	C	D	E	F	G	selected	T	F	F	T	T	F	F	dist	0	15	∞	12	17	19	14		
	A	B	C	D	E	F	G																				
selected	T	F	F	T	T	F	F																				
dist	0	15	∞	12	17	19	14																				
4	<p>dist가 최소인 G가 선택되고, G와 간선 (E, G)를 MST에 넣습니다. C, F가 아직 선택되지 않은 인접 정점인데, 기존 dist와 비교해 C를 16으로 갱신합니다.</p> <table> <tr> <th></th><th>A</th><th>B</th><th>C</th><th>D</th><th>E</th><th>F</th><th>G</th></tr> <tr> <td>selected</td><td>T</td><td>F</td><td>F</td><td>T</td><td>T</td><td>F</td><td>T</td></tr> <tr> <td>dist</td><td>0</td><td>15</td><td>16</td><td>12</td><td>17</td><td>19</td><td>14</td></tr> </table>		A	B	C	D	E	F	G	selected	T	F	F	T	T	F	T	dist	0	15	16	12	17	19	14		
	A	B	C	D	E	F	G																				
selected	T	F	F	T	T	F	T																				
dist	0	15	16	12	17	19	14																				
5	<p>dist가 최소인 B가 선택되고, B와 간선 (B, E)를 MST에 넣습니다. C가 인접 정점인데, 기존 dist보다 (B,C)의 가중치가 작아 C를 10으로 갱신합니다.</p> <table> <tr> <th></th><th>A</th><th>B</th><th>C</th><th>D</th><th>E</th><th>F</th><th>G</th></tr> <tr> <td>selected</td><td>T</td><td>T</td><td>F</td><td>T</td><td>T</td><td>F</td><td>T</td></tr> <tr> <td>dist</td><td>0</td><td>15</td><td>10</td><td>12</td><td>17</td><td>19</td><td>14</td></tr> </table>		A	B	C	D	E	F	G	selected	T	T	F	T	T	F	T	dist	0	15	10	12	17	19	14		
	A	B	C	D	E	F	G																				
selected	T	T	F	T	T	F	T																				
dist	0	15	10	12	17	19	14																				

프림 알고리즘(3/3)



단계	그래프	최소신장트리(MST)																								
<div data-bbox="227 529 249 554">6</div> <div data-bbox="293 434 803 541"> dist가 최소인 C가 선택되고, C와 간선 (B, C)를 MST에 넣습니다. 선택되지 않은 인접 정점은 없습니다. </div> <table data-bbox="293 569 797 656"> <tr> <td></td><td>A</td><td>B</td><td>C</td><td>D</td><td>E</td><td>F</td><td>G</td></tr> <tr> <td>selected</td><td>T</td><td>T</td><td>T</td><td>T</td><td>T</td><td>F</td><td>T</td></tr> <tr> <td>dist</td><td>0</td><td>15</td><td>10</td><td>12</td><td>17</td><td>19</td><td>14</td></tr> </table>		A	B	C	D	E	F	G	selected	T	T	T	T	T	F	T	dist	0	15	10	12	17	19	14		
	A	B	C	D	E	F	G																			
selected	T	T	T	T	T	F	T																			
dist	0	15	10	12	17	19	14																			
<div data-bbox="227 886 249 909">7</div> <div data-bbox="293 791 803 898"> 마지막으로 dist가 최소인 F를 간선 (E, F)와 함께 MST에 넣습니다. MST가 완성되었습니다. </div> <table data-bbox="293 925 797 1012"> <tr> <td></td><td>A</td><td>B</td><td>C</td><td>D</td><td>E</td><td>F</td><td>G</td></tr> <tr> <td>selected</td><td>T</td><td>T</td><td>T</td><td>T</td><td>T</td><td>T</td><td>T</td></tr> <tr> <td>dist</td><td>0</td><td>15</td><td>10</td><td>12</td><td>17</td><td>19</td><td>14</td></tr> </table>		A	B	C	D	E	F	G	selected	T	T	T	T	T	T	T	dist	0	15	10	12	17	19	14		
	A	B	C	D	E	F	G																			
selected	T	T	T	T	T	T	T																			
dist	0	15	10	12	17	19	14																			

