Chapter 4
그리디 알고리즘

차례

그리디 (Greedy) 알고리즘

- 그리디 알고리즘은 최적화 문제를 해결하는 알고리즘
  - 최적화 (optimization) 문제

    가능한 해들 중에서 가장 좋은 (최대 또는 최소) 해를 찾는 문제

- 욕심쟁이 방법, 탐욕적 방법, 탐욕 알고리즘 등으로 불림

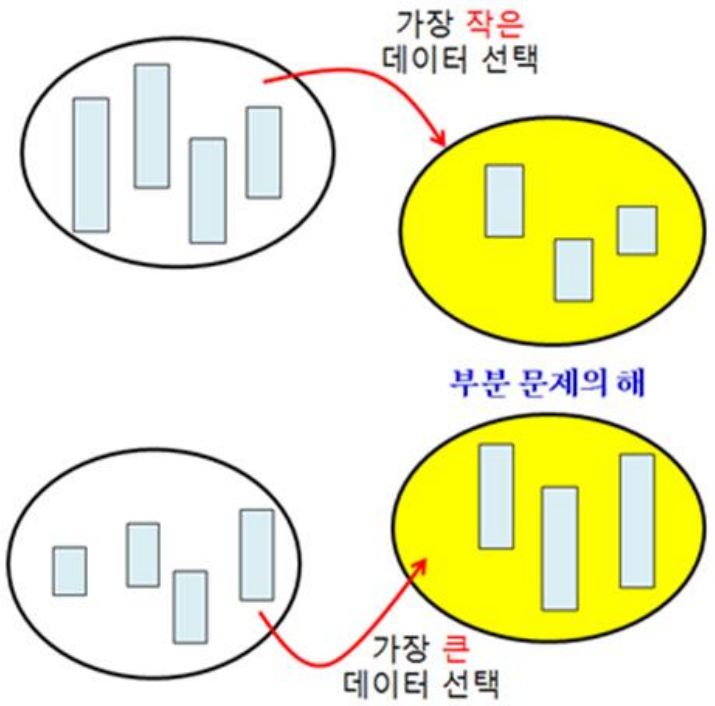
- 그리디 알고리즘은 (입력) 데이터 간의 관계를 고려하지 않고 수행 과정에서 '욕심내어' 최소값 또는 최대값을 가진 데이터를 선택
  - 이러한 선택을 '근시안적'인 선택이라고 말하기도 함

그리디 (Greedy) 알고리즘

- 그리디 알고리즘은 근시안적인 선택으로 부분적인 최적해를 찾고, 이들을 모아서(축적하여) 문제의 최적해를 얻는다.

최댓값 찾는 문제

최솟값 찾는 문제



그리디 (Greedy) 알고리즘

- 그리디 알고리즘은 일단 한 번 선택하면, 이를 절대로 번복하지 않는다.
  - 즉, 선택한 데이터를 버리고 다른 것을 취하지 않는다.

- 이러한 특성 때문에 대부분의 그리디 알고리즘들은 매우 단순하며, 또한 제한적인 문제만이 그리디 알고리즘으로 해결된다.

4.1 동전 거스름돈 문제

- 동전 거스름돈 (Coin Change) 문제를 해결하는 가장 간단하고 효율적인 방법
  - 남은 액수를 초과하지 않는 조건하에 '욕심내어' 가장 큰 액면의 동전을 취하는 것

- 동전 거스름돈 문제의 최소 동전 수를 찾는 그리디 알고리즘
  - 동전의 액면은 500원, 100원, 50원, 10원, 1원

알고리즘

CoinChange

입력: 거스름돈 액수 W

출력: 거스름돈 액수에 대한 최소 동전 수

1. change=W, n500=n100=n50=n10=n1=0
   // n500, n100, n50, n10, n1은 각각의 동전 카운트
2. `while` (change ≥ 500)
   change = change-500, n500++ // 500원 동전 수 1 증가
3. `while` (change ≥ 100)
   change = change-100, n100++ // 100원 동전 수 1 증가
4. `while` (change ≥ 50)
   change = change-50, n50++ // 50원 동전 수 1 증가
5. `while` (change ≥ 10)
   change = change-10, n10++ // 10원 동전 수 1 증가
6. `while` (change ≥ 1)
   change = change-1, n1++ // 1원 동전 수 1 증가
7. `return` (n500+n100+n50+n10+n1) // 총 동전 수 리턴

CoinChange 알고리즘

- 그리디 알고리즘의 근시안적인 특성
  - CoinChange 알고리즘은 남아있는 거스름돈인 change에 대해 가장 높은 액면의 동전을 거스르며,
  - 500원 동전을 처리하는 line 2에서는 100원, 50원, 10원, 1원 동전을 몇 개씩 거슬러 주어야 할 것인지에 대해서는 전혀 고려하지 않는다.

수행 과정

760원의 거스름돈에 대해

CoinChange 알고리즘의 문제점

- 한국은행에서 160원 동전을 추가로 발행한다면, CoinChange 알고리즘이 항상 최소 동전 수를 계산할 수 있을까?

  - 거스름돈이 200원이라면, CoinChange 알고리즘은 160원 동전 1개와 10원 동전 4개로서 총 5개를 리턴

CoinChange 알고리즘의 결과

- 200원에 대한 최소 동전 수는 100원짜리 동전 2개

CoinChange 알고리즘의 결과

최소 동전의 거스름돈

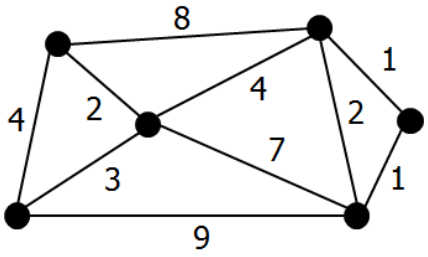
- CoinChange 알고리즘은 항상 최적의 답을 주지 못함

  그러나 실제로는 거스름돈에 대한 그리디 알고리즘이 적용되도록 동전이 발행됨

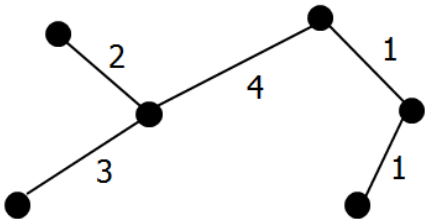
4.2 최소 신장 트리

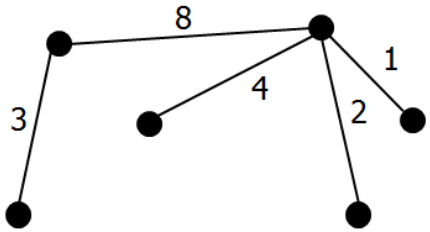
➢ 최소 신장 트리 (Minimum Spanning Tree)

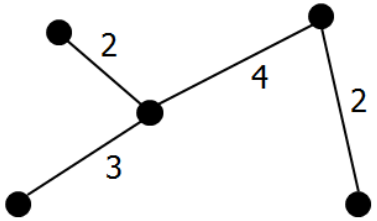
- 주어진 가중치 그래프에서 사이클이 없이 모든 점들을 연결시킨 트리들 중 간선들의 가중치 합이 최소인 트리



다음 중 최소 신장 트리는?



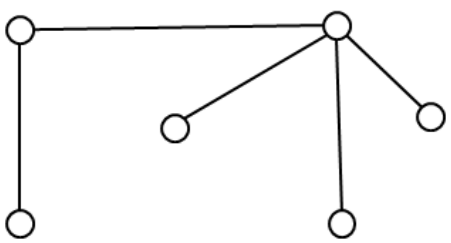




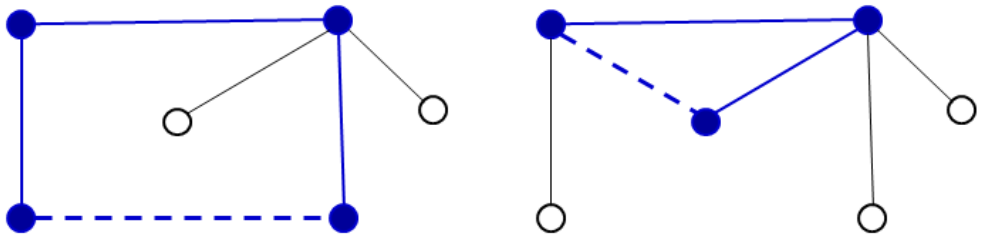
최소 신장 트리

- 주어진 그래프의 신장 트리를 찾는 방법
  - 사이클이 없도록 모든 점을 연결시킨다.

- 그래프의 점의 수 = n
  - 신장 트리에는 정확히 (n-1)개의 간선이 있다.
  - 트리에 간선을 하나 추가시키면, 반드시 사이클이 만들어진다.



트리



점선으로 된 선분을 추가하여 만들어진 사이클

최소 신장 트리 알고리즘

- 크러스컬(Kruskal) 알고리즘
  - 가중치가 가장 작은 간선이 사이클을 만들지 않을 때에만 '욕심내어' 그 간선을 추가시킨다.

- 프림(Prim) 알고리즘
  - 임의의 점 하나를 선택한 후, (n-1)개의 간선을 하나씩 추가시켜 트리를 만든다.

- 알고리즘의 입력은 1개의 연결 성분(connected component)로 된 가중치 그래프

알고리즘

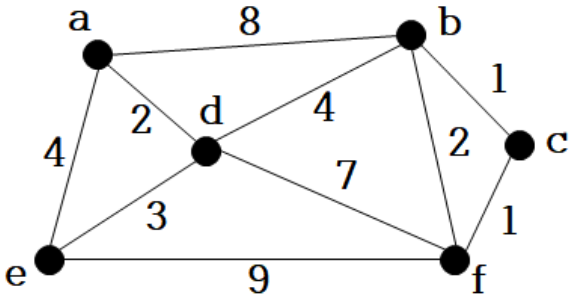
KruskalMST(G)

입력: 가중치 그래프 G=(V,E), |V|=n , |E|=m

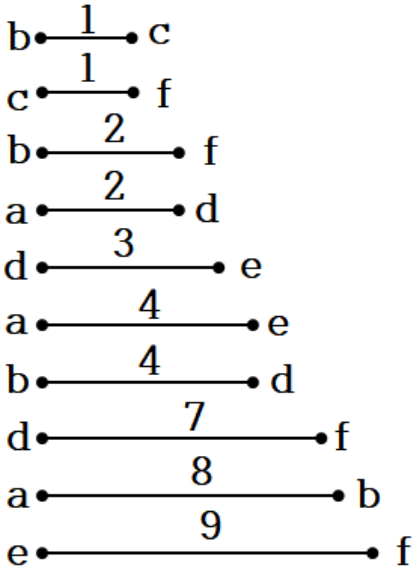
출력: 최소 신장 트리 T

1. 가중치의 오름차순으로 간선들을 정렬: L = 정렬된 간선 리스트
2. T=∅ // 트리 T를 초기화
3. while (T의 간선 수 < n-1)
4. L에서 가장 작은 가중치를 가진 간선 e를 가져오고, e를 L에서 제거
5. if (간선 e가 T에 추가되어 사이클을 만들지 않으면)
6. e를 T에 추가
7. else // e가 T에 추가되어 사이클이 생기는 경우
8. e를 버린다.
9. return 트리 T // T는 최소 신장 트리

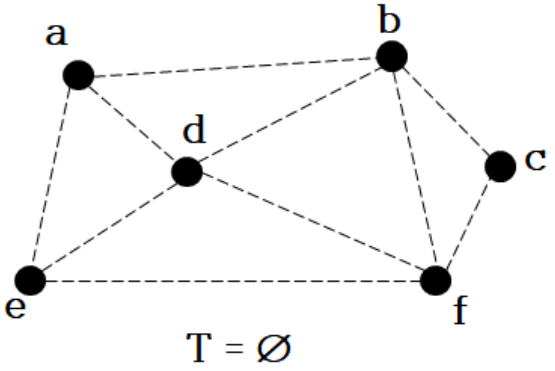
KruskalMST 알고리즘 수행 과정



입력 그래프



정렬된 리스트 L

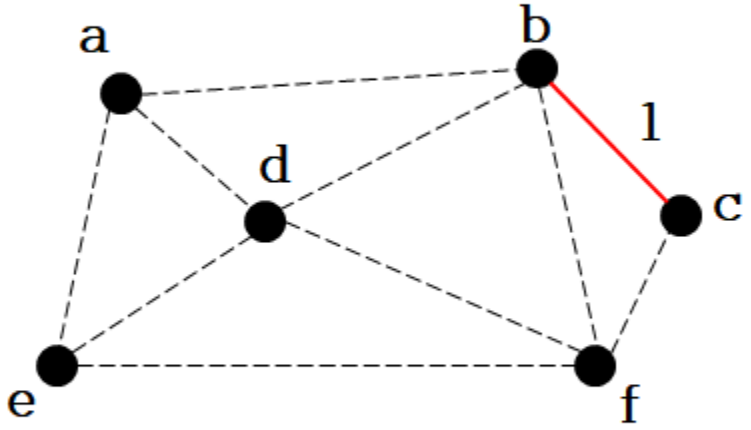


T = ∅

수행 과정

리스트 L

b 1 c
c 1 f
b 2 f
a 2 d
d 3 e
a 4 e
b 4 d
d 7 f
a 8 b
e 9 f

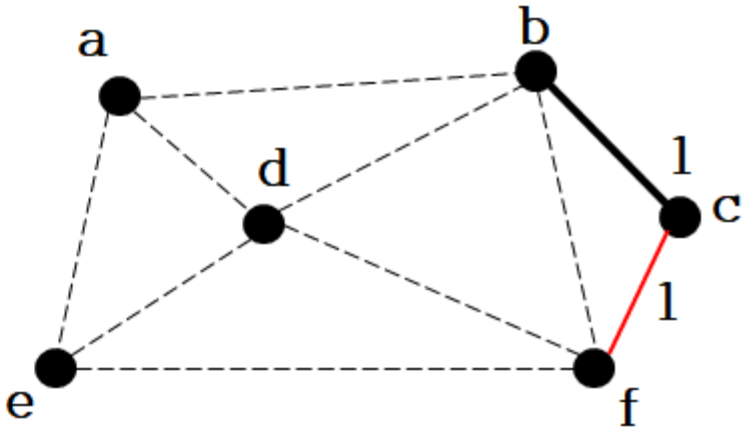


간선 (b, c) 추가

수행 과정

리스트 L

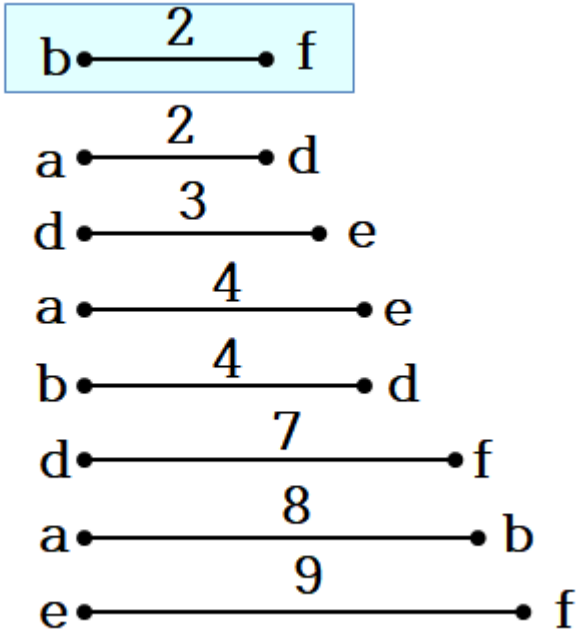
c 1 f
b 2 f
a 2 d
d 3 e
a 4 e
b 4 d
d 7 f
a 8 b
e 9 f

