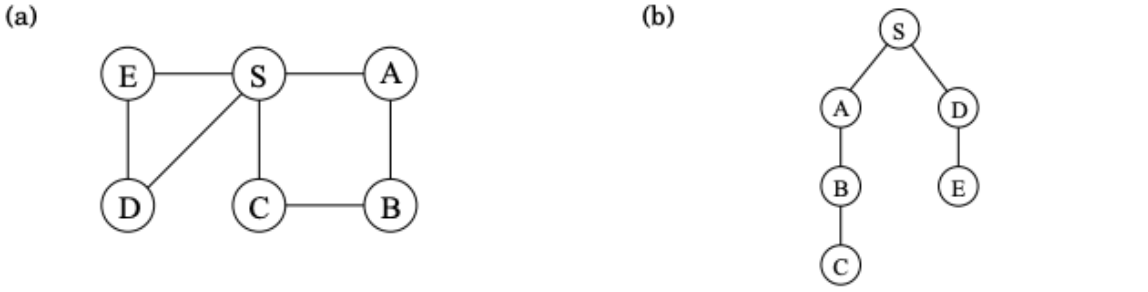
Chap 4. Paths in graphs

4.1 Distances

Chap3 에서 본 DFS 로 - starting point 로부터 갈 수 있는 모든 vertices 들을 identify 가능
- 이러한 vertices 에 대한 path 를 찾을 수 O

However, 이렇게 구한 path 가 항상 economical 한 건 아니다
: 아래 그림을 보면 graph 에서 S 와 C 의 거리는 1 인데, DFS tree 에서 보면 길이가 3 임

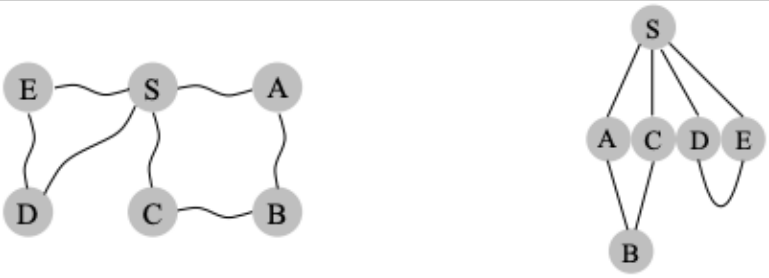
Figure 4.1 (a) A simple graph and (b) its depth-first search tree.



→ 이번 Chap4 에서는 graph 에서 shortest path 를 찾는 알고리즘에 대해…

Distance between two nodes 가 의미하는 것 = 두 노드 사이에 가장 짧은 path

Figure 4.2 A physical model of a graph.



→ 한 vertex 를 손으로 잡고 나머지를 늘이는 것처럼… (위 그림에서는 s 를 잡고 늘인)
→ S 와 B 의 distance(the shortest path)는 2
→ physical model 에서 보면, edge(D, E)는 아무 역할이 없다 = slack 으로 남음

4.2 Breadth-first search

Figure 4.2 를 보면 graph 가 layer 로 나뉘어져 있다
1) S 2) S 와 distance 가 1 인 노드들 3) S 와 distance 가 2 인 노드들
→ 이때 distance 가 0,1,2,…,d 인인 노드들을 pick out 했다면, d+1 인 노드들 찾는 건 쉽다
: distance 가 d 인 노드들에 adjacent 하면서, 아직 보지 않은 노드가 d+1 인 노드
→ 이 개념을 이용해서 2 개 layers 가 동작하는 방식으로 iterative algorithm
: fully identified layer d 가 있고, 이 layer 의 neighbors 을 scan 하면서 discover 하게 되는 d+1 layer

→ BFS 는 여기서 말하는 layer by layer 개념을 이용한 알고리즘

Figure 4.3 Breadth-first search.

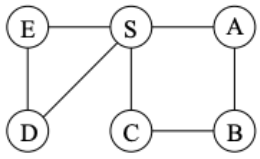
```
procedure bfs(G, s)
Input:    Graph G = (V, E), directed or undirected; vertex s ∈ V
Output:   For all vertices u reachable from s, dist(u) is set
          to the distance from s to u.

for all u ∈ V:
   dist(u) = ∞

dist(s) = 0
Q = [s]  (queue containing just s)
while Q is not empty:
   u = eject(Q)
   for all edges (u, v) ∈ E:
      if dist(v) = ∞:
         inject(Q, v)
         dist(v) = dist(u) + 1
```