프림 알고리즘

시간 복잡도는 $O(n^2)$



```
11: def MSTPrim(vertex, adj) :
12:     n = len(vertex)
13:     dist = [INF] * n
14:     dist[0] = 0
15:     selected = [False] * n
16:
17:     for _ in range(n) :
18:         u = getMinVertex(dist, selected)
19:         selected[u] = True
20:         print(vertex[u], end=' ')
21:         for v in range(n) :
22:             # 간선 (u,v)가 있고, v ∉ MST이면
23:             if adj[u][v] != INF and not selected[v] :
24:                 if adj[u][v] < dist[v] : # (u,v)가 dist[v]보다 작으면
25:                     dist[v] = adj[u][v] # dist[v] 갱신
26:
27:         print(':', ' ', dist) # 중간 결과 출력
```

인접 행렬

정점 리스트

dist와 selected 배열 초기화.
dist는 시작 정점 0을 제외하고 모두 INF를 갖고,
selected는 모두 False

n개의 정점을 MST에 추가하면 종료됨

최소 dist 정점 u를 찾아 화면에
출력하고 "방문"표시를 함

간선 (u,v)가 있고, v ∉ MST이면

인접 정점 v의 dist값 갱신.
이전 값보다 작은 경우에만
갱신함.

프림 알고리즘



- MST에 포함되지 않은 최소 dist의 정점 찾기

```
01: INF = 999
02: def getMinVertex(dist, selected) :
03:     minv = 0
04:     mindist = INF
05:     for v in range(len(dist)) :
06:         if selected[v]==False and dist[v]<mindist :
07:             mindist = dist[v]
08:             minv = v
09:     return minv
```

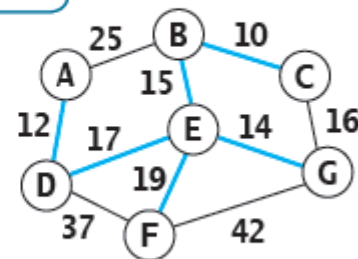
MST에 포함되지 않은 정점 중에서 최소 dist를 갖는 정점의 인덱스 minv를 구함

실행 결과

MS 정점이 선택되는 과정 rithm

```
A : [0, 25, 999, 12, 999, 999, 999]
D : [0, 25, 999, 12, 17, 37, 999]
E : [0, 15, 999, 12, 17, 19, 14]
G : [0, 15, 16, 12, 17, 19, 14]
B : [0, 15, 10, 12, 17, 19, 14]
C : [0, 15, 10, 12, 17, 19, 14]
F : [0, 15, 10, 12, 17, 19, 14]
```

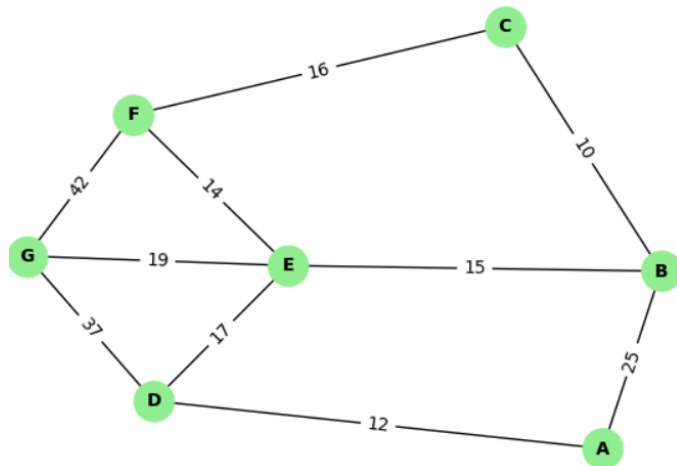
dist 배열 변화



결과 수정:

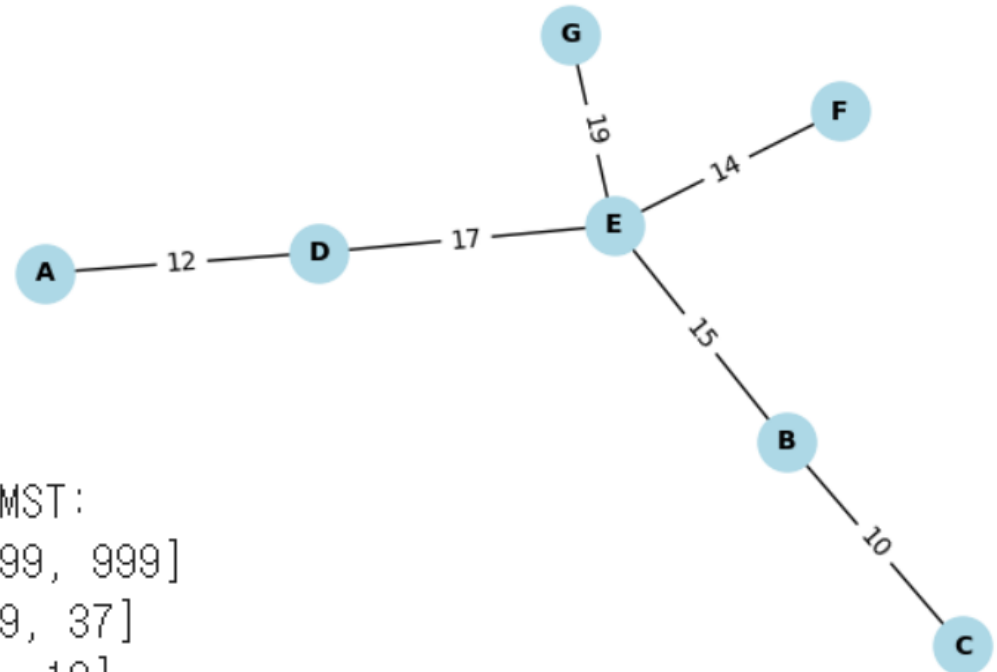


Initial Graph with Weights



최소 비용: 87

Prim's MST Graph



Prim 알고리즘에 의해 생성된 MST:

A : [0, 25, 999, 12, 999, 999, 999]

D : [0, 25, 999, 12, 17, 999, 37]

E : [0, 15, 999, 12, 17, 14, 19]

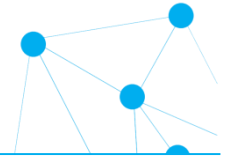
F : [0, 15, 16, 12, 17, 14, 19]

B : [0, 15, 10, 12, 17, 14, 19]

C : [0, 15, 10, 12, 17, 14, 19]

G : [0, 15, 10, 12, 17, 14, 19]

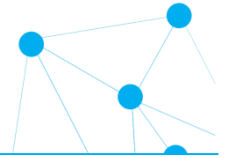
Note: Prim 알고리즘의 시간 복잡도



- **배열:**
 - **인접 행렬:** 모든 정점 쌍을 탐색하므로 시간 복잡도는 $O(V^2)$
 - **인접 리스트:** 최소 비용의 간선을 찾으므로, V 개의 정점을 반복하여 확인하는 과정이므로 $O(V^2)$
- **최소 힙:**
 - **인접 리스트** 방식에서 최소 힙을 사용하면 최솟값을 $O(\log_2 V)$ 시간에 찾을 수 있어 효율적
 - 전체 시간 복잡도는 $O(E \log_2 V)$ 로, 그래프가 희소할 때(즉, E 가 V^2 보다 작을 때) 더 효율적

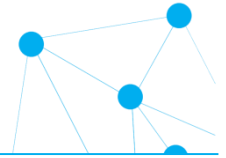
자료 구조	그래프 표현 방식	정점 수 (V)	간선 수 (E)	시간 복잡도
배열	인접 행렬	V	-	$O(V^2)$
배열	인접 리스트	V	E	$O(V^2)$
최소 힙	인접 리스트	V	E	$O(E \log V)$