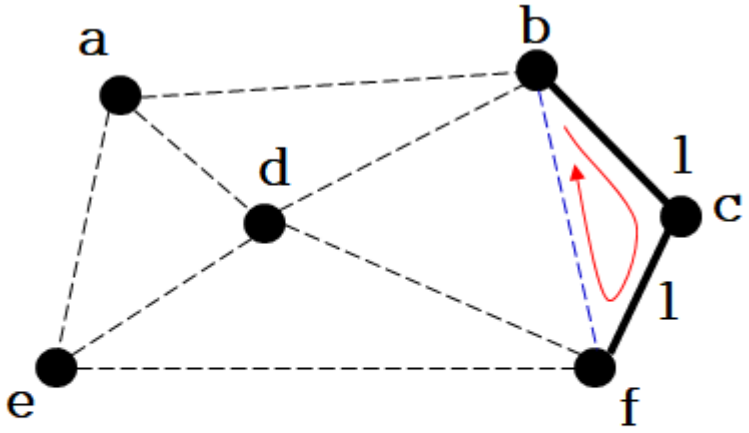


간선 (c, f) 추가

수행 과정

리스트
L



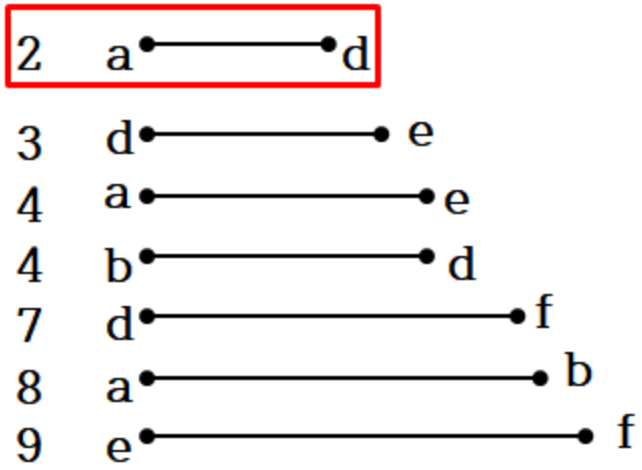


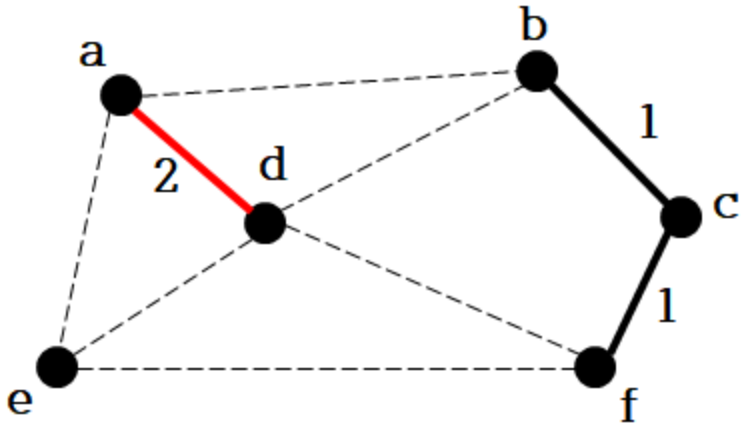
사이클 b-c-f-b

간선 (b, f) 버림

수행 과정

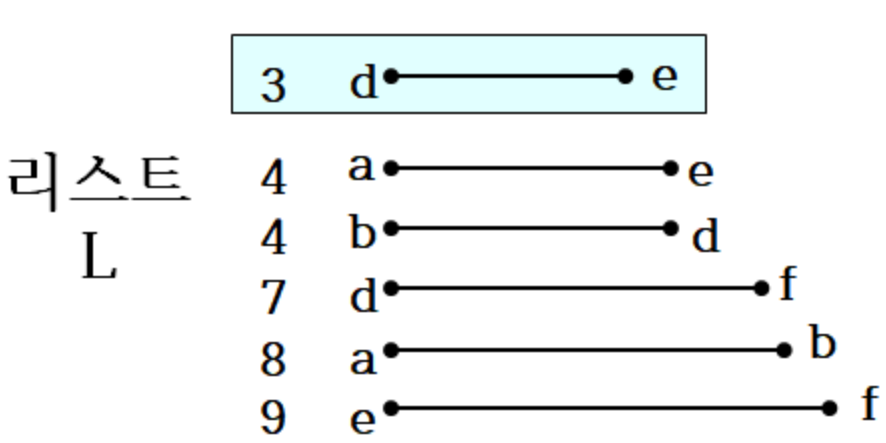
리스트
L

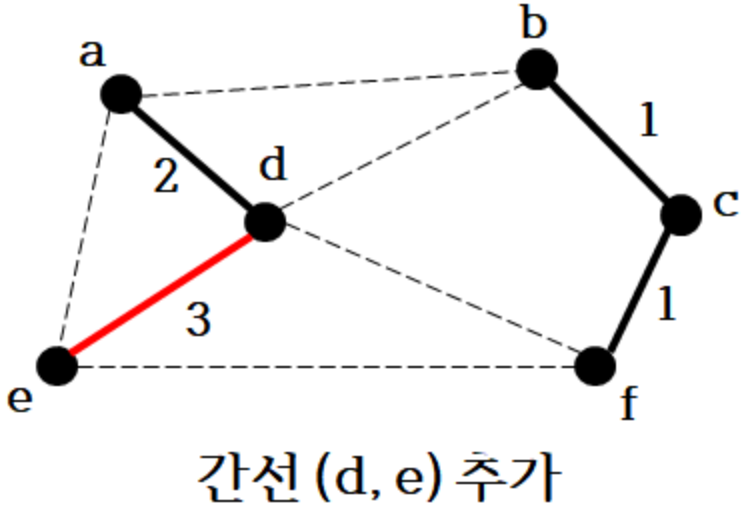




간선 (a, d) 추가

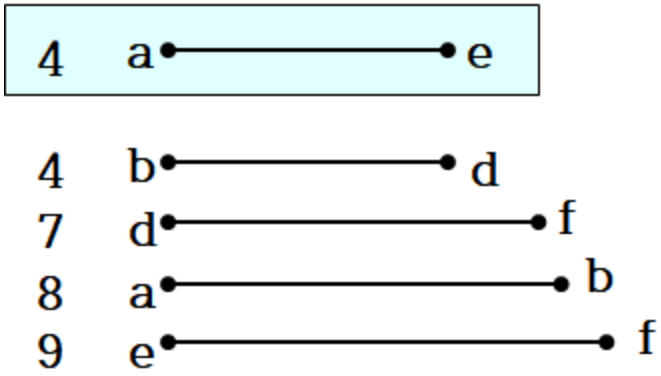
수행 과정

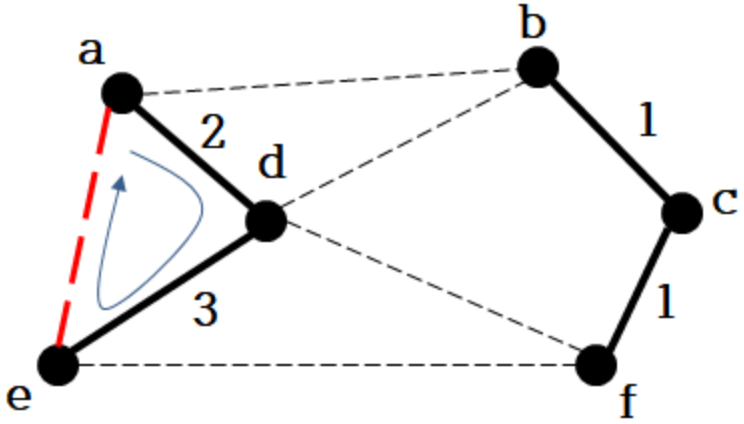




간선 (d, e) 추가

수행 과정

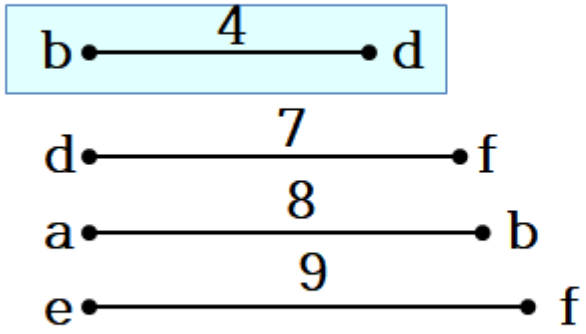


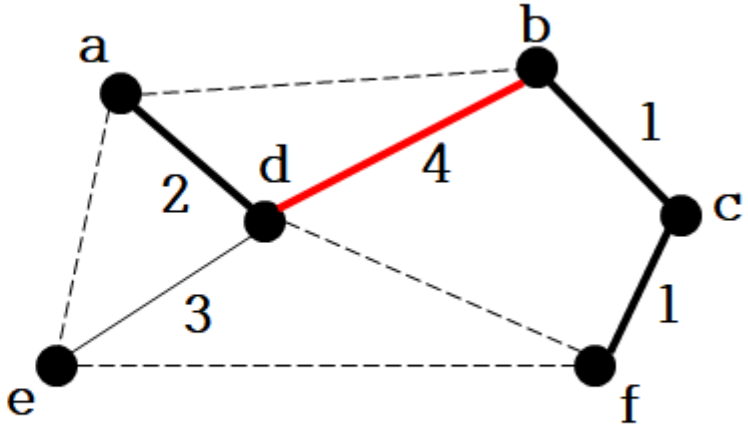


사이클 a-d-e-a

수행 과정

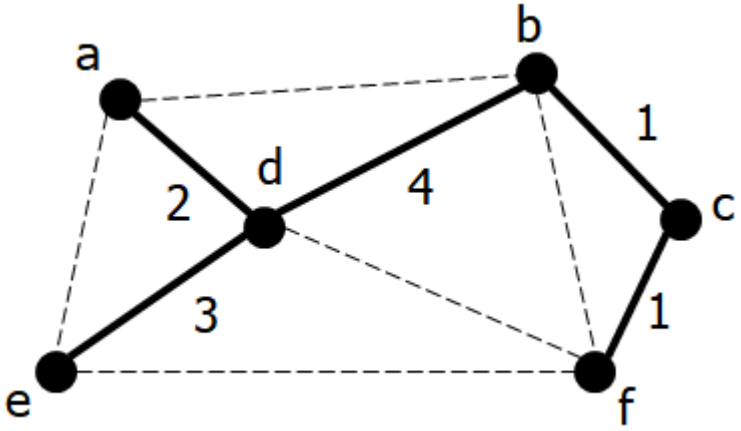
리스트
L





간선 (b, d) 추가

MST



시간 복잡도

- Line 1 : 간선 정렬하는데 **O(mlogm)** 시간

  단, m은 입력 그래프에 있는 간선의 수

- Line 2 : T를 초기화하는 것이므로 O(1) 시간

- Line 3~8
  - while-루프는 최대 m번 수행

    그래프의 모든 간선이 while-루프 내에서 처리되는 경우
  - while-루프 내에서는 L로부터 가져온 간선 e가 사이클을 만드는지를 검사하는데 거의 O(1) 시간

- Kruskal 알고리즘의 시간복잡도: **O(mlogm)**

프림 (Prim)의 MST 알고리즘

- 주어진 가중치 그래프에서 임의의 점 하나를 선택한 후, (n-1)개의 간선을 하나씩 추가시켜 트리 생성

- 추가되는 간선은 현재까지 만들어진 트리에 연결시킬 때 '욕심내어' 항상 최소의 가중치로 연결되는 간선

알고리즘

PrimMST(G)

입력: 가중치 그래프 G=(V,E), |V|=n, |E|=m

출력: 최소 신장 트리 T

```
1. G에서 임의의 점 p를 시작점으로 선택 D[p] = 0
     // D[v]는 T에 있는 u와 v를 연결하는 간선의 최소 가중치를 저장
     하기 위한 원소
2. for (점 p가 아닌 각 점 v에 대하여) {     // 배열 D의 초기화
3.     if ( 간선 (p, v)가 그래프에 있으면 )
4.         D[v] = 간선 (p, v)의 가중치
5.     else
6.         D[v]=∞
   }
```

Prim의 MST 알고리즘

7. T= {p} // 초기에 트리 T는 점 p만을 가진다.

8. while (T에 있는 점의 수 < n) {

9. T에 속하지 않은 각 점 v에 대하여, D[v]가 최소인 점 v_{min}과 연결된 간선 (u, v_{min})을 T에 추가, 여기서 u는 T에 속한 점이고, 점 v_{min}도 T에 추가

10. for (T에 속하지 않은 각 점 w에 대해서) {

11. if (간선 (v_{min}, w)의 가중치 < D[w])

12. D[w] = 간선 (v_{min}, w)의 가중치 // D[w]를 갱신

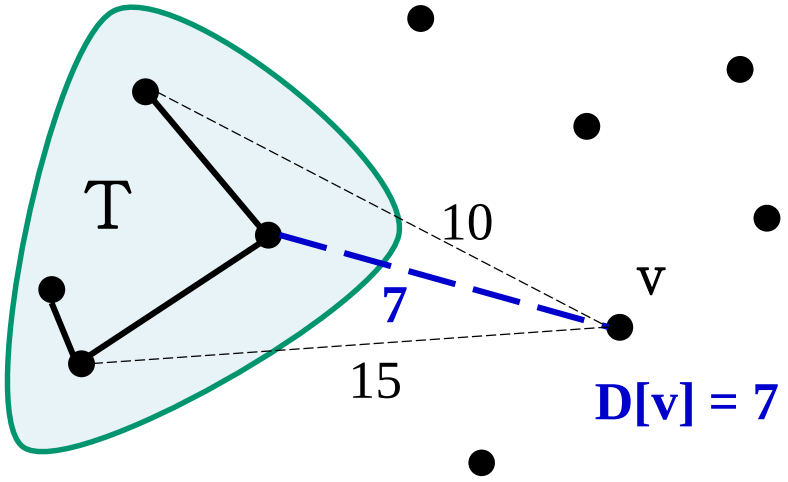
}

}

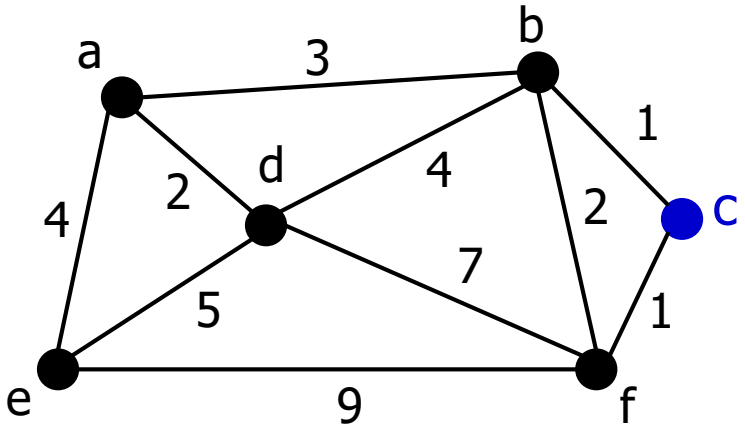
13. return T // T는 최소 신장 트리

D[v] 설명

- Line 1
  - 임의로 점 p를 선택하고, D[p]=0으로 놓는다.
  - D[v]에는 점 v와 T에 속한 점들을 연결하는 간선들 중에서 최소 가중치를 가진 간선의 가중치를 저장
  - 그림에서 D[v]에는 10, 7, 15 중에서 최소 가중치인 7이 저장



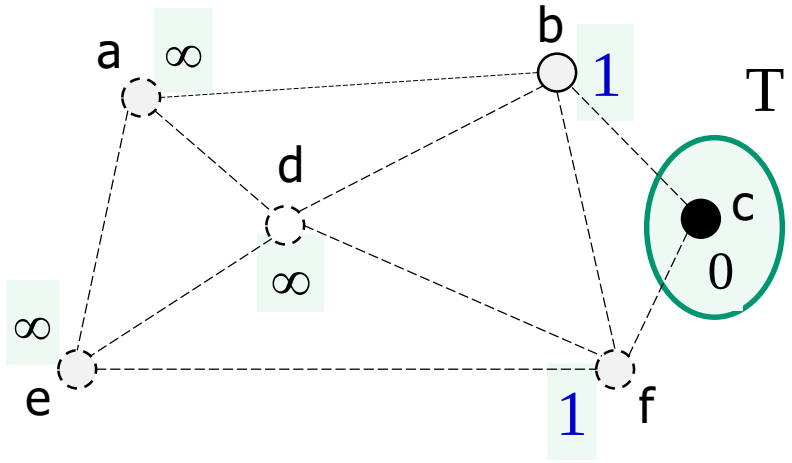
수행 과정



- Line 1: 임의의 점 c 선택, D[c]=0으로 초기화

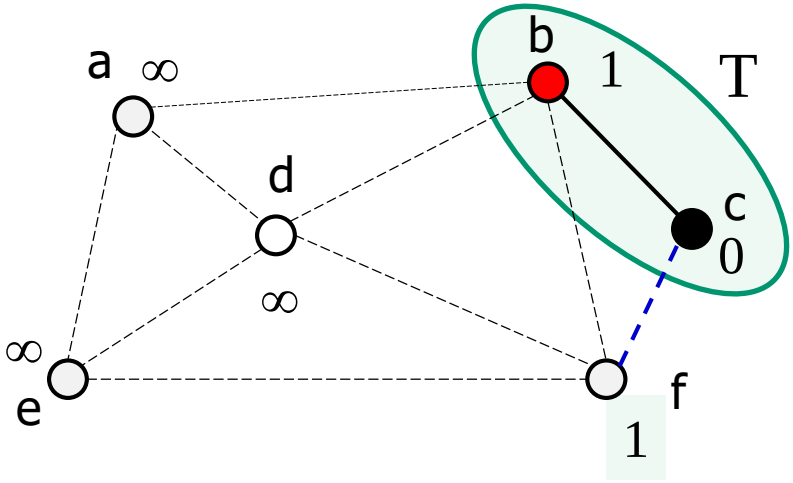
수행 과정

- Line 2~6:
  - 시작점 c와 간선으로 연결된 각 점 v에 대해서, D[v]를 각 간선의 가중치로 초기화
  - 나머지 각 점 v에 대해서, D[v]는 ∞로 초기화