- DFS 에서는 stack 이었다면, BFS 는 queue
- 시작 노드인 s 로 queue 를 초기화, 이때 이 노드의 거리는 0 (start node, end node 둘 다 s 니까)
- eject = queue 의 첫번째 원소 꺼내는
- eject(Q)를 통해 queue 의 첫번째 노드를 꺼내서 그 노드와 인접한 모든 노드들 중 아직 보지 않은 노드들을 queue 에 넣는다. 이때 queue 에 들어가는 노드들의 distance 는 +1

→ 위의 알고리즘의 example 을 보자면

Figure 4.1 (a) A simple graph and (b

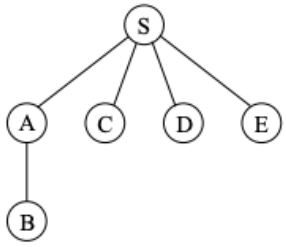
(a)



→

Figure 4.4 The result of breadth-first search on the graph of Figure 4.1.

Order of visitation	Queue contents after processing node
	$[S]$
S	$[A\ C\ D\ E]$
A	$[C\ D\ E\ B]$
C	$[D\ E\ B]$
D	$[E\ B]$
E	$[B]$
B	$[\,]$



→ DFS 와는 달리, BFS 는 모든 path 가 shortest path 이다

→ 그래서, BFS tree 를 shortest-path tree 라고 하기도 한다

*** Correctness and efficiency**

BFS 의 basic intuition = layer by layer + checking as-yet-unseen adjacent nodes

- Correctness

BFS 의 correctness 를 확인하기 위해서는 아래 3 가지를 확인해야함

For each d = 0,1,2,⋯, there is a moment at which

1) all nodes at distance <= d from s have their distance correctly set
2) all other nodes have their distances set to ∞
3) the queue contains exactly the nodes at distance d

- Efficiency

Overall running time = linear, O(|V|+|E|) → DFS 와 동일!

: 각 vertex 들은 한번씩 queue 에 존재하게 된다 = 2|V| queue operation (inject, eject)

: directed graph 인 경우 모든 edge 를 두번씩, undirected graph 인 경우 한번씩 = O(|E|)

*** DFS vs. BFS**

- DFS

: graph 에 깊게 들어가서 더이상 탐색할 노드가 없을 때 retreating

: 유용하기도 하지만, 실제로는 가까운데 더 길고 복잡한 route 를 taking 할 수도 있다

- BFS

: starting point 에서 distance 가 증가하는 순서대로 vertices 를 방분

: DFS 보다 더 넓고, shallow 한 search

: DFS 와 거의 같은 코드를 사용해서 stack 대신 queue 를 사용하여 구현할 수 있다

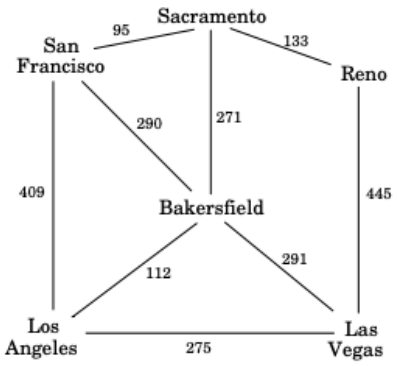
: start node 와 의 distance 를 생각하기 때문에, s 와 연결되어 있지 않는 노드들은 고려하지 않는다

4.3 Lengths on edges

위에서 BFS 는 모든 edge 들을 같은 길이로 취급했지만, 이런 경우는 매우 드뭄

아래와 같이 edge 의 길이가 주어져 있고, 가장 빠른 길을 찾는 경우

Figure 4.5 Edge lengths often matter.



4.4 Dijkstra's algorithm

4.4.1 An adaptation of breadth-first search

위에서 배운 BFS 는 unit length 의 edge 를 가진 어떤 그래프에서든 shortest path 를 찾을 수 있다

→ edge length 가 l_e인 좀 더 general 한 graph $G = (V, E)$에서는?

*** A more convenient graph**

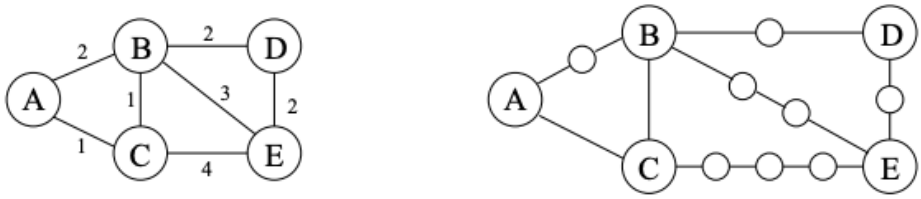
BFS 로 할 수 있게끔 G 를 변환하는 simple trick

: G 의 long edge 들을 dummy node 를 추가하는 방식으로 unit-length piece 들로 쪼갬

: Figure 4.6 처럼 기존 graph G 를 다음의 방법으로 새로운 G'으로 만듦

For any edge $e = (u, v)$ of E, replace it by l_e edges of length 1, by adding $l_e - 1$ dummy nodes between u and v.