Note: 최소 힙(Min-Heap)

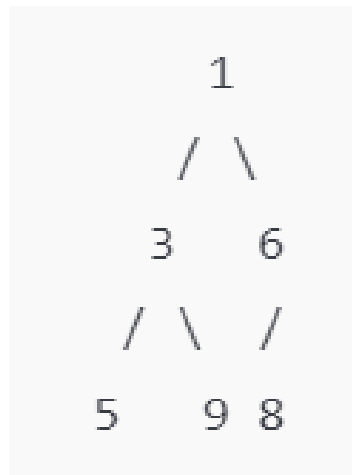


- 최소 힙(Min-Heap) :
 - **완전 이진 트리**: 트리의 모든 레벨이 가득 차 있으며, 마지막 레벨의 노드는 왼쪽부터 차례로 채워짐.
 - **Heap 조건**: 부모 노드의 값이 항상 자식 노드의 값보다 작거나 같은 특성을 가지며, 루트 노드가 항상 최솟값.
- 특징:
 - 트리의 최상단(루트)에 항상 최소값이 위치
 - 이진 트리 형태를 유지하며
 - 노드가 삽입되거나 삭제될 때 재정렬
- 사용 예시:
 - Prim, Dijkstra 알고리즘 등 최적화 문제에서 사용

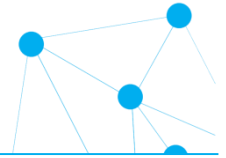
최소 힙 구조 표현



- 트리 표현
 - 루트 노드가 최소값이며, 왼쪽과 오른쪽 서브트리도 각각 최소 힙 조건을 만족
 - 예: [1, 3, 6, 5, 9, 8]
- 배열 표현:
 - 부모 노드: 인덱스 i 에서 부모는 $(i - 1) // 2$
 - 왼쪽 자식: 인덱스 i 에서 왼쪽 자식은 $2 * i + 1$
 - 오른쪽 자식: 인덱스 i 에서 오른쪽 자식은 $2 * i + 2$

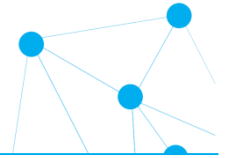


하향 Heapify



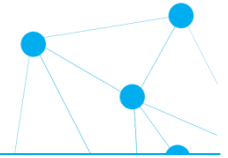
- 하향 Heapify
 - 요소가 올바른 위치에 도달할 때까지 **아래로 이동**시키며 힙 특성을 유지
- 과정:
 - 자식 노드와 비교
 - 요소가 자식보다 크면 **더 작은 자식과 위치를 교환**
 - 이 과정을 반복하여 힙 특성이 만족될 때까지 **아래로 이동**
- 시간 복잡도:
 - 이는 요소가 이동할 수 있는 최대 거리가 트리의 높이에 비례 $O(\log_2 V)$

상향 Heapify



- 상향 Heapify
 - 요소가 올바른 위치에 도달할 때까지 **위로 이동**하며 힙 특성을 유지하는 과정
 - 이를 통해 부모 노드보다 작은 값은 위로 올라가며 Min-Heap 조건을 만족
- 과정:
 1. 부모 노드와 비교
 - 새 요소가 부모보다 작으면 둘의 위치를 교환
 2. 이 과정을 반복하여 힙 특성이 만족될 때까지 **위로 이동**
- 시간 복잡도:
 - 이는 요소가 이동할 수 있는 최대 거리가 트리의 높이에 비례 $O(\log_2 V)$

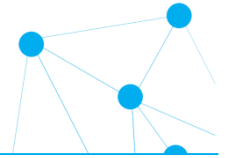
배열을 이용하여 최소 힙 생성



- 주어진 배열에서 Min-Heap을 생성하는 과정
 - 마지막 비단말 노드부터 시작하여 **하향 Heapify**를 수행
 - 각 레벨에서 하향 Heapify를 반복하여 Min-Heap 구조를 형성
- 시간 복잡도:
 - 각 노드에서 하향 Heapify를 한 번만 수행하므로 $O(n)$
- 예: 배열 [9, 5, 6, 2, 3]을 최소 힙으로 변환하는 과정