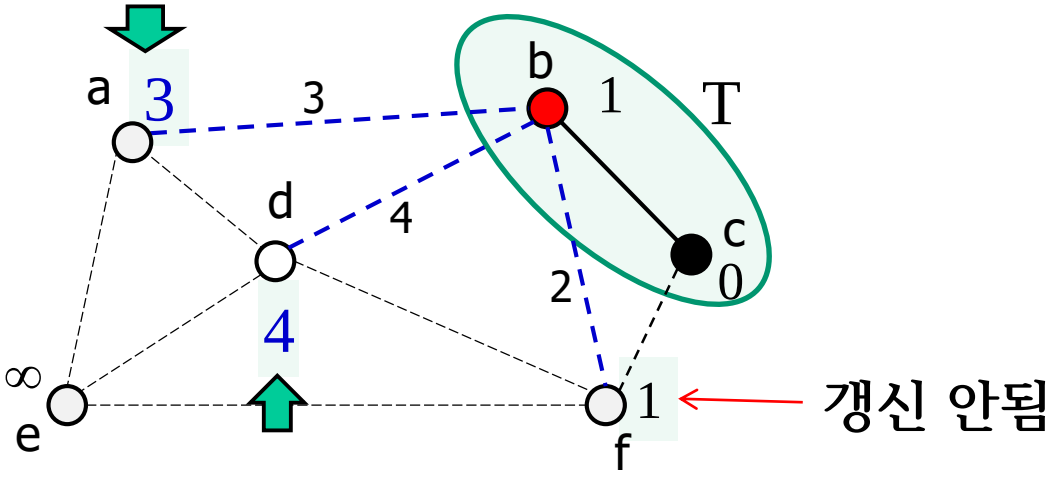
수행 과정

- Line 7: T={c}로 초기화
- Line 8-9: T에 가장 가까운 점 b를 추가

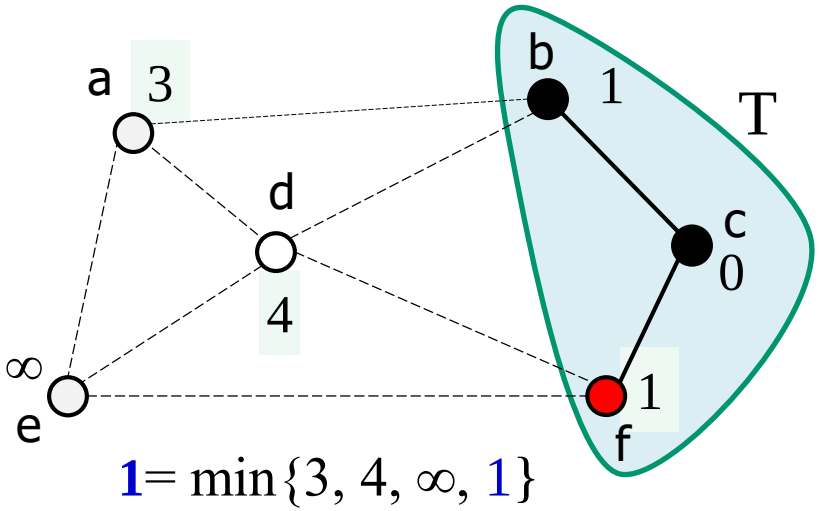


$1 = \min\{\infty, 1, \infty, \infty, 1\}$

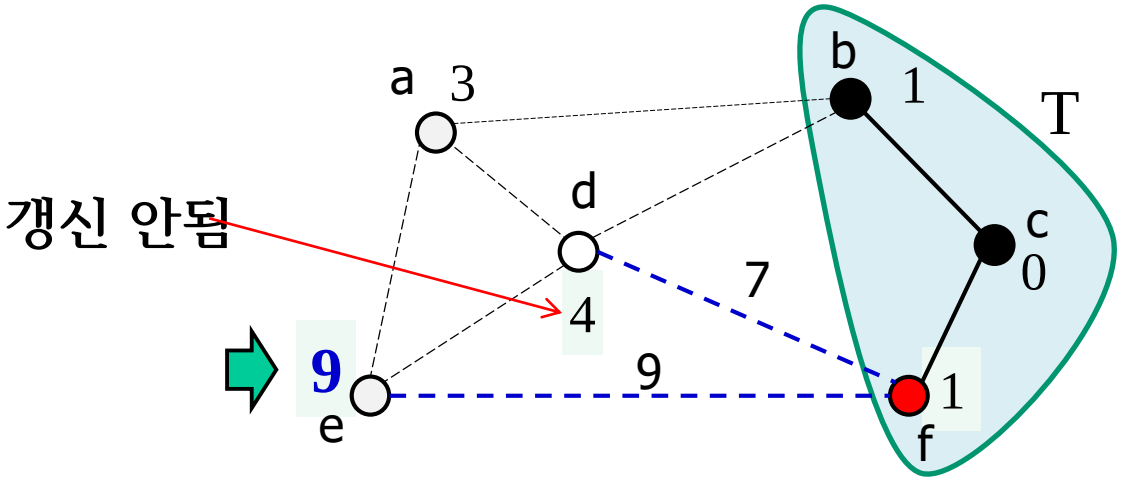
b에 연결된 a와 d의 D[a]와 D[d] 갱신



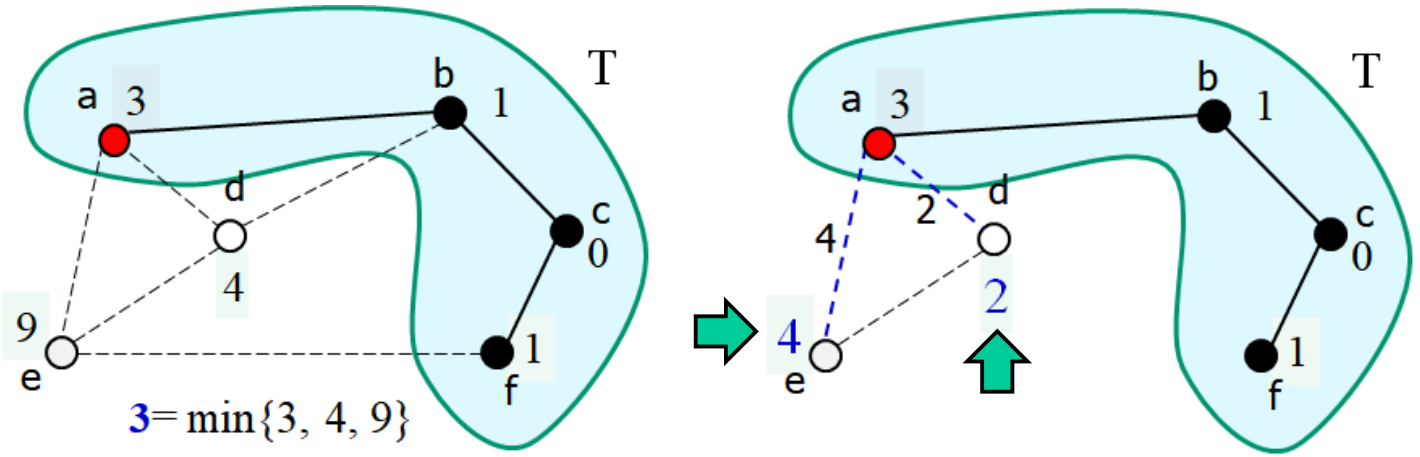
T에 가장 가까운 점 f를 추가



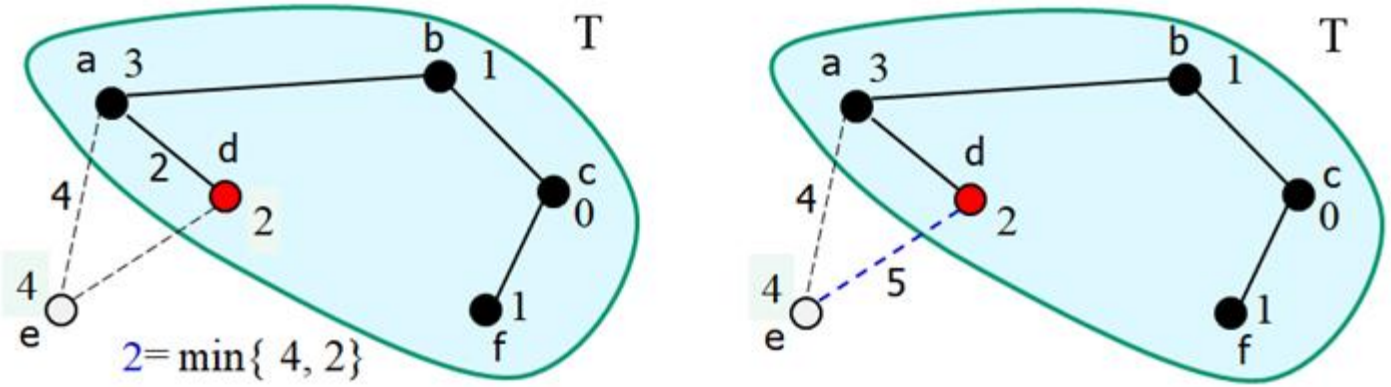
f에 연결된 e의 D[e] 갱신



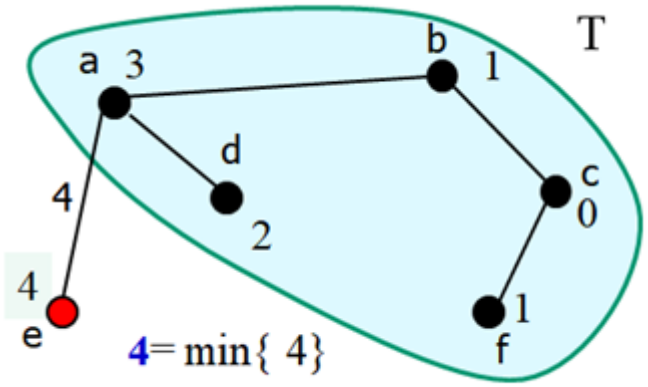
a를 T에 추가

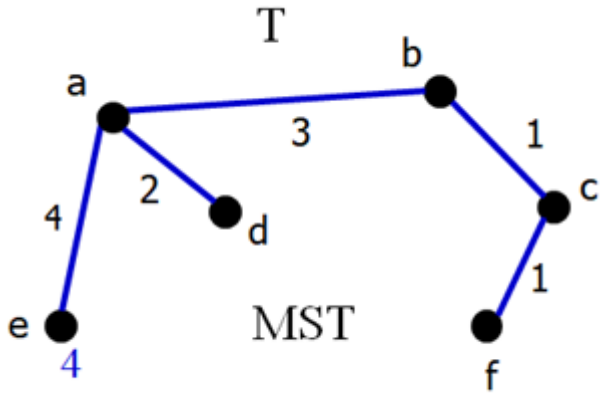


d를 T에 추가



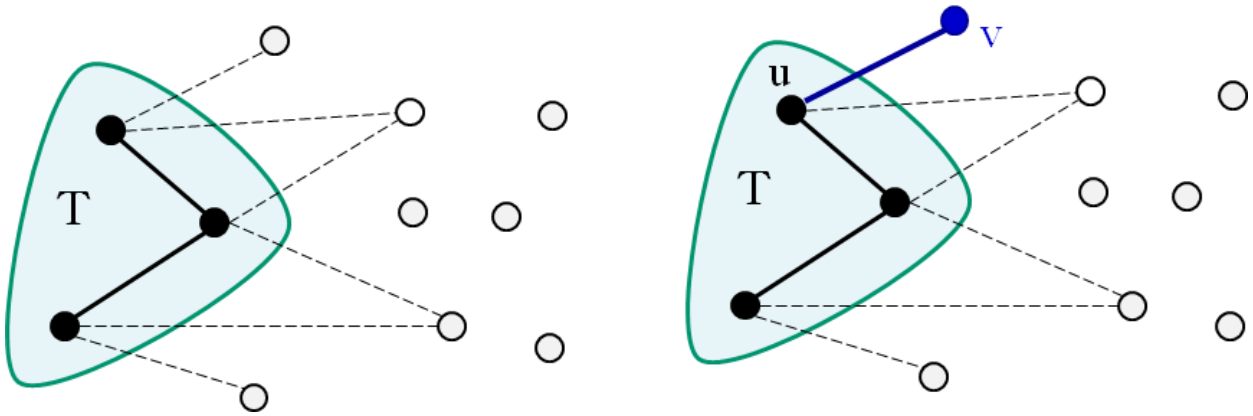
e를 T에 추가





PrimMST가 찾은 T에는 왜 사이클이 없을까?

- 프림 알고리즘은 T 밖에 있는 점을 항상 추가하므로 사이클이 만들어지지 않는다.



시간 복잡도

- while-루프가 (n-1)회 반복되고,
  - 1회 반복될 때 line 9에서 T에 속하지 않은 각 점 v에 대하여, D[v]가 최소인 점 v_{min}을 찾는데 O(n) 시간 소요
  - 배열 D에서 (현재 T에 속하지 않은 점들에 대해서) 최솟값을 찾는 것이고, 배열의 크기는 n이기 때문

- 프림 알고리즘의 시간 복잡도
  - (n-1) x O(n) = $O(n^2)$
  - 최소 힙(Binary Heap)을 사용하여 v_{min}을 찾으면 O(mlogn), m은 간선 수. 따라서 간선 수가 O(n)이면 O(nlogn)

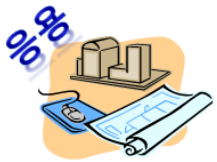
Kruskal과 Prim 알고리즘의 수행 과정 비교

크러스컬 알고리즘

- 간선이 1개씩 T에 추가되는데, 이는 마치 n개의 점들이 각각의 트리인 상태에서 간선이 추가되면 2개의 트리가 1개의 트리로 합쳐지는 것과 같음
- 크러스컬 알고리즘은 이를 반복하여 1개의 트리인 T를 생성
- n개의 트리들이 점차 합쳐져서 1개의 신장 트리가 만들어진다.

프림 알고리즘

- T가 점 1개인 트리에서 시작되어 간선을 1개씩 추가
- 1개의 트리가 자라나서 신장 트리가 된다.



Applications

- 최소 비용으로 선로 또는 파이프 네트워크 (인터넷 광케이블 선로, 케이블 TV선로, 전화선로, 송유관로, 가스관로, 배수로 등)를 설치하는데 활용

4.3 최단 경로 찾기

- 최단 경로 (Shortest Path) 문제
  - 주어진 가중치 그래프에서 어느 한 출발점에서 또 다른 도착점까지의 최단 경로를 찾는 문제

- 최단 경로를 찾는 가장 대표적인 알고리즘
  - 다익스트라(Dijkstra) 최단 경로 알고리즘

- 다익스트라 알고리즘
  - 주어진 출발점에서 시작
  - 출발점으로부터 최단 거리가 확정되지 않은 점들 중에서 출발점으로부터 가장 가까운 점을 추가하고, 그 점의 최단 거리를 확정

알고리즘

ShortestPath(G, s)

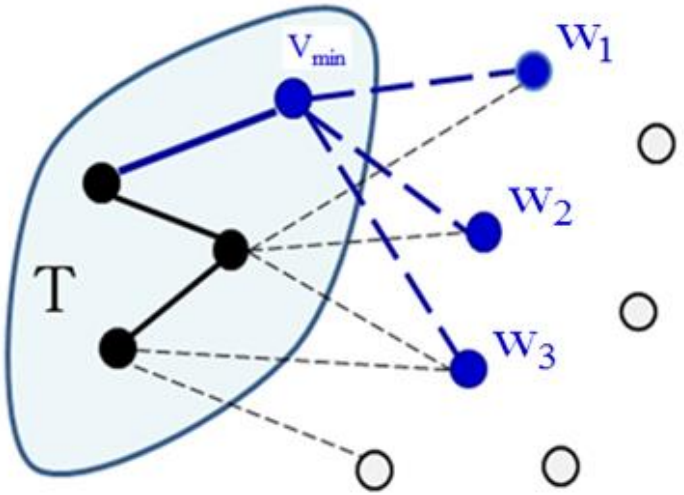
입력: 가중치 그래프 G=(V,E), |V|=n , |E|=m

출력: 출발점 s로부터 (n-1)개의 점까지 각각 최단 거리를 저장한 배열 D

1. 배열 D를 ∞로 초기화. 단, D[s]=0으로 초기화
   // 배열 D[v]에는 출발점 s로부터 점 v까지의 거리를 저장
2. while (s로부터의 최단 거리가 확정되지 않은 점이 있으면)
3. 현재까지 최단 거리가 확정되지 않은 각 점 v에 대해서 최소의 D[v]의 값을 가진 점 v_{min}을 선택하고, s로부터 점 v_{min}까지의 최단 거리 D[v_{min}]을 확정
4. s로부터 현재보다 짧은 거리로 점 v_{min}을 통해 우회 가능한 각 점 w에 대해서 D[w]를 갱신 // 간선 완화
5. return D

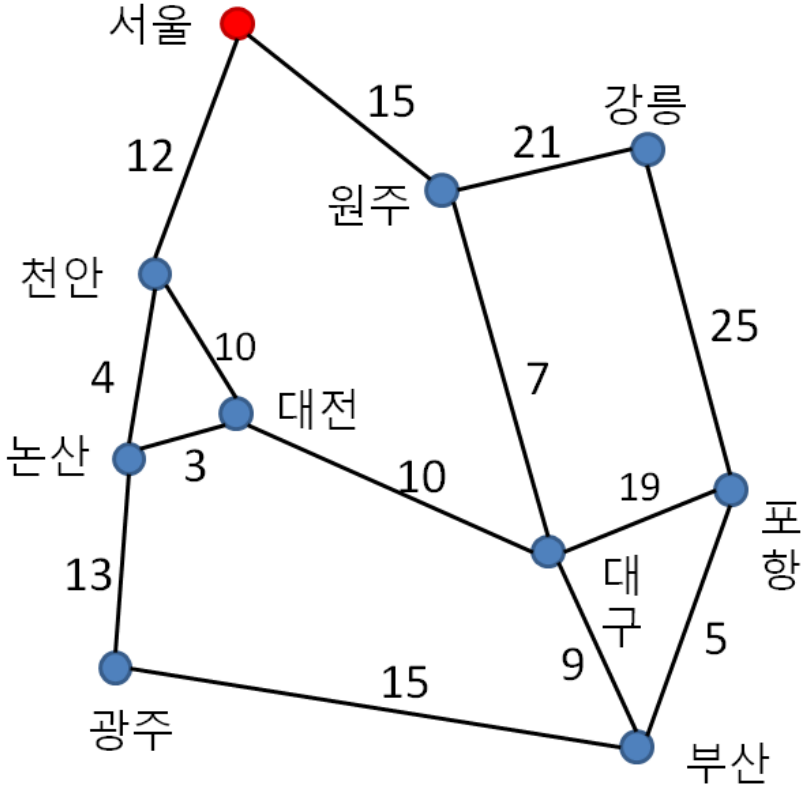
간선 완화(Edge Relaxation)

- Line 4
  - V-T에 속한 점들 중 v_{min}을 거쳐 감 (경유함)으로서 s로부터의 거리가 현재보다 더 짧아지는 점 w가 있으면, 그 점의 D[w]를 갱신
  - v_{min}이 T에 포함된 상태에서 v_{min}에 인접한 점 w_1, w_2, w_3 각각에 대해서 만일 (D[v_{min}]+간선 (v,w_i)의 가중치) < D[w_i]이면, D[w_i] = (D[v_{min}]+간선(v_{min},w_i)의 가중치)로 갱신