Figure 4.6 Breaking edges into unit-length pieces.



: 이렇게 변환된 G'은 우리가 알고 싶은 vertices V 들을 모두 포함하고 있으며 기존 G 에서와 동일한 distance 를 가지게 한다

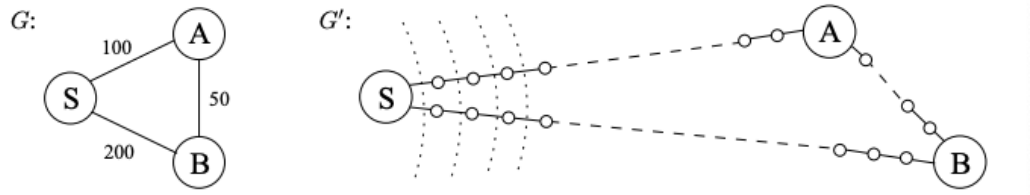
: G'의 모든 edges 는 unit length 이기 때문에 G'에 BFS 를 함으로써 G 에 대한 distance 를 계산할 수 있다

*** Alarm clocks**

- 위의 dummy node 를 추가하는 방법은 efficiency 측면에서 좋지 않음

  : G 에 long edges 가 많아서, G'로 변환하는 과정에서 dummy node 가 많이 추가되는 경우 BFS 과정에서 고려하지 않아도 되는 노드들의 거리도 계산하기 때문에 efficiency 가 좋지 않다.

Figure 4.7 BFS on G' is mostly uneventful. The dotted lines show some early "wavefronts."



: Figure 4.7 에서 G'의 s 로부터 시작해서 BFS 를 한다면

(one unit of distance per minute = 분당 one unit distance 라고 가정)

: BFS 를 시작하고 99 분 동안은 S – A, S – B 를 따라서 tediously(지루하게) 진행된다

→ 이 지루한 과정 동안은 재우고, real node(G 에 존재하는 node)에 도달할 때만 알람으로 깨운다면?

- **Alarm clock algorithm**

: 처음 시작할 때, 알람 두개 세팅

A 에 대해 T = 100 으로, B 에 대해 T = 200 으로 (edge 의 length)

: 이때 이 T 값들은 estimated times of arrival 이므로 현재 traversed 중인 edge 들을 중심으로 변경된다

: A 를 찾기 위해 재웠다가 T = 100 이 되면 깨움

: 이때(알람이 울려서 A 를 찾았을 때), estimated time of arrival for B 는 T = 150 으로 변경된다

- More generally…

: 어떤 순간에서도 BFS 는 G 의 edge 에서 이동하고 있고, 모든 end point node 에는 해당 노드에 도달하는 예상 시간에 알람이 울리도록 설정되어 있다

: 이 설정한 예상시간은 실제보다 overestimates 될 수도 있다

(앞의 예시처럼 처음에 B 에 대한 알람을 200 이라고 했지만, 실제로는 A 를 거쳐 B 로 가는 더 빠른 길을 찾을 수도 있기 때문에)

: 하지만, 어떠한 경우에도 알람이 울리기 전에 interesting 한 일은 일어날 수 없다

: 알람이 울리면 그 노드에서 나갈 수 있는 새로운 edge 로 advancing 할 수 있으며 그 edge 의 end point 에 대한 알람도 설정해줘야 한다

- **Alarm clock algorithm**

- Set an alarm clock for node s at time 0.
- Repeat until there are no more alarms:

  Say the next alarm goes off at time T, for node u. Then:

  - The distance from s to u is T.
  - For each neighbor v of u in G:
    - If there is no alarm yet for v, set one for time $T + l(u, v)$.
    - If v's alarm is set for later than $T + l(u, v)$, then reset it to this earlier time.

*** Dijkstra's algorithm**

Alarm clock algorithm 은 positive integral edge length 를 가진 어떤 그래프에서도 distance 를 계산할 수 있다

그럼, 이 알고리즘에서 알람을 어떻게 구현할건가?