단계	설명	배열 상태
초기	초기 배열 상태	[9, 5, 6, 2, 3]
1	마지막 비단말 노드인 5에서 하향 Heapify 적용	[9, 5, 6, 2, 3]
	5과 자식 2를 비교하여 교환 ($5 > 2$)	[9, 2, 6, 5, 3]
2	루트 노드 9에서 하향 Heapify 적용	[9, 2, 6, 5, 3]
	9과 자식 2를 비교하여 교환 ($9 > 2$)	[2, 9, 6, 5, 3]
	9과 자식 3을 비교하여 교환 ($9 > 3$)	[2, 3, 6, 5, 9]

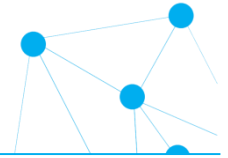
최소 힙의 삽입 연산



- Min-Heap에 새로운 요소를 삽입하는 과정
 - 요소를 트리의 끝에 추가하여 완전 이진 트리의 형태를 유지
 - **상향 Heapify**: 추가한 요소와 부모 노드를 비교하며, 부모 노드보다 작으면 서로 위치를 교환. 이 과정을 Min-Heap 조건을 만족할 때까지 반복
- 시간 복잡도 : 트리의 높이에 따라 이동해야 하므로 최대 $O(\log_2 n)$
- 예: [1, 5, 6, 9] 최소 힙에 3을 삽입

단계	설명	배열 상태
1	3을 트리의 끝에 추가	[1, 5, 6, 9, 3]
2	3과 부모 5를 비교하여 $3 < 5$ 이므로 교환	[1, 3, 6, 9, 5]

최소 힙의 삭제 연산



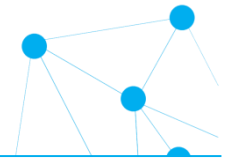
- Min-Heap에서 루트 노드(최솟값)를 삭제하는 과정
 - 루트 노드를 제거하고, 트리의 **마지막 요소를 루트 위치에 배치**
 - **하향 Heapify**: 루트 요소와 자식 노드를 비교하여 더 작은 자식 노드와 위치를 교환. 이 과정을 Min-Heap 조건이 만족될 때까지 반복
- 시간 복잡도 : 요소가 트리의 마지막 레벨까지 이동할 수 있어 최대 $O(\log_2 V)$ 시간이 소요
- 예: 최소 힙 [1, 3, 6, 9, 5] 에서 최소값 1을 삭제

단계	설명	배열 상태
1	1을 삭제하고, 마지막 요소 5를 루트로 이동	[5, 3, 6, 9]
2	5와 자식 노드 3을 비교하여 $3 < 5$ 이므로 교환	[3, 5, 6, 9]

Note: Dijkstra Algorithm과 Greedy 접근

- Dijkstra's Algorithm: 특정 시작 정점에서 다른 모든 정점까지의 **최단 경로**를 찾기 위한 알고리즘
- Greedy 접근을 사용하여 가장 짧은 경로를 계산
 - 현재 상황에서 가장 짧은 거리에 있는 정점을 선택하여 그 정점에서 다른 정점까지의 경로를 업데이트
 - 각 단계에서 최적의 선택을 반복적으로 수행하여 전체 최단 경로를 구하는 것
- 간선의 가중치가 **음수가 아닌 경우에만** 사용이 가능
- 시간 복잡도 :
 - 최소 힙을 사용하지 않으면 최단 거리 정점을 찾는 데 매번 $O(V)$ 이 걸리므로 $O(V^2)$
 - **최소 힙을 사용하면** : $O(\log_2 V)$ 로 줄일 수 있으며, 결과적으로 전체 시간 복잡도는 $O(E \log V)$