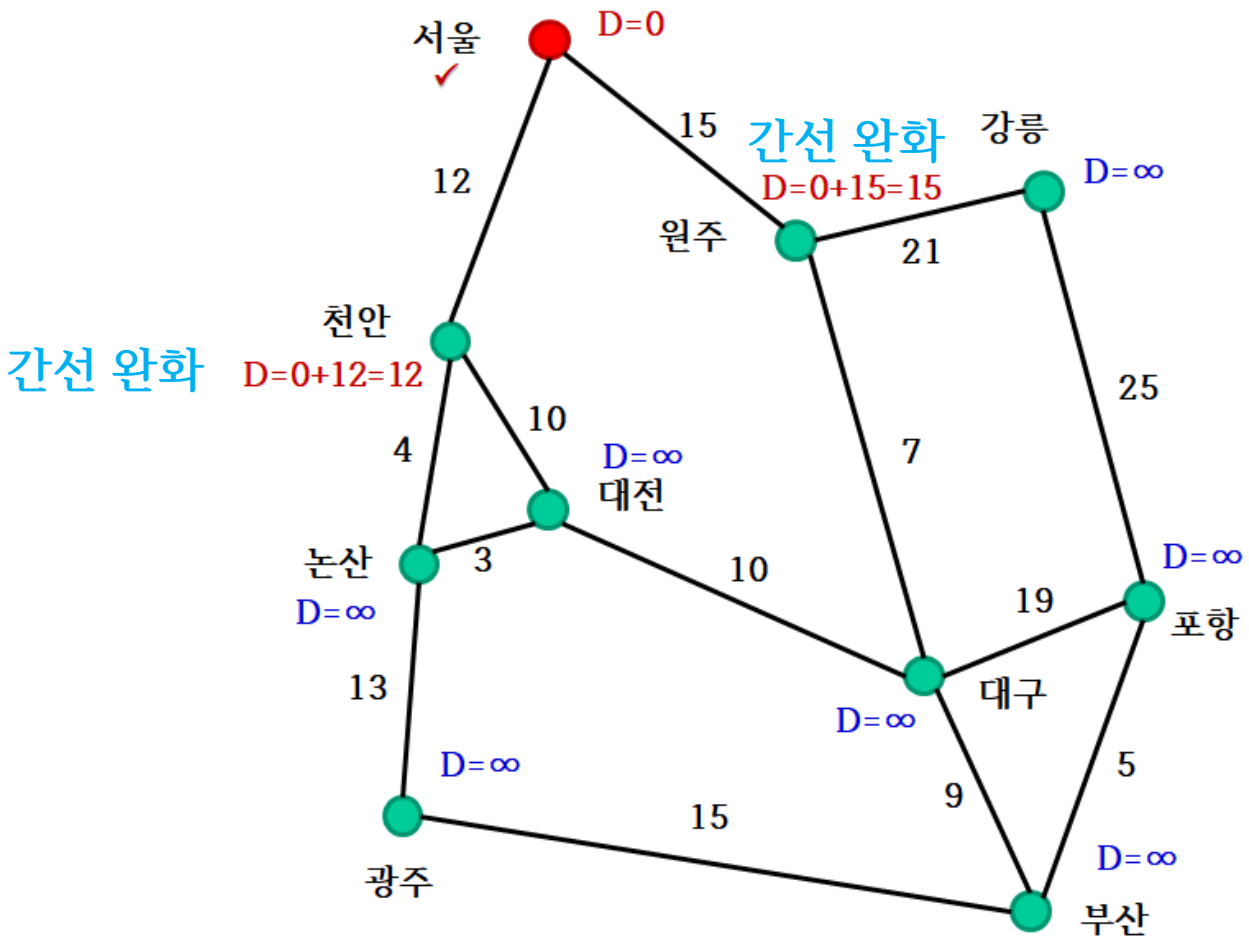


수행 과정

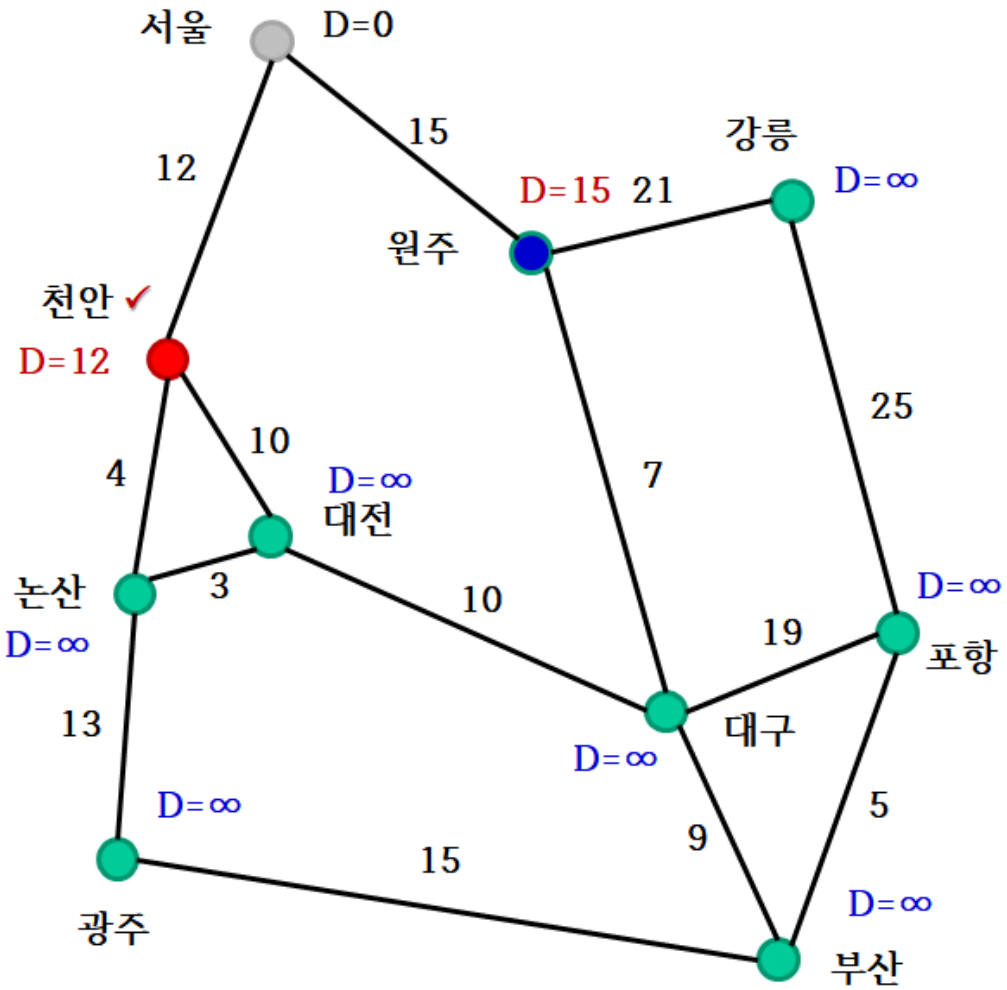
- 서울 확정



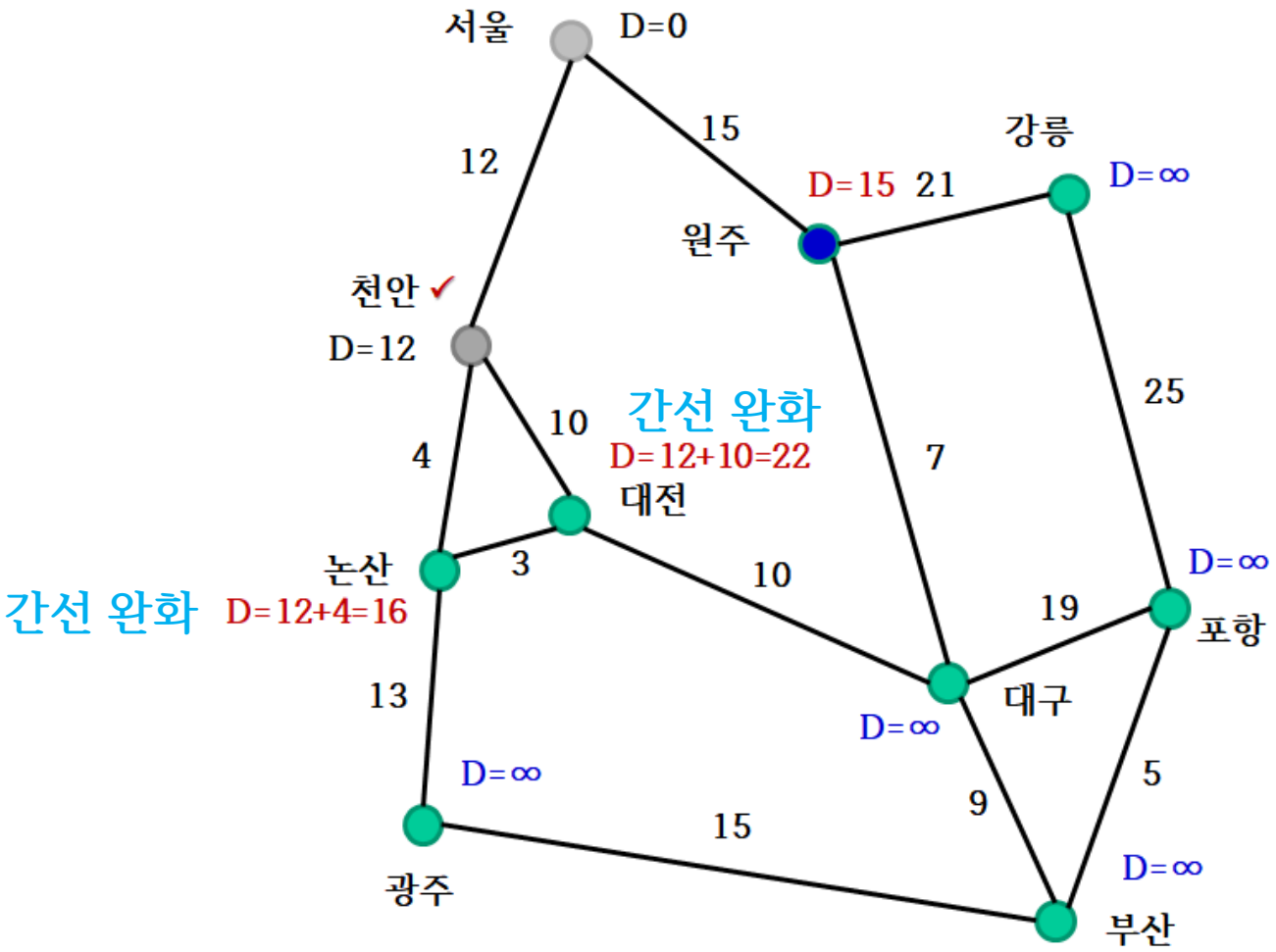
간선 완화


서울
D=0
15
간선 완화
D=0+15=15
강릉
D=∞
12
원주
21
천안
간선 완화
D=0+12=12
25
10
4
D=∞
대전
7
논산
3
10
D=∞
D=∞
19
포항
13
대구
D=∞
D=∞
5
9
15
광주
D=∞
부산

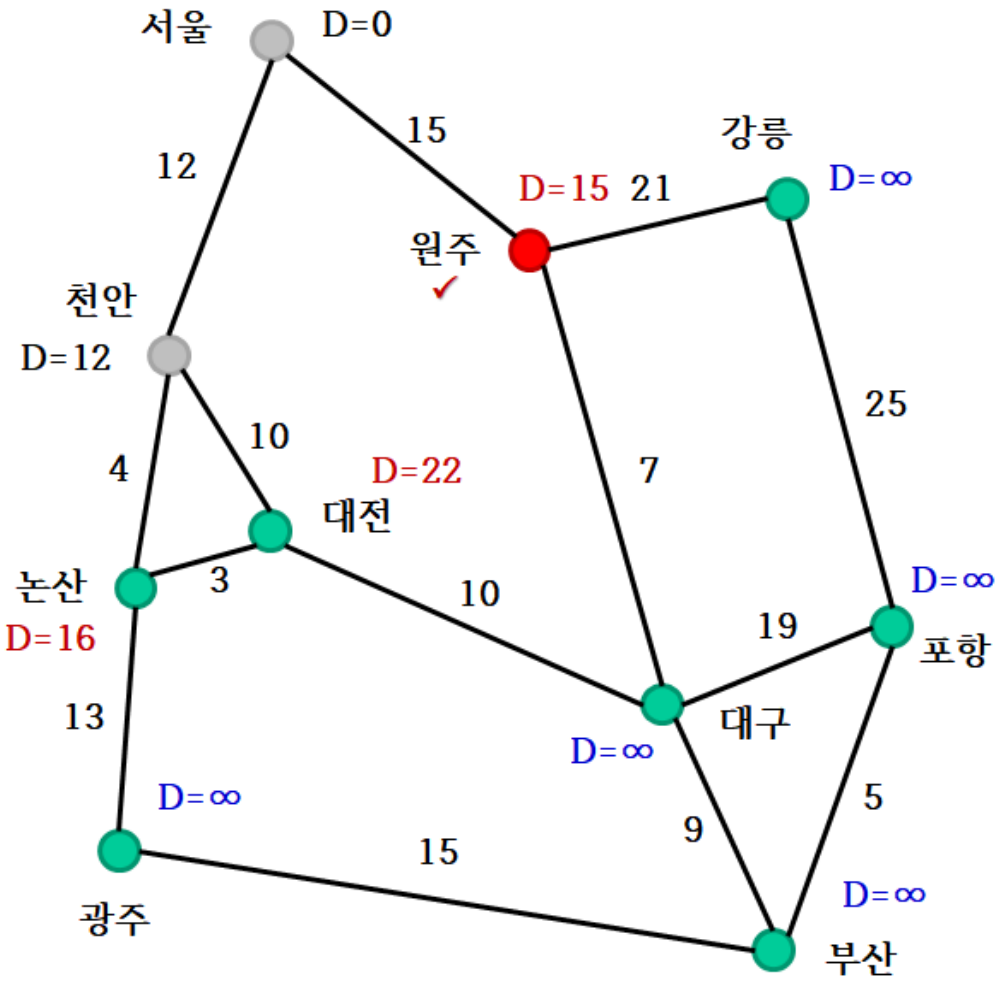
천안 확정


서울
D=0
15
12
강릉
D=15
21
D=∞
원주
천안 ✓
D=12
25
10
4
D=∞
7
대전
논산
3
10
D=∞
D=∞
19
포항
13
대구
D=∞
D=∞
5
9
15
D=∞
광주
부산

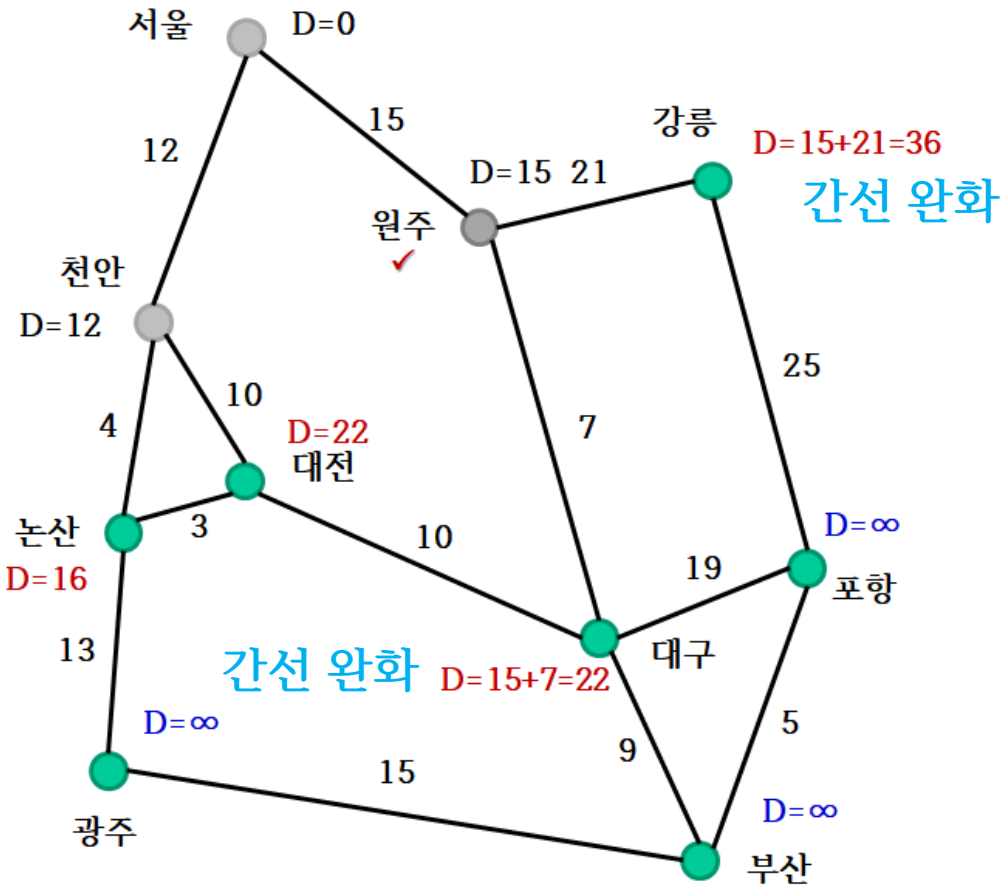
간선 완화


서울
D=0
15
12
강릉
D=15
21
D=∞
원주
천안 ✓
D=12
간선 완화
D=12+10=22
10
25
4
7
대전
논산
3
10
D=∞
간선 완화
D=12+4=16
19
포항
13
대구
D=∞
D=∞
5
9
15
D=∞
광주
부산

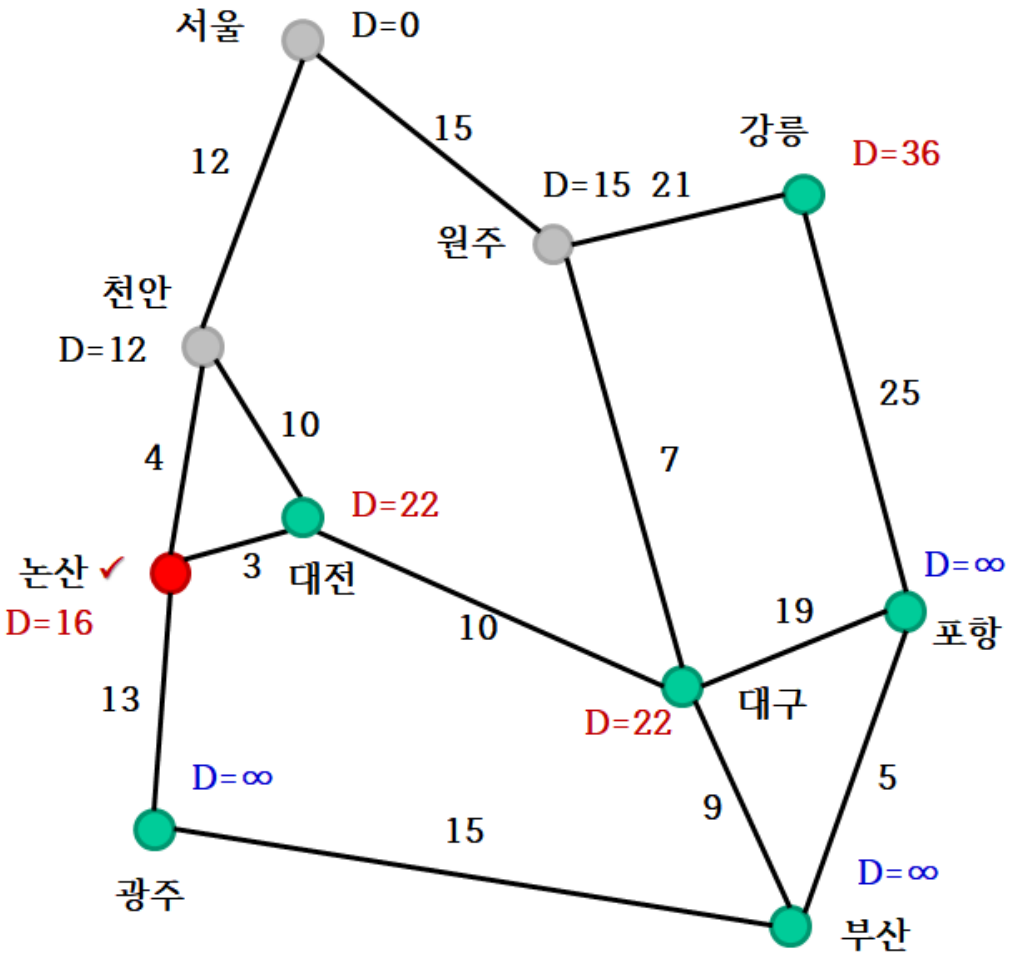
원주 확정


서울
D=0
15
12
강릉
D=15
21
D=∞
원주
✓
천안
D=12
25
10
4
D=22
7
대전
논산
3
10
D=∞
D=16
19
포항
13
대구
D=∞
D=∞
5
9
15
D=∞
광주
부산

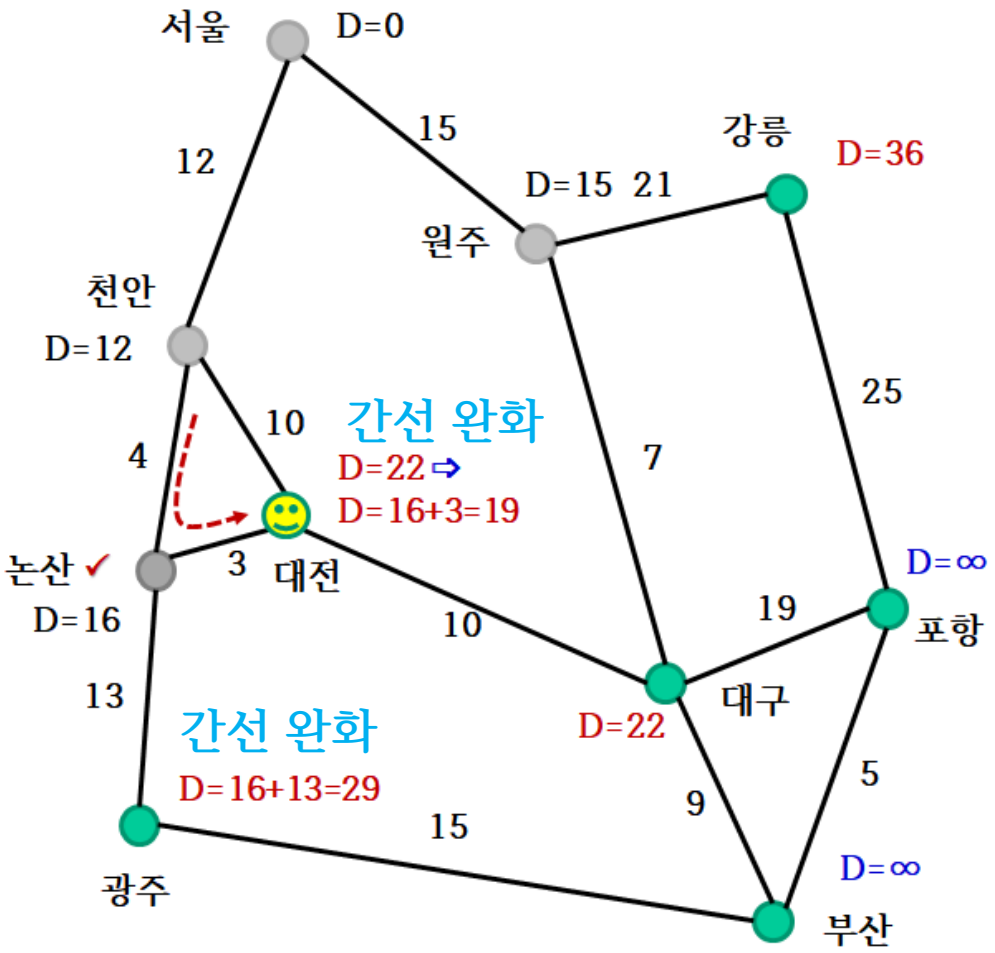
간선 완화


서울
D=0
15
12
강릉
D=15+21=36
간선 완화
D=15 21
원주
천안
D=12
25
10
4
D=22
대전
7
논산
3
10
D=∞
D=16
19
포항
13
간선 완화
D=15+7=22
대구
D=∞
5
9
15
D=∞
광주
부산

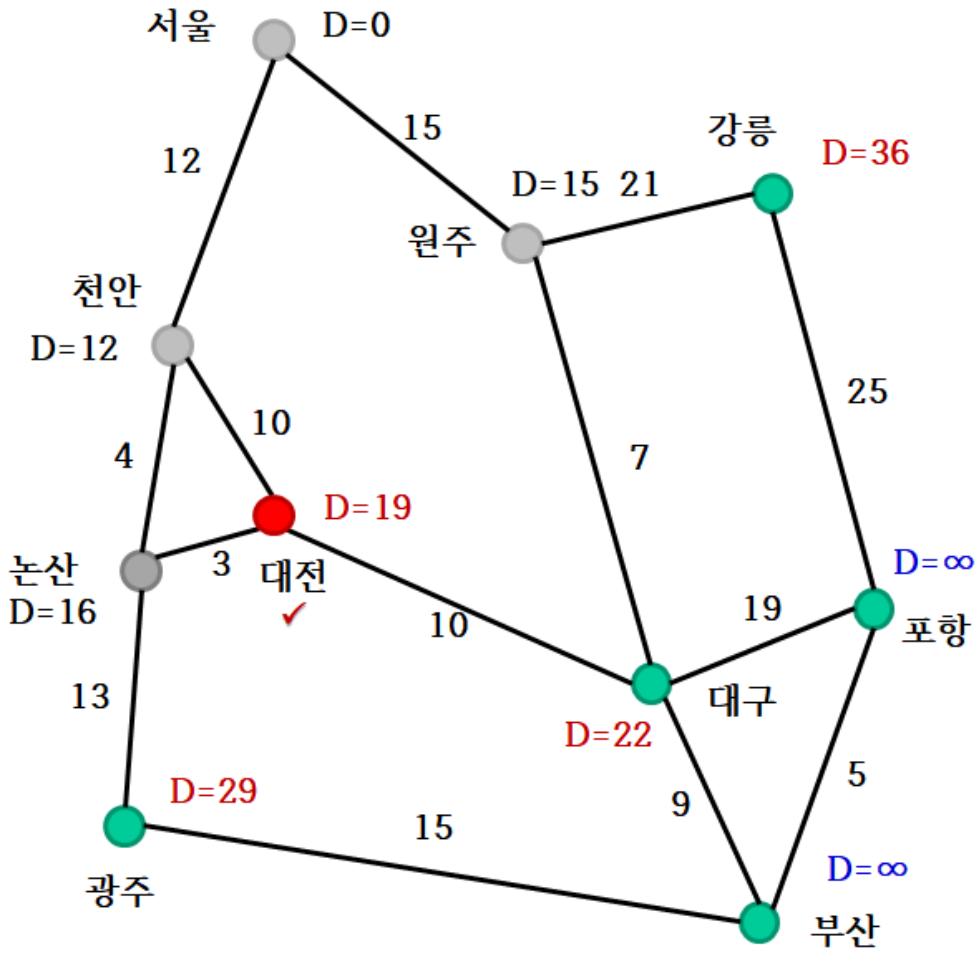
논산 확정


서울
D=0
15
12
강릉
D=36
D=15
21
원주
천안
D=12
25
10
4
7
D=22
논산 ✓
3
대전
D=∞
D=16
19
10
포항
13
대구
D=22
D=∞
5
9
15
D=∞
광주
부산