다익스트라(Dijkstra) 알고리즘



- **거리 테이블:** 각 단계에서 각 정점까지의 최단 거리
- **최소 힙:** 아직 방문하지 않은 정점들의 거리와 해당 정점 정보가 저장
- **거리 갱신 공식:** 현재 정점 u 와 인접한 정점 v 에 대해, 최단 거리를 갱신
 - $\text{dist}[u]$: 정점 u 까지의 현재 최단 거리
 - $\text{dist}[v]$: 정점 v 까지의 현재 최단 거리 (갱신할 값)
 - $\text{weight}(u,v)$: 정점 u 와 v 사이의 간선 가중치

• 알고리즘:

$$\text{dist}[v] = \min(\text{dist}[v], \text{dist}[u] + \text{weight}(u, v))$$

1. 거리 및 최소 힙 초기화:

- 거리 테이블을 시작 정점의 거리는 0, 나머지 정점은 INF로 설정
- 최소 힙에 시작 정점과 거리를 (거리, 정점) 형태로 추가

2. 최소 힙에서 정점 처리: 최소 힙이 빌 때까지 다음 과정을 반복

1. **최소 거리 정점 추출:** 최소힙에서 현재 최단 거리를 가진 정점을 추출

2. **간선 완화(업데이트):** 추출된 정점의 이웃 정점에 대해 업데이트

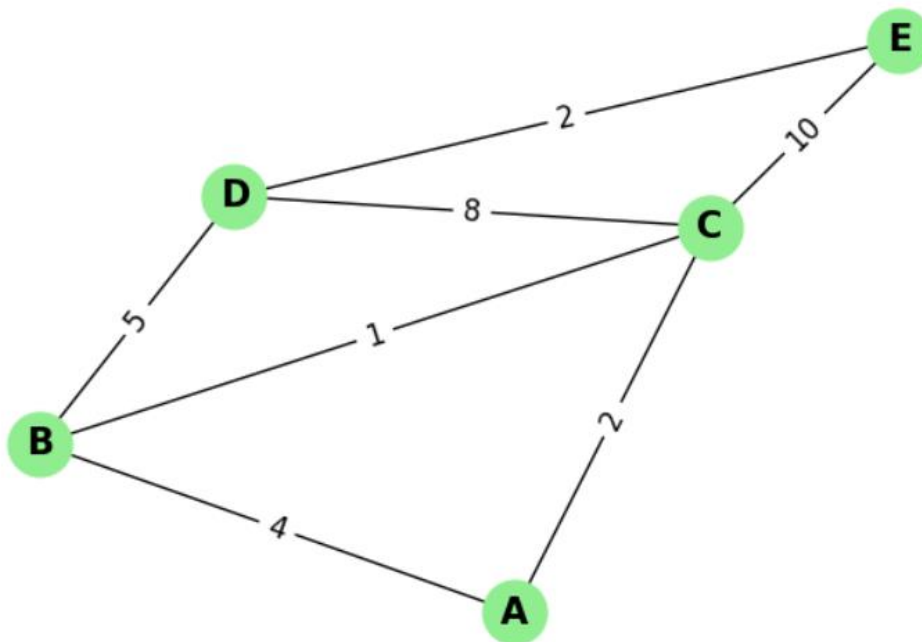
- 이웃 정점까지 가는 새로운 경로 거리를 계산
- 새로운 경로 거리가 현재 저장된 거리보다 짧으면, 이웃 정점의 거리를 갱신하고 최소 힙에 갱신된 거리로 삽입