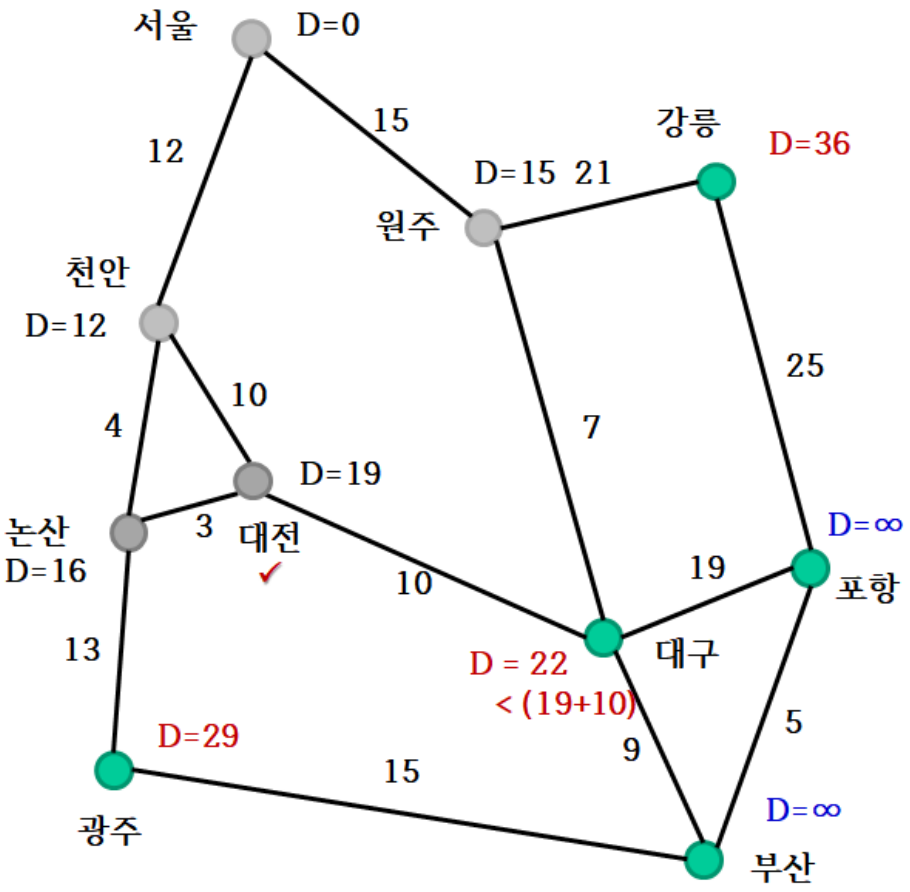
간선 완화


서울
D=0
15
12
강릉
D=36
D=15
21
원주
천안
D=12
25
10
간선 완화
D=22⇨
D=16+3=19
4
7
논산 ✓
3
대전
D=∞
D=16
19
포항
10
13
대구
간선 완화
D=22
D=16+13=29
5
9
15
D=∞
광주
부산

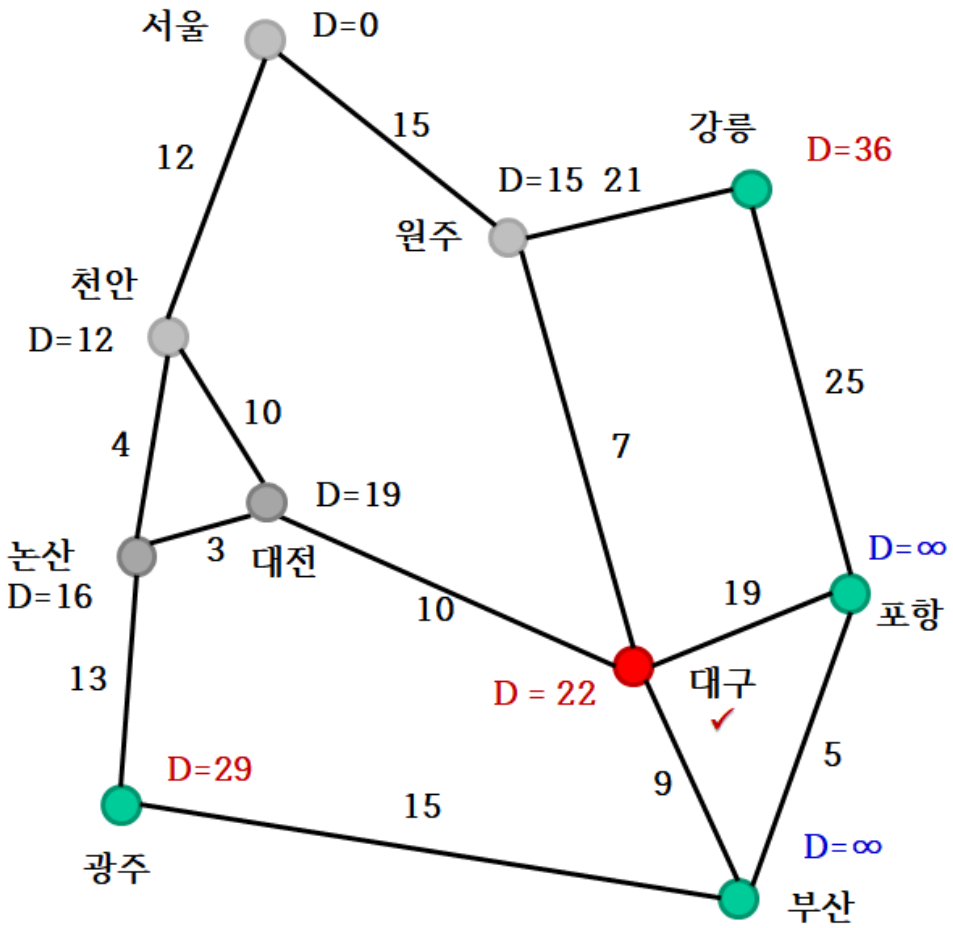
대전 확정


서울
D=0
15
12
강릉
D=36
D=15
21
원주
천안
D=12
25
10
4
7
D=19
논산
D=16
3
대전
D=∞
19
10
포항
13
대구
D=22
5
D=29
9
15
D=∞
광주
부산

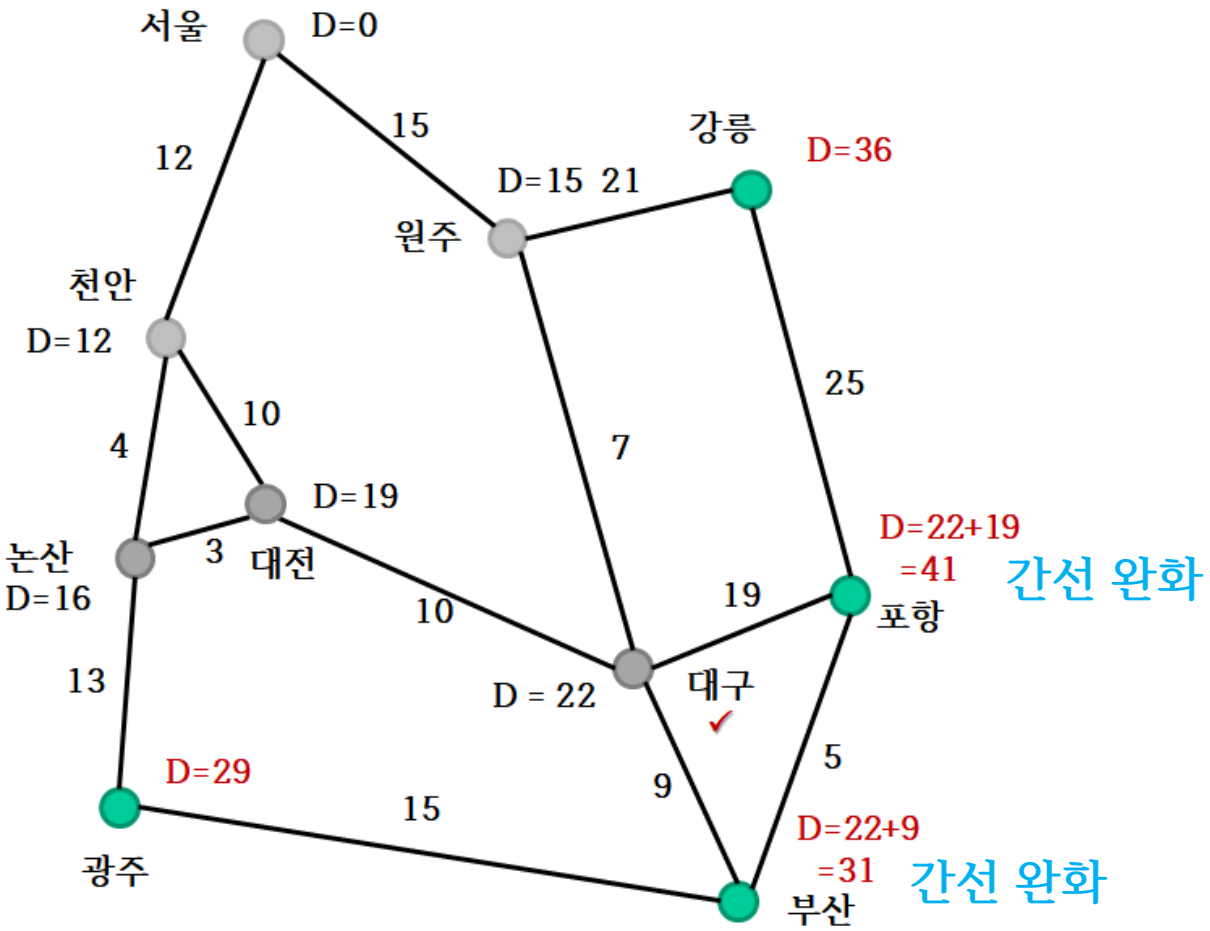
간선 완화 없음


서울
D=0
12
15
강릉
D=36
D=15
21
원주
천안
D=12
25
10
4
7
D=19
논산
D=16
3
대전
D=∞
19
10
포항
13
대구
D = 22
< (19+10)
D=29
5
9
15
D=∞
광주
부산

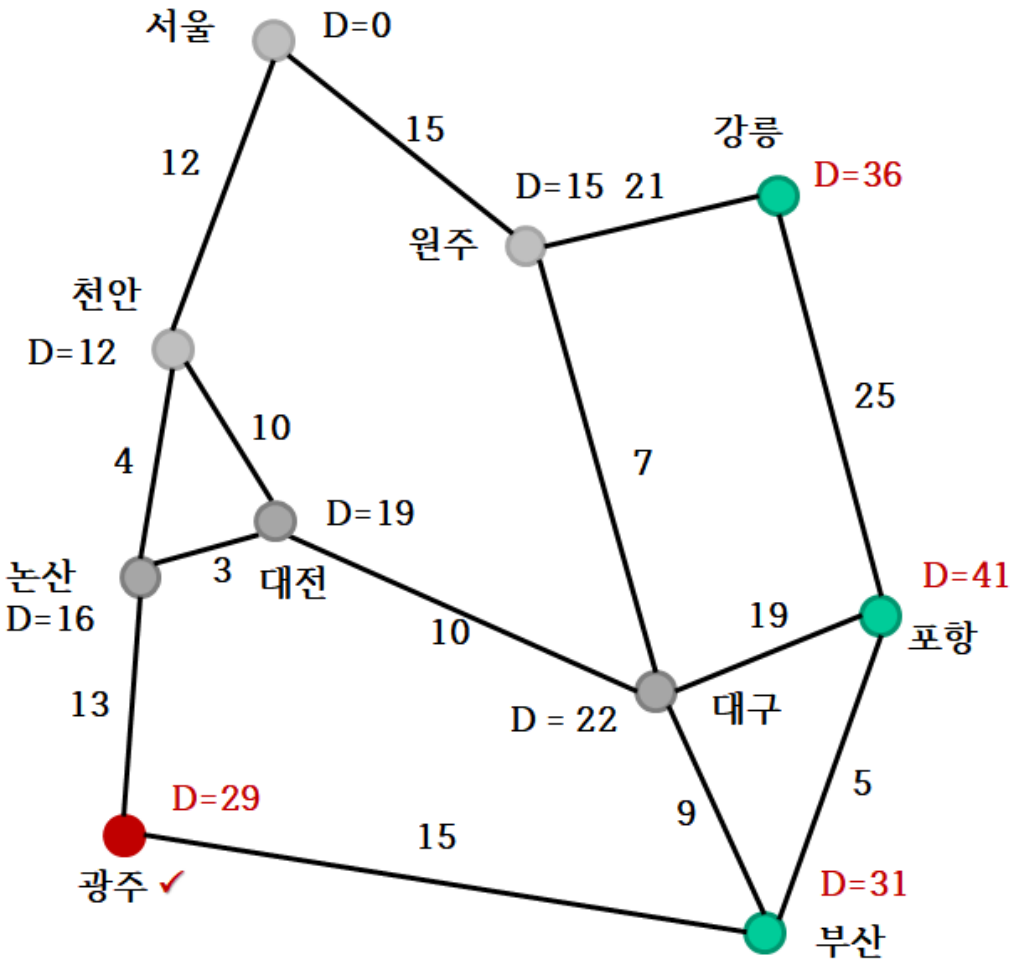
대구 확정


서울
D=0
15
12
강릉
D=36
D=15
21
원주
천안
D=12
25
10
4
7
D=19
논산
D=16
3
대전
D=∞
19
포항
10
13
D = 22
대구
D=29
5
9
15
D=∞
광주
부산

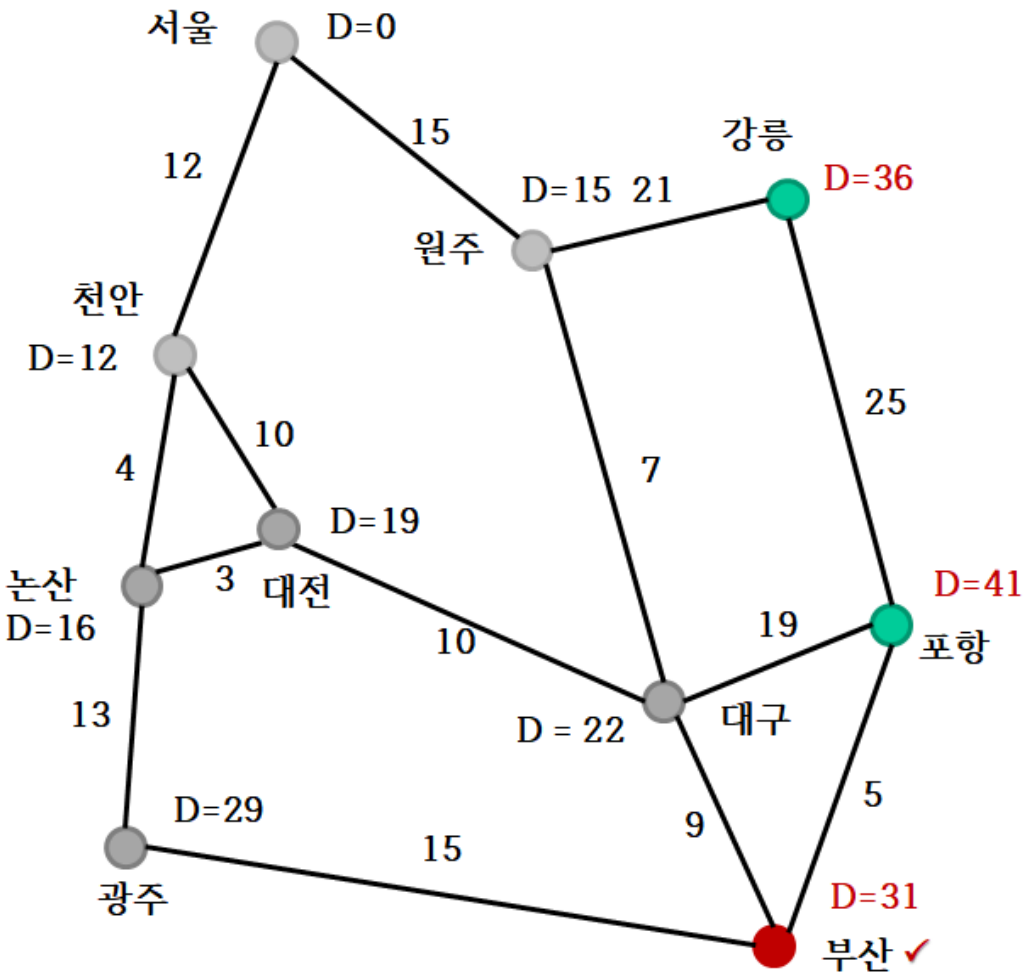
간선 완화


서울
D=0
12
15
강릉
D=36
D=15
21
원주
천안
D=12
25
10
4
7
D=19
D=22+19
=41
간선 완화
논산
D=16
3
대전
19
10
포항
13
D = 22
대구
5
D=29
9
15
D=22+9
=31
간선 완화
광주
부산