예: 다익스트라 알고리즘

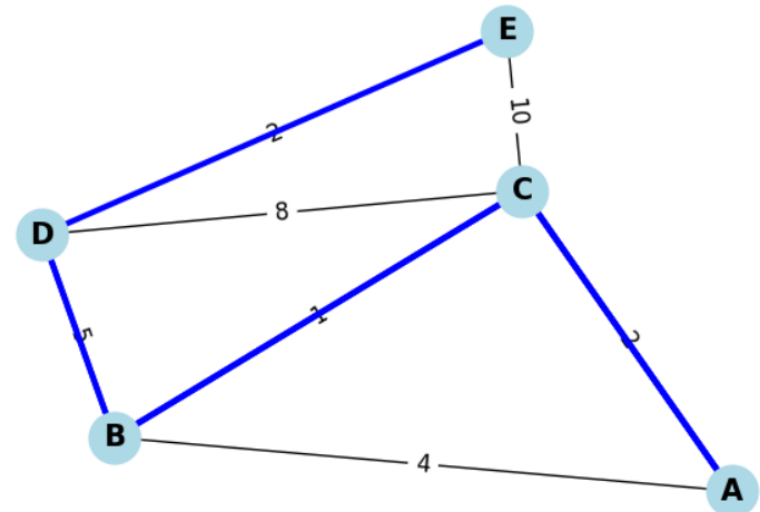


- 정점 A, B, C, D, E로 이루어진 그래프에서 A에서 출발하여 모든 정점까지의 최단 거리를 구하기

Initial Graph with Weights



Dijkstra's Algorithm - Shortest Paths from Node A



단계	정점	거리 테이블	최소 힙	설명
1	A	{'A': 0, 'B': ∞ , 'C': ∞ , 'D': ∞ , 'E': ∞ }	[]	정점 A 처리 시작
2	A	{'A': 0, 'B': 4, 'C': ∞ , 'D': ∞ , 'E': ∞ }	[(4, 'B')]	A에서 B의 거리 갱신 (새 거리: 4)
3	A	{'A': 0, 'B': 4, 'C': 2, 'D': ∞ , 'E': ∞ }	[(2, 'C'), (4, 'B')]	A에서 C의 거리 갱신 (새 거리: 2)
4	C	{'A': 0, 'B': 4, 'C': 2, 'D': ∞ , 'E': ∞ }	[(4, 'B')]	정점 C 처리 시작
5	C	{'A': 0, 'B': 3, 'C': 2, 'D': ∞ , 'E': ∞ }	[(3, 'B'), (4, 'B')]	C에서 B의 거리 갱신 (새 거리: 3)
6	C	{'A': 0, 'B': 3, 'C': 2, 'D': 10, 'E': ∞ }	[(3, 'B'), (4, 'B'), (10, 'D')]	C에서 D의 거리 갱신 (새 거리: 10)
7	C	{'A': 0, 'B': 3, 'C': 2, 'D': 10, 'E': 12}	[(3, 'B'), (4, 'B'), (10, 'D'), (12, 'E')]	C에서 E의 거리 갱신 (새 거리: 12)

단계	정점	거리 테이블	최소 힙	설명
8	B	{'A': 0, 'B': 3, 'C': 2, 'D': 10, 'E': 12}	[(3, 'B'), (4, 'B'), (10, 'D'), (12, 'E')]	정점 B 처리 시작
9	B	{'A': 0, 'B': 3, 'C': 2, 'D': 8, 'E': 12}	[(4, 'B'), (8, 'D'), (10, 'D'), (12, 'E')]	B에서 D의 거리 갱신 (새 거리: 8)
10	B	{'A': 0, 'B': 3, 'C': 2, 'D': 8, 'E': 12}	[(8, 'D'), (12, 'E'), (10, 'D')]	정점 B 처리 시작
11	D	{'A': 0, 'B': 3, 'C': 2, 'D': 8, 'E': 12}	[(10, 'D'), (12, 'E')]	정점 D 처리 시작
12	D	{'A': 0, 'B': 3, 'C': 2, 'D': 8, 'E': 10}	[(10, 'D'), (12, 'E'), (10, 'E')]	D에서 E의 거리 갱신 (새 거리: 10)
13	D	{'A': 0, 'B': 3, 'C': 2, 'D': 8, 'E': 10}	[(10, 'E'), (12, 'E')]	정점 D 처리 시작
14	E	{'A': 0, 'B': 3, 'C': 2, 'D': 8, 'E': 10}	[(12, 'E')]	정점 E 처리 시작
15	E	{'A': 0, 'B': 3, 'C': 2, 'D': 8, 'E': 10}	[]	정점 E 처리 시작