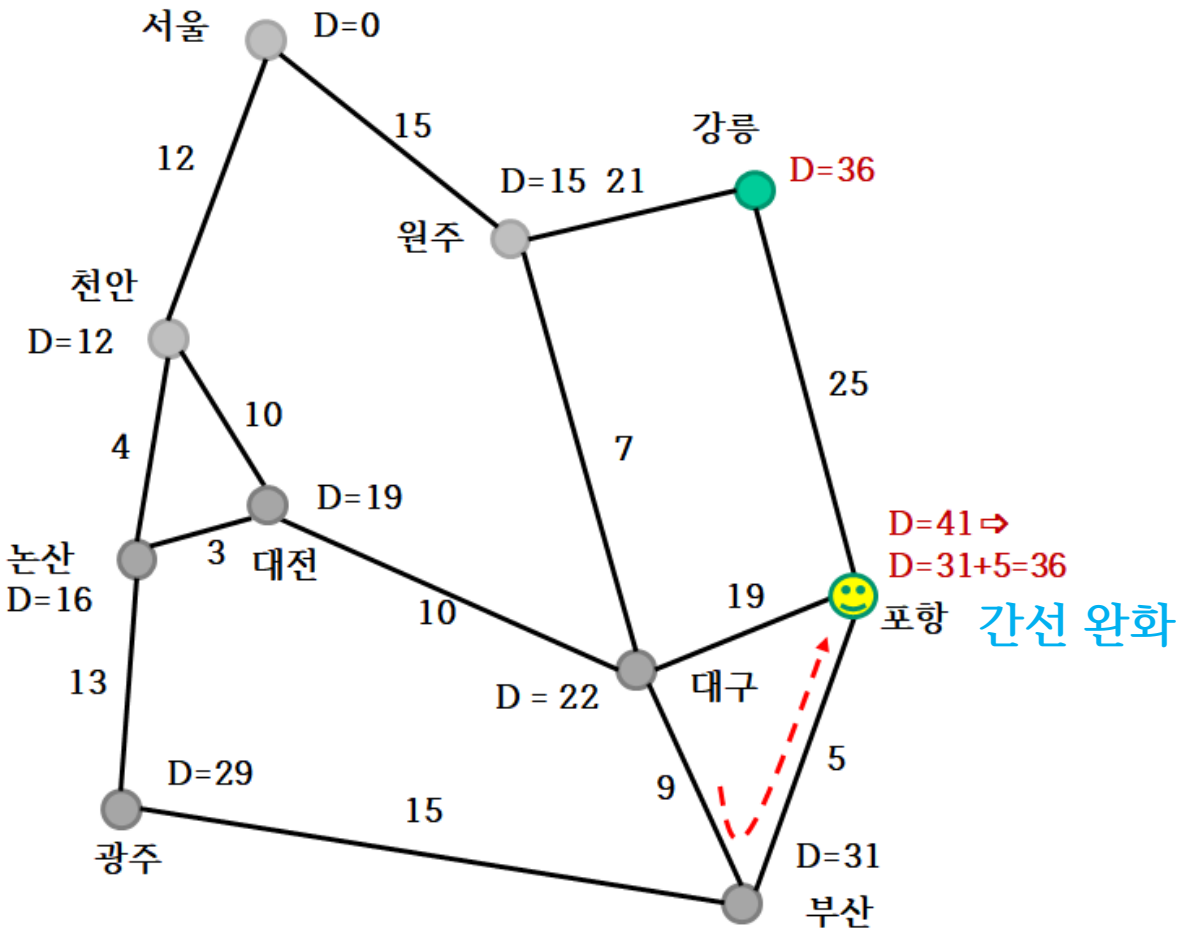
광주 확정


서울
D=0
12
15
강릉
D=15
21
D=36
원주
천안
D=12
25
10
4
7
D=19
논산
D=16
3
대전
D=41
19
10
포항
13
D = 22
대구
5
D=29
9
15
광주 ✓
D=31
부산

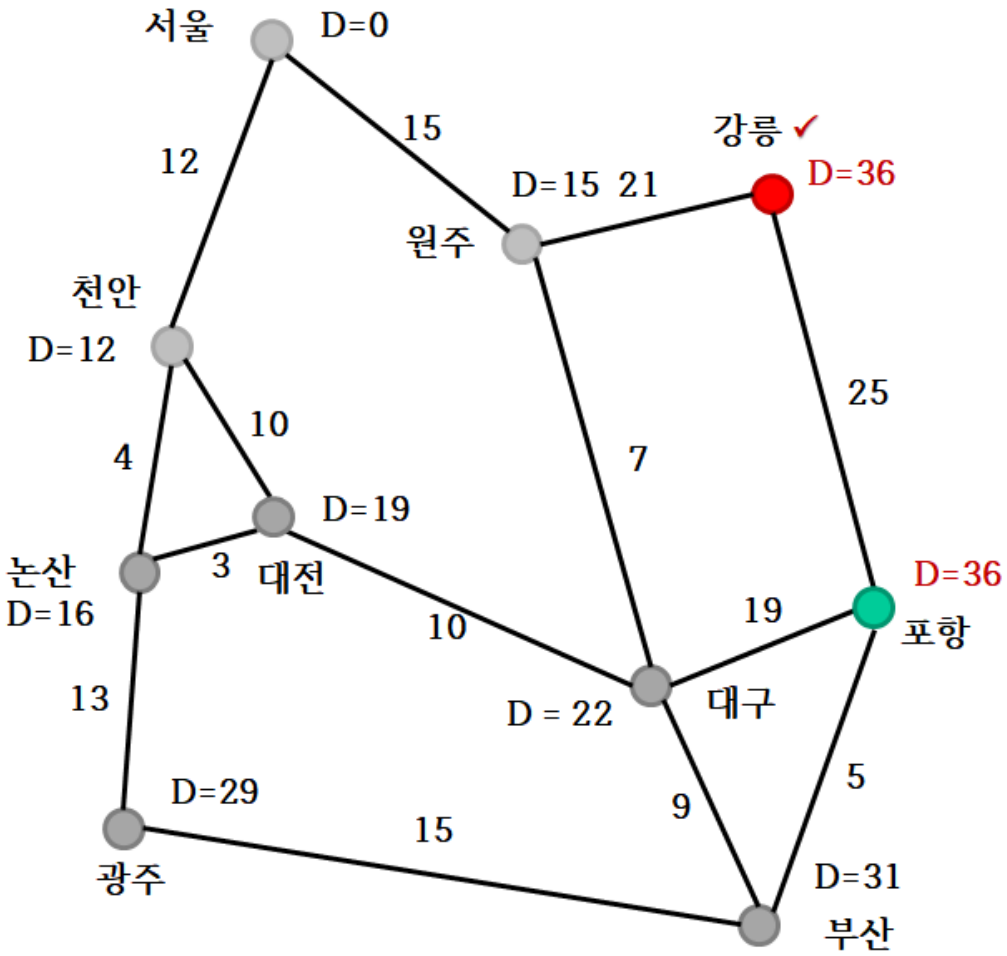
부산 확정


서울
D=0
15
12
강릉
D=15
21
D=36
원주
천안
D=12
25
10
4
7
D=19
논산
3
대전
D=41
D=16
19
10
포항
13
D = 22
대구
5
D=29
9
15
D=31
광주
부산 ✓

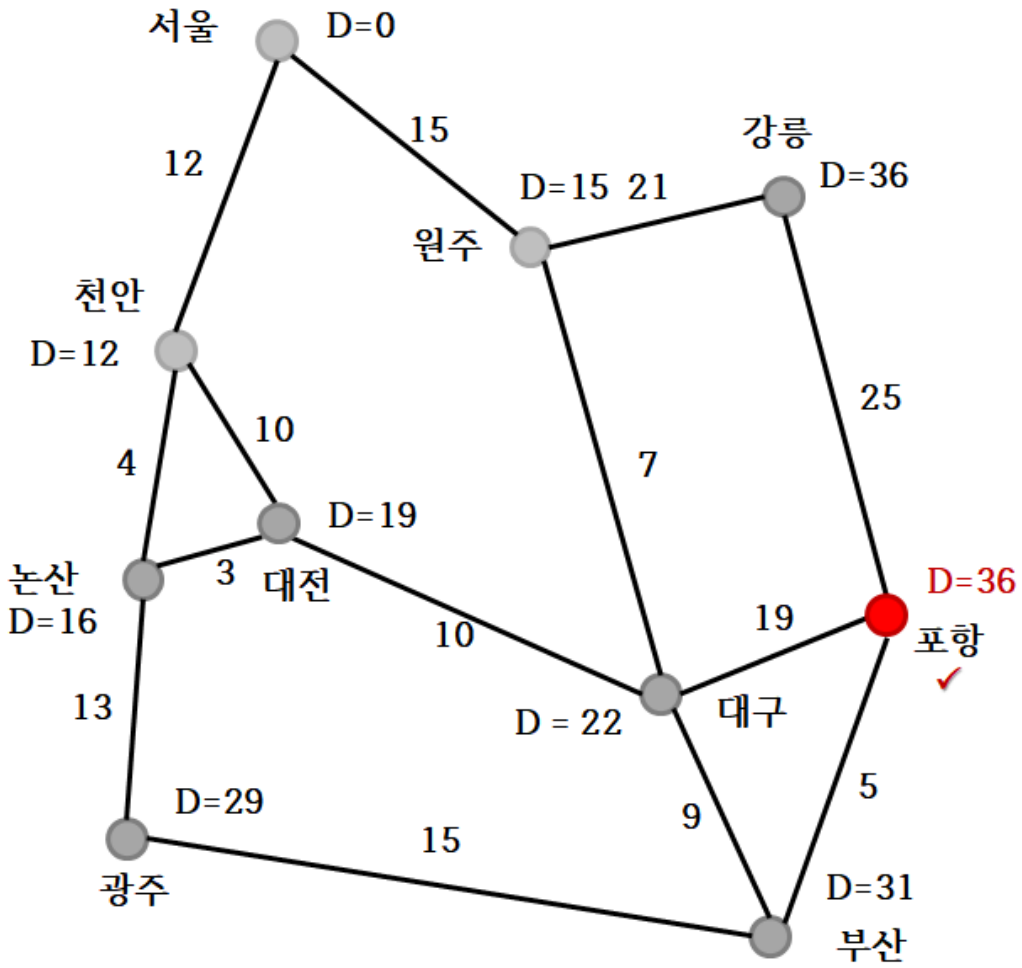
간선 완화


서울
D=0
15
12
강릉
D=36
D=15
21
원주
천안
D=12
25
10
4
7
D=19
D=41
D=31+5=36
논산
D=16
3
대전
19
10
포항
간선 완화
13
D = 22
대구
5
D=29
9
15
D=31
광주
부산

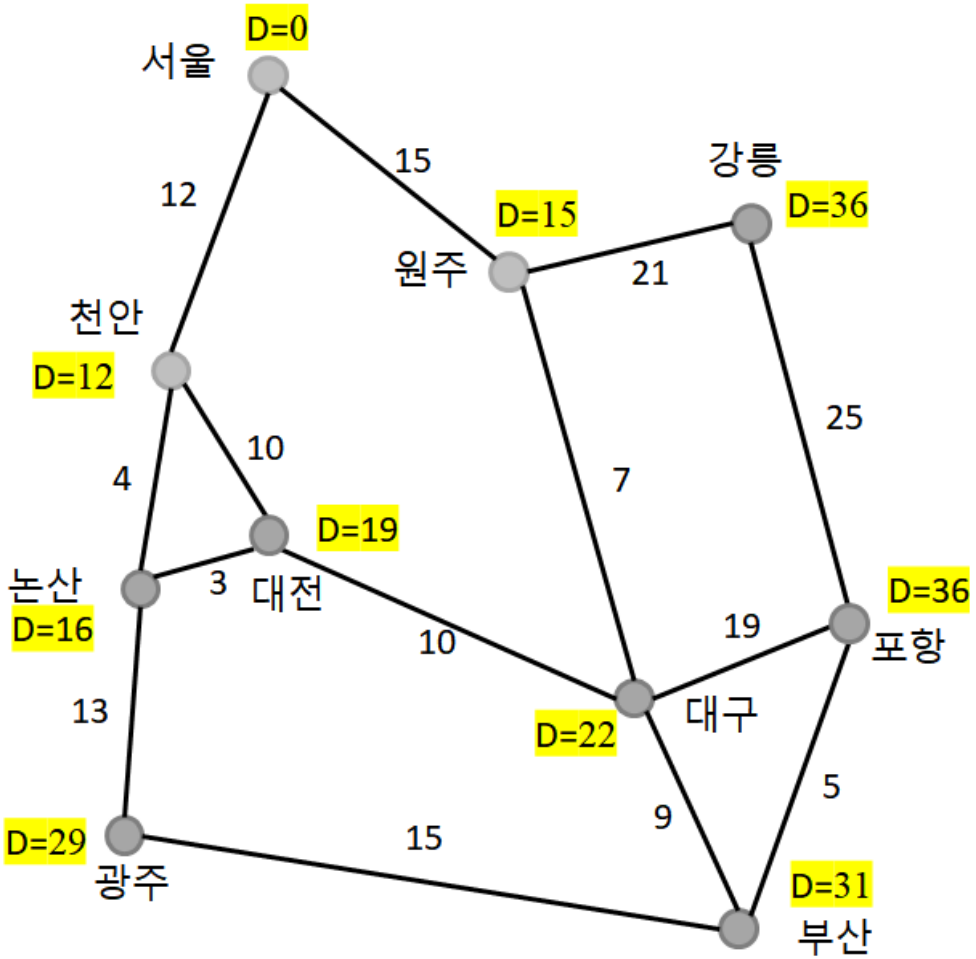
강릉 확정


서울
D=0
15
12
강릉 ✓
D=36
D=15 21
원주
천안
D=12
25
10
4
7
D=19
논산
D=16
3
대전
D=36
19
10
포항
13
D = 22
대구
5
D=29
9
15
D=31
광주
부산

포항 확정


서울
D=0
15
12
강릉
D=36
D=15 21
원주
천안
D=12
25
10
4
7
D=19
논산
D=16
3
대전
D=36
19
포항
10
13
D = 22
대구
5
D=29
9
15
D=31
광주
부산

ShortestPath 알고리즘의 수행 결과



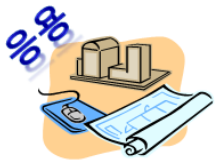
시간 복잡도

- while-루프가 (n-1)회 반복되고, 1회 반복될 때
  - line 3에서 최소의 D[v]를 가진 점 v_{min}을 찾는데 O(n) 시간 소요

    왜냐하면 배열 D에서 최솟값을 찾기 때문

  - line 4에서도 v_{min}에 연결된 점의 수가 최대 (n-1)개이므로, 각 D[w]를 갱신하는데 걸리는 시간은 O(n)

- ShotestPath의 시간 복잡도는
  - (n-1) x {O(n)+O(n)} = $O(n^2)$
  - 프림 알고리즘과 같이 최소 힙(Binary Heap)을 사용하면 O(mlogn), m은 간선 수. 따라서 간선 수가 O(n)이면 O(nlogn)



Applications

- 맵퀘스트 (MapQuest)와 구글 (Google) 맵
- 자동차 네비게이션
- 네트워크와 통신 분야
- 모바일 네트워크
- 산업 공학/경영 공학의 운영 (Operation) 연구
- 로봇 공학
- 교통 공학
- VLSI 디자인 분야 등

4.4 부분 배낭 문제

- 배낭 (Knapsack) 문제
  - n개의 물건이 각각 1개씩 있고,
  - 각 물건은 무게와 가치를 가지고 있으며,
  - 배낭이 한정된 무게의 물건들을 담을 수 있을 때,
  - 최대의 가치를 갖도록 배낭에 넣을 물건들을 정하는 문제

- 부분 배낭 (Fractional Knapsack) 문제
  - 물건을 부분적으로 담는 것을 허용
  - 그리디 알고리즘으로 해결

- 0-1 배낭 문제
  - 부분 배낭 문제의 원형으로 물건을 통째로 배낭에 넣어야 한다.
  - 동적 계획 알고리즘, 백트래킹 기법, 분기 한정 기법으로 해결

아이디어

- 부분 배낭 문제에서는 물건을 부분적으로 배낭에 담을 수 있으므로, 최적해를 위해서 '욕심을 내어' 단위 무게 당 가장 값나가는 물건을 배낭에 넣고, 계속해서 그 다음으로 값나가는 물건을 넣는다.

- 만일 물건을 '통째로' 배낭에 넣을 수 없으면, 배낭에 넣을 수 있을 만큼만 물건을 부분적으로 배낭에 담는다.

알고리즘

FractionalKnapsack

입력: n개의 물건, 각 물건의 무게와 가치, 배낭의 용량 C

출력: 배낭에 담은 물건 리스트 L과 배낭 속의 물건 가치의 합 v

1. 각 물건에 대해 단위 무게 당 가치를 계산한다.
2. 물건들을 단위 무게 당 가치를 기준으로 내림차순으로 정렬하고, 정렬된 물건 리스트를 S라고 하자.
3. L=∅, w=0, v=0

   // L은 배낭에 담을 물건 리스트, w는 배낭에 담긴 물건들의 무게의 합, v는 배낭에 담긴 물건들의 가치의 합
4. S에서 단위 무게 당 가치가 가장 큰 물건 x를 가져온다.

5. while w + (x의 무게) ≤ C
6. x를 L에 추가
7. w = w + (x의 무게)
8. v = v + (x의 가치)
9. x를 S에서 제거
10. S에서 단위 무게 당 가치가 가장 큰 물건 x를 가져온다.
11. If C-w > 0 // 배낭에 물건을 부분적으로 담을 여유가 있으면
12. 물건 x를 (C-w) 만큼만 L에 추가
13. v = v + (C-w) 만큼의 x의 가치
14. return L, v

수행 과정

- 배낭의 최대 용량 = 40그램

- 단위 무게 당 가치로 정렬: S=[백금, 금, 은, 주석]

물건	단위 그램당 가치
백금	6만원
금	5만원
은	4천원
주석	1천원

수행 과정

- 백금을 통째로 담는다.
- 배낭에 담긴 물건(들)의 무게 w = 10, 얻는 가치 v = 60

60만원

- 금을 통째로 담는다.
- 배낭에 담긴 물건(들)의 무게 w = 25,
  v = 60+ 75 = 135

75만원

수행 과정

- 은을 통째로 담으려 하지만
- 배낭에 담긴 물건(들)의 무게 w = 25 + 25 = 50이 되어 배낭 용량 초과

- 은을 40 -25 = 15 만큼만 담는다.
- 배낭에 담긴 물건(들)의 무게 w = 40,
  v = 135 + (0.4x15) = 141 만원

시간 복잡도

- Line 1: n개의 물건 각각의 단위 무게 당 가치를 계산하는 데는 O(n) 시간 소요
- Line 2: 물건의 단위 무게 당 가치에 대해서 정렬하기 위해 O(nlogn) 시간 소요
- Line 5~10: while-루프의 수행은 n번을 넘지 않으며, 루프 내부의 수행은 O(1) 시간 소요
- Line 11~14: 각각 O(1) 시간 소요

- 알고리즘의 시간 복잡도: O(n)+O(nlogn)+nxO(1)+O(1) = O(nlogn)

Applications

- 0-1 배낭 문제는 최소의 비용으로 자원을 할당하는 문제로서, 조합론, 계산이론, 암호학, 응용 수학 분야에서 기본적인 문제로 다뤄진다.
- '버리는 부분 최소화하는' 원자재 자르기
- 자산투자 및 금융 포트폴리오에서의 최선의 선택
- Merkle-Hellman 배낭 암호 시스템에 사용

4.5 집합 커버 문제

- 문제
  - n개의 원소를 가진 집합 U가 있고,
  - U의 부분집합들을 원소로 하는 집합 F가 주어질 때,
  - F의 원소들인 집합들 중에서 어떤 집합들을 선택하여 합집합하면 U와 같게 되는가?

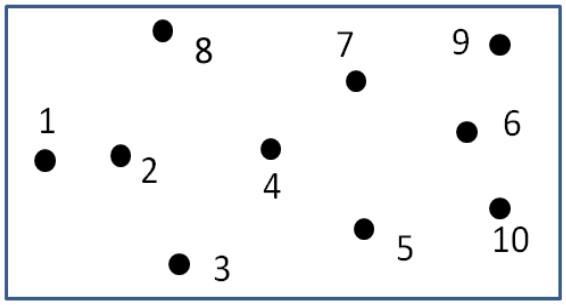
- 집합 커버 (Set Cover) 문제
  - 집합 F에서 선택하는 집합들의 수를 최소화하는 문제

신도시 학교 배치

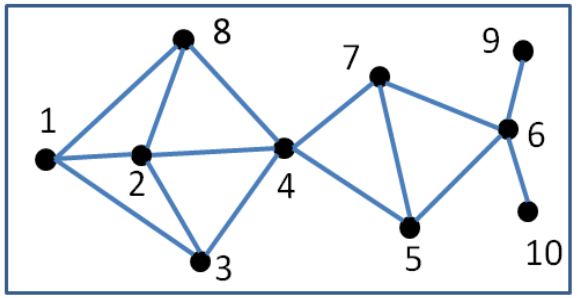
- 신도시를 계획하는 데 있어서 학교 배치의 예
  - 10개의 마을이 신도시에 만들어질 계획이다.
  - 다음 조건이 만족되도록 학교 위치를 선정해야 한다.

    학교는 마을에 위치해야 한다.

    등교 거리는 걸어서 15분 이내이어야 한다.
  - 어느 마을에 학교를 신설해야 학교의 수가 최소가 되는가?



10개 마을의 위치



등교 거리가 15분 이내인 마을 간의 관계

최적해

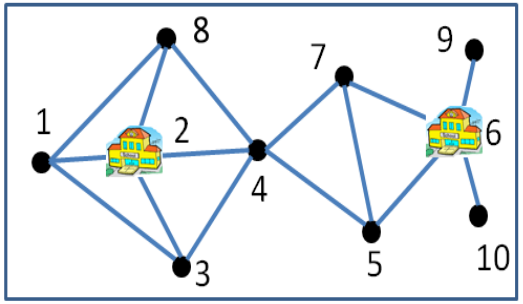
- 어느 마을에 학교를 신설해야 학교의 수가 최소가 되는가?
  - 2번 마을에 학교를 만들면

    1, 2, 3, 4, 8 마을의 학생들이 15분 이내에 등교 가능

    즉, 마을 1, 2, 3, 4, 8이 커버된다.
  - 6번 마을에 학교를 만들면

    마을 5, 6, 7, 9, 10이 커버된다.
  - 2번과 6번 마을에 학교를 배치하면 모든 마을이 커버된다.

    최소의 학교 수 = 2개



집합 커버

- 신도시 계획 문제를 집합 커버 문제로 변환

  U = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10} // 신도시의 마을 10개

  F = {S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9, S_{10}}

  // S_i는 마을 i에 학교를 배치했을 때 커버되는 마을의 집합

  S_1={1, 2, 3, 8} S_5={4, 5, 6, 7} S_9 ={6, 9}

  S_2={1, 2, 3, 4, 8} S_6={5, 6, 7, 9, 10} S_{10}={6, 10}

  S_3={1, 2, 3, 4} S_7={4, 5, 6, 7}

  S_4={2, 3, 4, 5, 7, 8} S_8={1, 2, 4, 8}

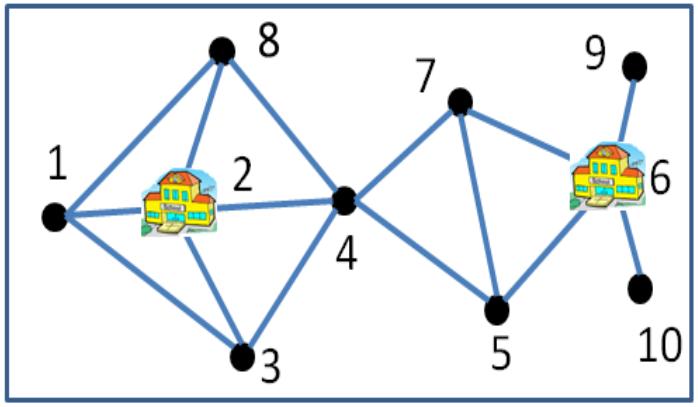
  - S_i 집합들 중에서 어떤 집합들을 선택해야 그들의 합집합이 U와 같은가?

    단, 선택된 집합의 수는 최소이어야

최적해

- $S_2 \cup S_6 = \{1, 2, 3, 4, 8\} \cup \{5, 6, 7, 9, 10\}$

  $= \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\} = U$



단순한 해결 방법

- 집합 커버 문제의 최적해는 어떻게 찾아야 할까?
  - F에 n개의 집합들이 있다고 가정해보자.

- 가장 단순한 방법
  - F에 있는 집합들의 모든 조합을 1개씩 합집합하여 U가 되는지 확인하고,
  - U가 되는 조합의 집합 수가 최소인 것을 찾는다.
  - F={S_1, S_2, S_3}일 경우 모든 조합

    S_1, S_2, S_3, $S_1 \cup S_2$, $S_1 \cup S_3$, $S_2 \cup S_3$, $S_1 \cup S_2 \cup S_3$

    집합이 1개인 경우 3개 = ${}_3C_1$

    집합이 2개인 경우 3개 = ${}_3C_2$

    집합이 3개인 경우 1개 = ${}_3C_3$

    총합은 3+3+1= 7 = 2^3-1 개

- n개의 원소가 있을 경우
  - 최대 (2^n-1)개를 검사하여야
  - n이 커지면 최적해를 찾는 것은 실질적으로 불가능

- 이를 극복하기 위한 방법
  - 최적해를 찾는 대신에 최적해에 근접한 근사해 (Approximation solution)를 찾는다.

알고리즘

SetCover

입력: U, F = $\{S_i\}$, i=1,⋯,n

출력: 집합 커버 C

1. C = ∅
2. while U ≠ ∅
3. U의 원소를 가장 많이 가진 집합 S_i를 F에서 선택
4. U = U – S_i
5. S_i를 F에서 제거하고, S_i를 C에 추가
6. return C

수행 과정

$U = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$

$F = \{S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9, S_{10}\}$

$S_1=\{1, 2, 3, 8\}$

$S_2=\{1, 2, 3, 4, 8\}$

$S_3=\{1, 2, 3, 4\}$

$S_4=\{2, 3, 4, 5, 7, 8\}$

$S_5=\{4, 5, 6, 7\}$

$S_6=\{5, 6, 7, 9, 10\}$

$S_7=\{4, 5, 6, 7\}$

$S_8=\{1, 2, 4, 8\}$

$S_9=\{6, 9\}$

$S_{10}=\{6, 10\}$

수행 과정

- U의 원소를 가장 많이 커버하는 집합

  S_4={2, 3, 4, 5, 7, 8} 선택

- $U = U - S_4$

  $= \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\} - \{2, 3, 4, 5, 7, 8\}$

  $= \{1, 6, 9, 10\}$

- S_4를 F에서 제거 $F = \{ S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9, S_{10}\} - \{S_4\} = \{S_1, S_2, S_3, S_5, S_6, S_7, S_8, S_9, S_{10}\}$

- S_4를 C에 추가. 즉, $C = \{S_4\}$

수행 과정

- U = {1, 6, 9, 10}을 가장 많이 커버하는 집합

  S_6 = {5, 6, 7, 9, 10} 선택

- U = U − S_6

  = {1, 6, 9, 10} − {5, 6, 7, 9, 10}

  = {1}

- S_6을 F에서 제거 F = { S_1, S_2, S_3, S_5, ~~S_6~~, S_7, S_8, S_9, S_{10}} − {S_6} = {S_1, S_2, S_3, S_5, S_7, S_8, S_9, S_{10}}

- S_6을 C에 추가. 즉, C = {S_4, S_6}

수행 과정

- U = {1}을 가장 많이 커버하는 집합

  S_1 = {1, 2, 3, 8} 선택

- U = U − S_1

  = {1} − {1, 2, 3, 8}

  = { }

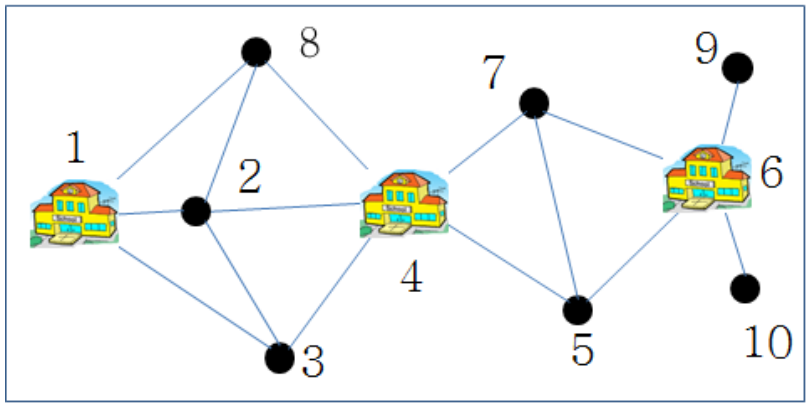
- S_1을 F에서 제거, F = { ~~S_1~~, S_2, S_3, S_5, S_7, S_8, S_9, S_{10}} − {S_1} = {S_2, S_3, S_5, S_7, S_8, S_9, S_{10}}

- S_1을 C에 추가. 즉, C = {S_4, S_6, S_1}

수행 과정

- Line 6: C={S_1, S_4, S_6} 리턴

SetCover 알고리즘의 최종해



시간 복잡도

- 먼저 while-루프가 수행되는 횟수는 최대 n회
  - 루프가 1회 수행될 때마다 집합 U의 원소 1개씩만 커버된다면, 최악의 경우 루프가 n번 수행되어야 하기 때문
- 루프가 1회 수행될 때
  - Line 3: U의 원소들을 가장 많이 포함하고 있는 집합 S를 찾으려면, 현재 남아있는 S_i들 각각을 U와 비교하여야
  - S_i들의 수가 최대 n이라면, 각 S_i와 U의 비교는 O(n) 시간이 걸리므로, line 3은 $O(n^2)$ 시간
  - 집합 U에서 집합 S_i의 원소를 제거하므로 O(n) 시간
  - S_i를 F에서 제거하고, S_i를 C에 추가하는 것은 O(1) 시간
- 시간 복잡도: $n \times O(n^2) = O(n^3)$

Applications

- 도시 계획 (City Planning)에서 공공 기관 배치하기
- 경비 시스템: 경비가 요구되는 장소의 CCTV 카메라의 최적 배치
- 컴퓨터 바이러스 찾기
- 대기업의 구매 업체 선정
- 기업의 경력 직원 고용
- 그 외에도 비행기 조종사 스케줄링, 조립 라인 균형화, 정보 검색 등에 활용

4.6 작업 스케줄링

- 작업 스케줄링 (Job Scheduling) 문제
  - 작업의 수행 시간이 중복되지 않도록 모든 작업을 가장 적은 수의 기계에 배정하는 문제
  - 학술대회에서 발표자들을 강의실에 배정하는 문제와 같다.

    발표= '작업', 강의실= '기계'

- 작업 스케줄링 문제에 주어진 문제 요소
  - 작업의 수

    입력의 크기이므로 알고리즘을 고안하기 위해 고려되어야 하는 직접적인 요소는 아니다.
  - 각 작업의 시작시간과 종료시간
  - 작업의 길이

    작업의 시작시간과 종료시간은 정해져 있으므로 작업의 길이도 주어진 것

작업 스케줄링

- 시작시간, 종료시간, 작업 길이에 대한 그리디 알고리즘
  - 빠른 시작시간 작업 우선 (Earliest start time first) 배정
  - 빠른 종료시간 작업 우선 (Earliest finish time first) 배정
  - 짧은 작업 우선 (Shortest job first) 배정
  - 긴 작업 우선 (Longest job first) 배정

- 위 4가지 중 첫 번째 알고리즘을 제외하고 나머지 3가지는 항상 최적해를 찾지 못함

작업 스케줄링 알고리즘

JobScheduling

입력: n개의 작업 $t_1, t_2, \cdots, t_n$

출력: 각 기계에 배정된 작업 순서

1. 시작 시간으로 정렬한 작업 리스트: L
2. while L $\neq \varnothing$
3. L에서 가장 이른 시작 시간 작업 t_i를 가져온다.
4. if t_i를 수행할 기계가 있으면
5. t_i를 수행할 수 있는 기계에 배정
6. else
7. 새 기계에 t_i를 배정
8. t_i를 L에서 제거
9. return 각 기계에 배정된 작업 순서

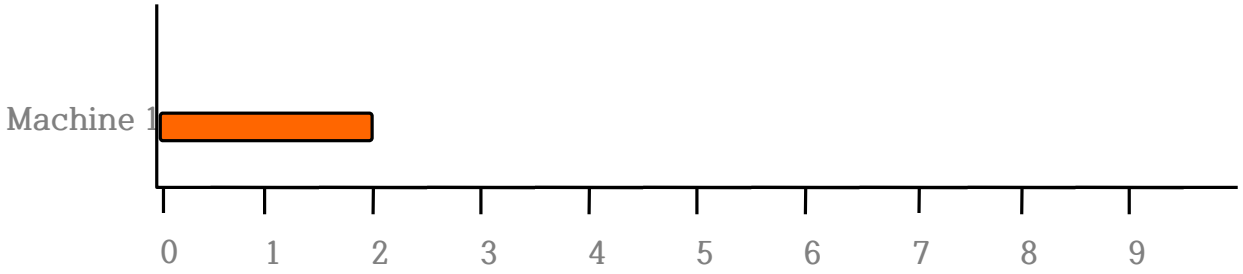
수행 과정

- t_1=[7,8], t_2=[3,7], t_3=[1,5], t_4=[5,9], t_5=[0,2], t_6=[6,8], t_7=[1,6]
  - [s, f]에서, s는 시작 시간, f는 종료 시간

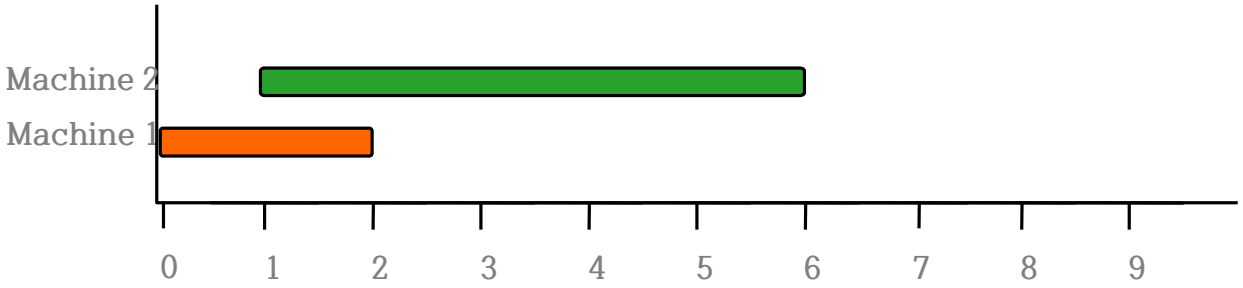
- 정렬: L = {[0,2], [1,6], [1,5], [3,7], [5,9], [6,8], [7,8]}

수행 과정

[0,2]

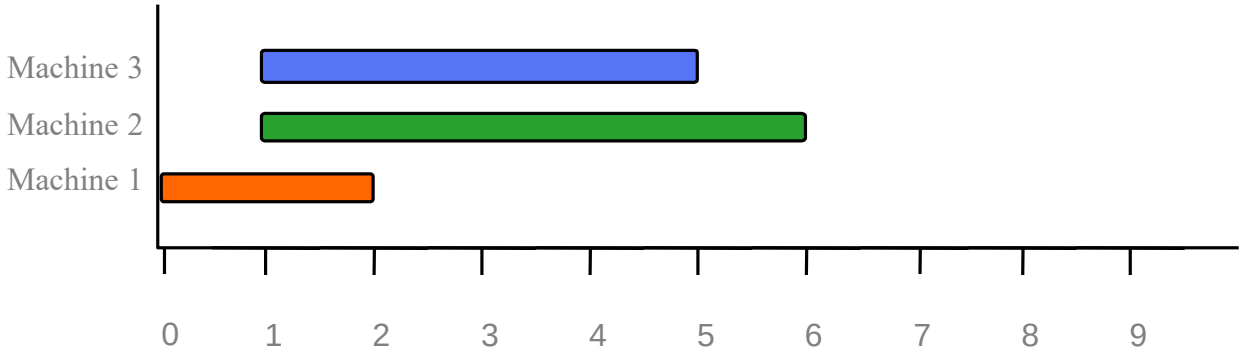


[0,2], [1,6]

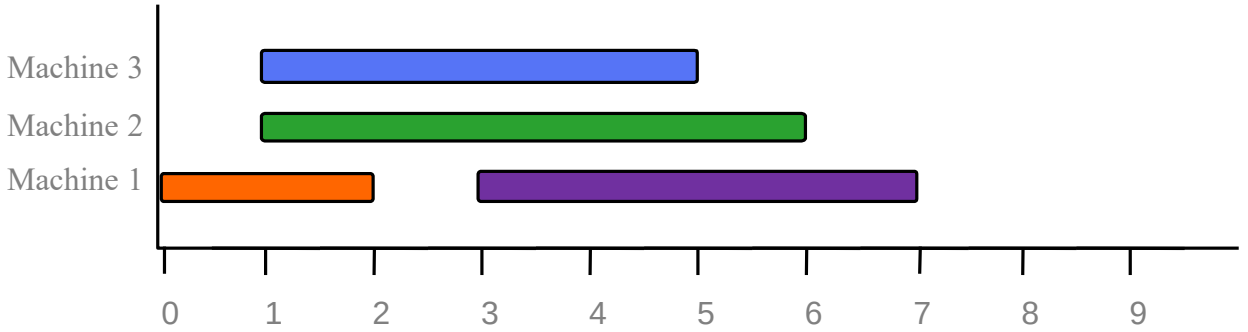


수행 과정

[0,2], [1,6], [1,5]

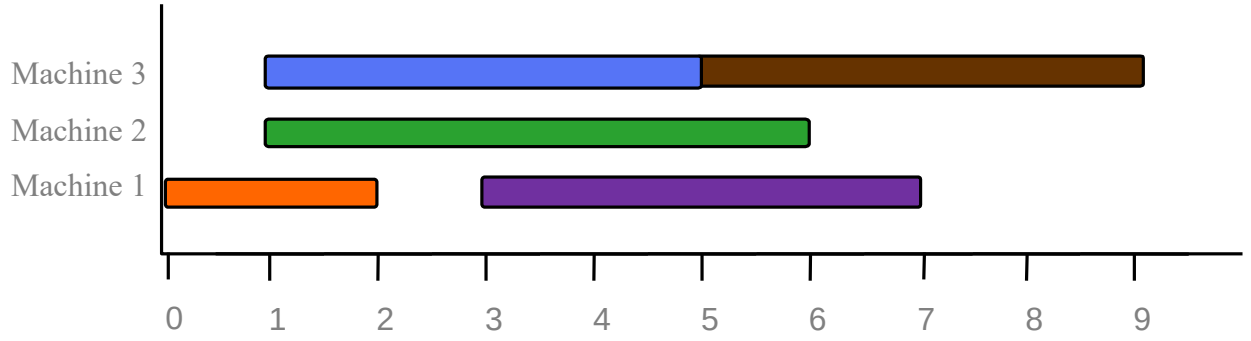


[0,2], [1,6], [1,5], [3,7]

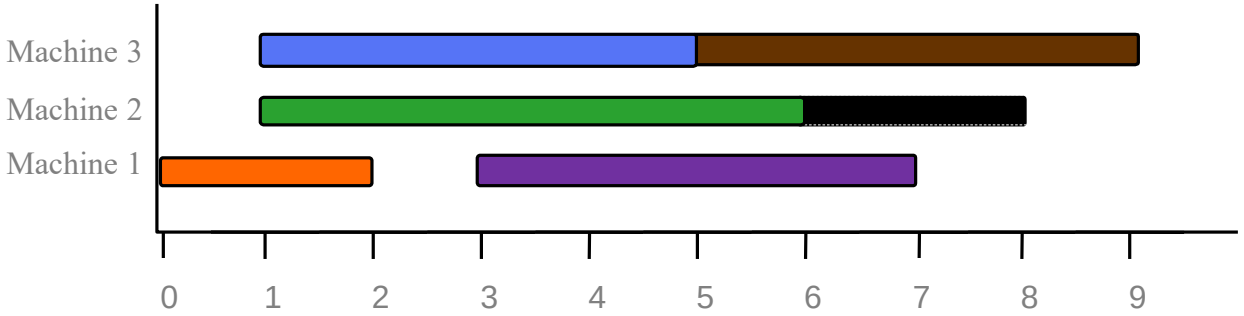


수행 과정

[0,2], [1,6], [1,5], [3,7], [5,9]

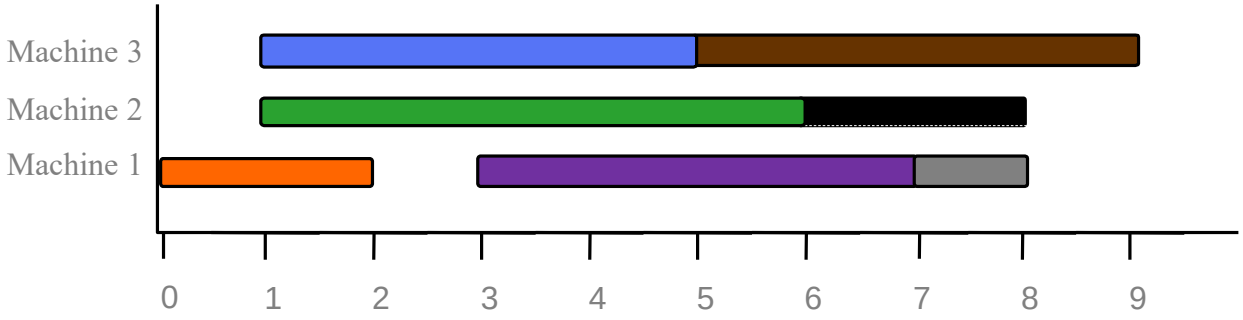


[0,2], [1,6], [1,5], [3,7], [5,9], [6,8]



수행 과정

[0,2], [1,6], [1,5], [3,7], [5,9], [6,8], [7,8]

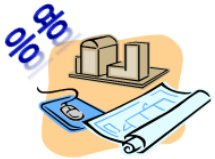


시간 복잡도

- Line 1: 정렬 시간 O(nlogn)

- while-루프
  - 작업을 L에서 가져다 수행 가능한 기계를 찾아서 배정하므로 O(m) 시간 소요, 단, m은 사용된 기계의 수
  - while-루프가 수행된 총 횟수는 n번이므로, line 2~9까지는 O(m) x n = O(mn) 시간 소요

- 시간 복잡도: O(nlogn)+O(mn)



Applications

- 비즈니스 프로세싱
- 공장 생산 공정
- 강의실/세미나 룸 배정
- 컴퓨터 태스크 스케줄링 등

4.7 허프만 압축

- 파일의 각 문자가 8 bit 아스키 (ASCII) 코드로 저장되면, 그 파일의 bit 수는 8 x (파일의 문자 수)
- 파일의 각 문자는 일반적으로 고정된 크기의 코드로 표현
- 고정된 크기의 코드로 구성된 파일을 저장하거나 전송할 때 파일의 크기를 줄이고, 필요시 원래의 파일로 변환할 수 있으면, 메모리 공간을 효율적으로 사용할 수 있고, 파일 전송 시간을 단축
- 파일의 크기를 줄이는 방법을 파일 압축 (file compression)이라 함

아이디어

- 허프만 (Huffman) 압축은 파일에 빈번히 나타나는 문자에는 짧은 이진 코드를 할당하고, 드물게 나타나는 문자에는 긴 이진 코드를 할당

- 허프만 압축 방법으로 변환시킨 문자 코드들 사이에는 접두부 특성 (prefix property)이 존재
  - 각 문자에 할당된 이진 코드는 어떤 다른 문자에 할당된 이진 코드의 접두부 (prefix)가 되지 않는다.
  - [예제] 문자 'a'에 할당된 코드가 '101'이라면, 모든 다른 문자의 코드는 '101'로 시작되지 않으며 또한 '1'이나 '10'도 아니다.

허프만 압축

- 접두부 특성의 장점은 코드와 코드 사이를 구분할 특별한 코드가 필요 없다.
  - 101#10#1#111#0#…에서 '#'가 인접한 코드를 구분 짓고 있는데, 허프만 압축에서는 이러한 특별한 코드 없이 파일을 압축/해제 가능

- 허프만 압축은 입력 파일에 대해 각 문자의 빈도수(문자가 파일에 나타나는 횟수)에 기반을 둔 이진 트리를 만들어서, 각 문자에 이진 코드를 할당
  - 이러한 이진 코드를 허프만 코드라고 함

HuffmanCoding

입력: 입력 파일의 n개의 문자에 대한 각각의 빈도수

출력: 허프만 트리

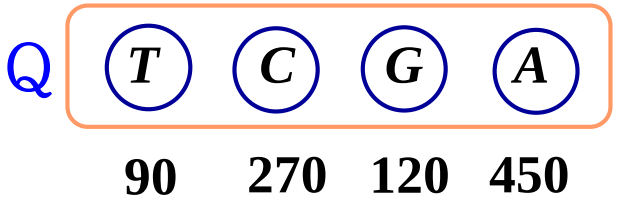
1. 각 문자 당 노드를 만들고, 그 문자의 빈도수를 노드에 저장
2. n 노드의 빈도수에 대해 우선 순위 큐 Q를 만든다.
3. `while` Q에 있는 노드 수 ≥ 2
4. 빈도수가 가장 적은 2개의 노드 (A와 B)를 Q에서 제거
5. 새 노드 N을 만들고, A와 B를 N의 자식 노드로 만든다
6. N의 빈도수 = A의 빈도수 + B의 빈도수
7. 노드 N을 Q에 삽입
8. `return` Q // 허프만 트리의 루트를 리턴

수행 과정

- 각 문자의 빈도수에 대해

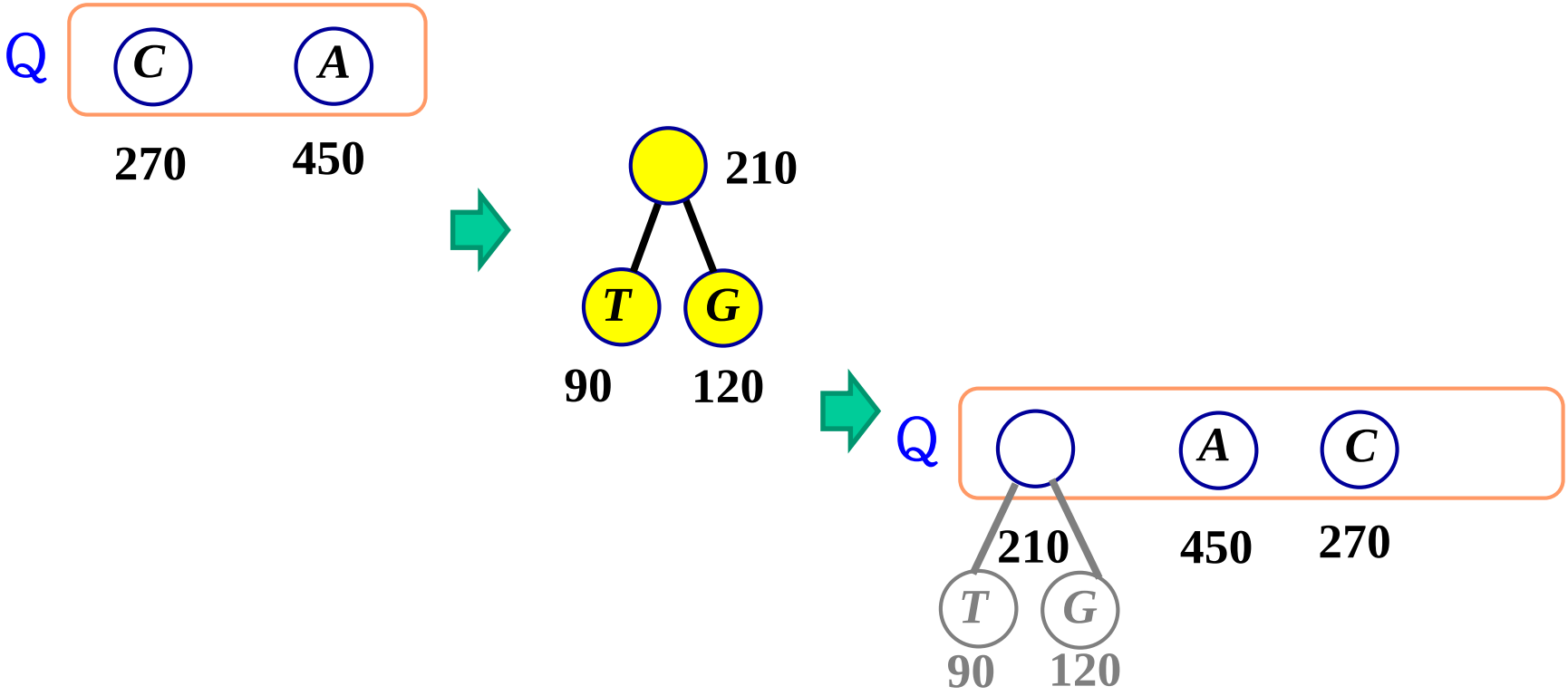
  A: 450 T: 90 G: 120 C: 270

- Line 2를 수행한 후의 Q
  - 우선 순위 큐 Q를 생성



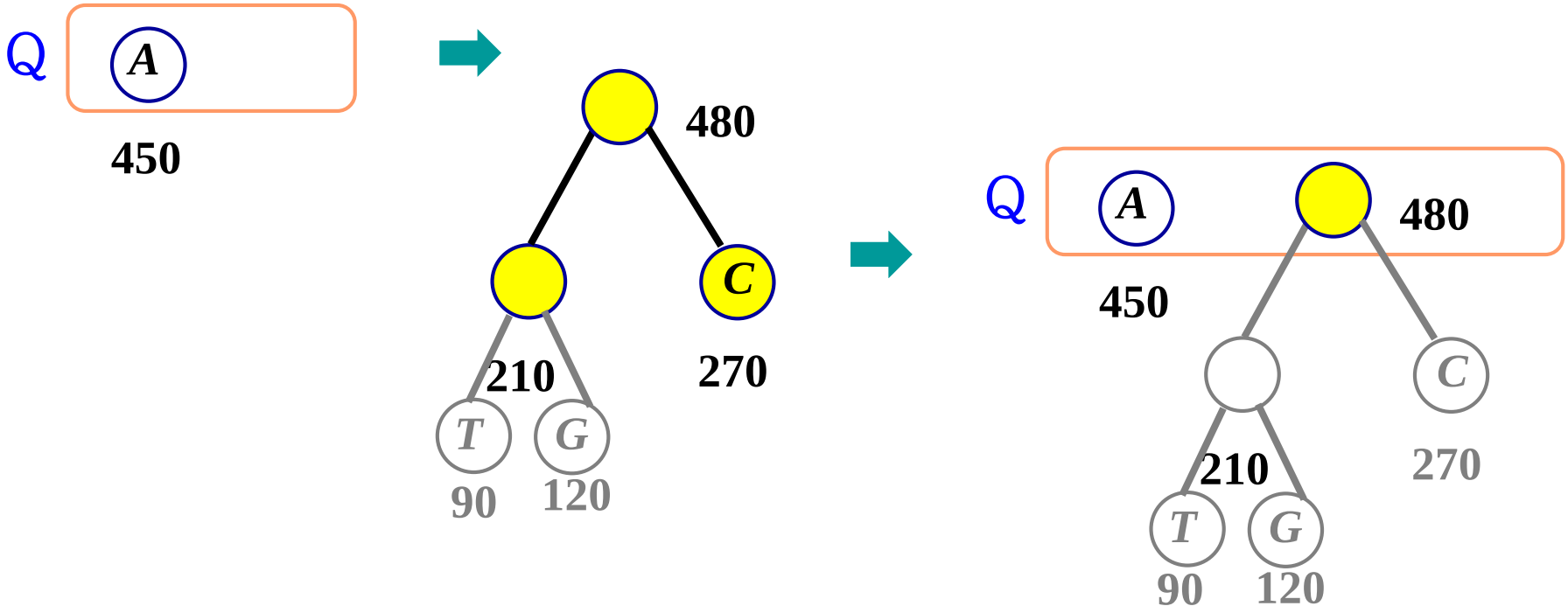
수행 과정

- Line 3: Q에서 'T'와 'G'를 제거한 후, 새 부모 노드를 Q에 삽입



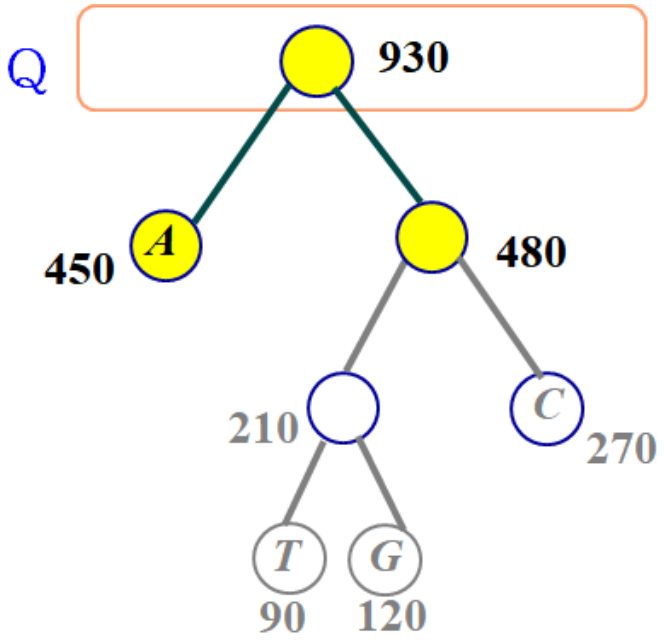
수행 과정

- Line 3: Q에서 'T'와 'G'의 부모 노드와 'C'를 제거한 후, 새 부모 노드를 Q에 삽입



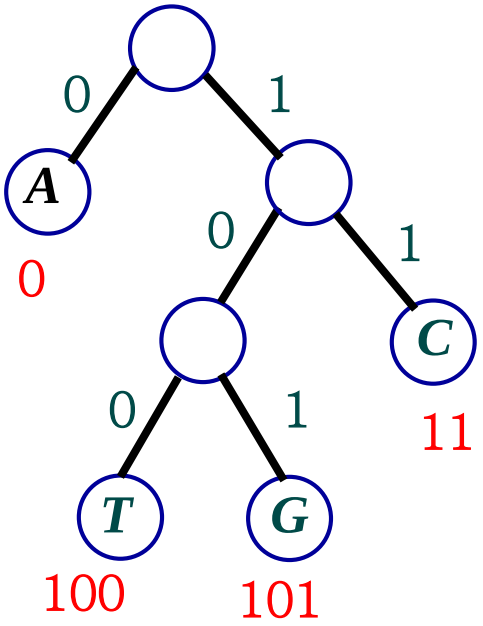
수행 과정

- Line 3: Q에서 'C'의 부모 노드와 'A'를 제거한 후, 새 부모 노드 Q에 삽입



수행 과정

- 반환된 트리를 살펴보면 각 이파리 (단말) 노드에만 문자가 있다.
  - 루트로부터 왼쪽 자식 노드로 내려가면 '0'을, 오른쪽 자식 노드로 내려가면 '1'을 부여하면서, 각 이파리에 도달할 때까지의 이진수를 추출하여 문자의 이진 코드를 얻는다.



압축률

- 예제에서 'A'는 '0', 'T'는 '100', 'G'는 '101', 'C'는 '11'의 코드가 각각 할당된다.
  - 할당된 코드들을 보면, 가장 빈도수가 높은 'A'가 가장 짧은 코드를 가지고, 따라서 루트의 자식이 되어 있고, 빈도수가 낮은 문자는 루트에서 멀리 떨어지게 되어 긴 코드를 가진다.
  - 이렇게 얻은 코드는 접두부 특성을 가진다.

- 압축된 파일의 bit 수
  - (450x1)+(90x3)+(120x3)+(270x2) = 1,620 bits

- 아스키 코드로 된 파일 크기
  - (450+90+120+270)x8 = 7,440 bits

- 파일 압축률
  - (1,620/7,440)x100 = 21.8%이며, 원래의 약 1/5 크기로 압축

복호화

- 예제에서 얻은 허프만 코드로 아래의 압축된 부분에 대해서 압축을 해제하여 보자.

10110010001110101010100

101 / 100 / 100 / 0 / 11 / 101 / 0 / 101 / 0 / 100

⇨ G T T A C G A G A T

시간 복잡도

- Line 1: n개의 노드를 만들고, 각 빈도수를 노드에 저장하므로 O(n) 시간

- Line 2: n개의 노드로 우선순위 큐 Q를 만든다.
  - 여기서 우선 순위 큐로서 이진 힙 자료구조를 사용하면 O(n) 시간

시간 복잡도

- Line 3~7
  - 최소 빈도수를 가진 노드 2개를 Q에서 제거하는 힙의 삭제 연산과 새 노드를 Q에 삽입하는 연산을 수행하므로 O(logn) 시간 소요
  - while-루프는 (n-1)번 반복

    왜냐하면 루프가 1번 수행될 때마다 Q에서 2개의 노드를 제거하고 1개를 Q에 추가하기 때문
  - (n-1) x O(logn) = O(nlogn)

- Line 8
  - 트리의 루트를 반환하는 것이므로 O(1) 시간

- 시간 복잡도는 O(n)+O(n)+O(nlogn)+ O(1) = O(nlogn)

Applications

- 팩스(FAX), 대용량 데이터 저장, 멀티미디어 (Multimedia), MP3 압축 등에 활용

- 정보 이론 (Information Theory) 분야에서 엔트로피 (Entropy)를 계산하는데 활용
  - 이는 자료의 불특정성을 분석하고 예측하는데 이용

요약

- 그리디 알고리즘은 (입력) 데이터 간의 관계를 고려하지 않고 수행 과정에서 '욕심내어' 최적값을 가진 데이터를 선택하며, 선택한 값들을 모아서 문제의 최적해를 찾는다.

- 그리디 알고리즘은 문제의 최적해 속에 부분 문제의 최적해가 포함되어 있고, 부분 문제의 해 속에 그 보다 작은 부분 문제의 해가 포함되어 있다. 이를 최적 부분 구조 (Optimal Substructure) 또는 최적성 원칙 (Principle of Optimality) 이라고 한다.

- 동전 거스름돈 문제를 해결하는 가장 간단한 방법은 남은 액수를 초과하지 않는 조건하에 가장 큰 액면의 동전을 취하는 것이다. 단, 일반적인 경우에는 최적해를 찾으나 항상 최적해를 찾지는 못한다.

요약

- 크러스컬의 알고리즘은 가중치가 가장 작으면서 사이클을 만들지 않는 간선을 추가시키어 트리를 만든다. 시간 복잡도는 $O(m \log m)$. 단, m은 그래프의 간선의 수

- 프림의 알고리즘은 최소의 가중치로 현재까지 만들어진 트리에 연결되는 간선을 트리에 추가시킨다. 시간 복잡도는 $O(n^2)$

- 다익스트라의 알고리즘은 출발점으로부터 최단 거리가 확정되지 않은 점들 중에서 출발점으로부터 가장 가까운 점을 추가하고, 그 점의 최단 거리를 확정한다. 시간 복잡도는 $O(n^2)$

요약

- 부분 배낭(Fractional Knapsack) 문제에서는 단위 무게 당 가장 값나가는 물건을 계속해서 배낭에 담는다. 마지막엔 배낭에 넣을 수 있을 만큼만 물건을 부분적으로 배낭에 담는다. 시간 복잡도는 $O(n\log n)$

- 집합 커버(Set Cover) 문제는 근사(Approximation) 알고리즘을 이용하여 근사해를 찾는 것이 보다 실질적이다. U의 원소들을 가장 많이 포함하고 있는 집합을 항상 F에서 선택한다. 시간 복잡도는 $O(n^3)$

- 작업 스케줄링(Job Scheduling) 문제는 빠른 시작시간 작업 먼저(Earliest start time first) 배정하는 그리디 알고리즘으로 최적해를 찾는다. 시간 복잡도는 $O(n\log n)+O(mn)$. n은 작업의 수이고, m은 기계의 수

요약

- 허프만 압축은 파일에 빈번히 나타나는 문자에는 짧은 이진 코드를 할당하고, 드물게 나타나는 문자에는 긴 이진 코드를 할당
- n이 문자의 수일 때, 시간 복잡도는 O(nlogn)