→ 적절한 data structure = priority queue (heap 으로 구현되는)

이유 1) node 들을 연관된 numeric key value 와 함께 set 으로 저장할 수 있다

이유 2) 아래 4 가지 연산들이 가능

1. Insert – add a new element to the set → set alarm
2. Decrease-key – 특정 element 의 key 값을 감소시킬 수 있다 → set alarm
3. Delete-min – 가장 작은 key 을 가지는 element 를 return 하고 delete → 다음 알람을 어디에 설정할지
4. Make-queue – elements 의 key 값을 통해 elements 의 priority queue 를 생성

- 알고리즘 pseudo code

Figure 4.8 Dijkstra's shortest-path algorithm.

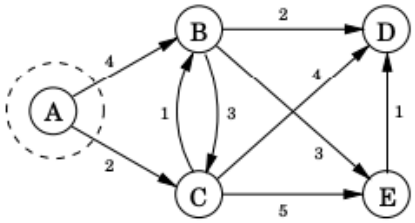
```
procedure dijkstra(G, l, s)
Input:    Graph G = (V, E), directed or undirected;
          positive edge lengths {l_e : e ∈ E}; vertex s ∈ V
Output:   For all vertices u reachable from s, dist(u) is set
          to the distance from s to u.

for all u ∈ V:
   dist(u) = ∞
   prev(u) = nil
dist(s) = 0

H = makequeue(V)   (using dist-values as keys)
while H is not empty:
   u = deletemin(H)
   for all edges (u, v) ∈ E:
      if dist(v) > dist(u) + l(u, v):
         dist(v) = dist(u) + l(u, v)
         prev(v) = u
         decreasekey(H, v)
```

dist(u): u node 에 대한 alarm clock setting

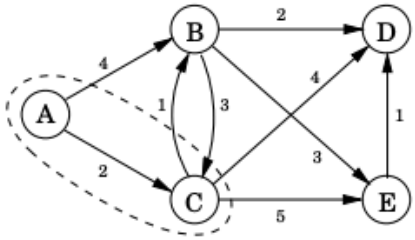
prev(u): s(시작노드)에서 u node 까지의 최단 경로에서 u node 직전에 있는 노드를 저장하는 = back-pointer

: 이 정보를 이용해서 최단 경로를 쉽게 reconstruct 할 수 있다

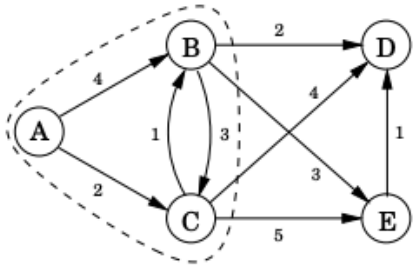
Figure 4.9 A complete run of Dijkstra's algorithm, with node A as the starting point. Also shown are the associated `dist` values and the final shortest-path tree.



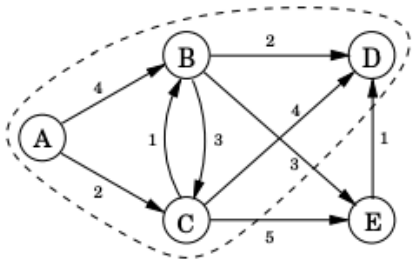
A: 0	D: ∞
B: 4	E: ∞
C: 2	



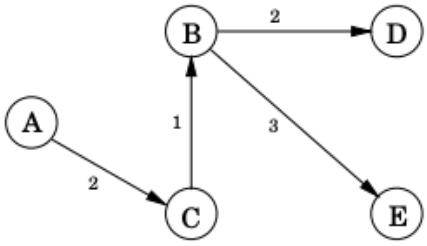
A: 0	D: 6
B: 3	E: 7
C: 2	



A: 0	D: 5
B: 3	E: 6
C: 2	



A: 0	D: 5
B: 3	E: 6
C: 2	



→ 위의 알고리즘을 적용하여 최단경로를 찾은 예

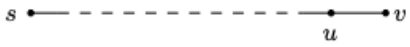
4.4.2 An alternative derivation

Shortest path 를 계산하기 위해, starting node 에서 바깥쪽으로 확장해 나가면서 shortest path 와 distance 를 알 수 있는 그래프의 영역을 확장시켜 나간다.

좀 더 구체적으로…

- “known region”이 s 를 포함하는 vertices R 의 subset 이면, 그 다음으로 “known region”에 추가될 노드는 영역 바깥 쪽에 있는 노드 중 가장 가까운 노드여야 한다.
- 이렇게 추가되는 노드를 v 라고 하면, 우리는 어떻게 이 v 를 알 수 있을까?

: 아래처럼 s 에서 v 까지의 shortest path 에서 v 바로 이전의 노드가 u 라고 하면

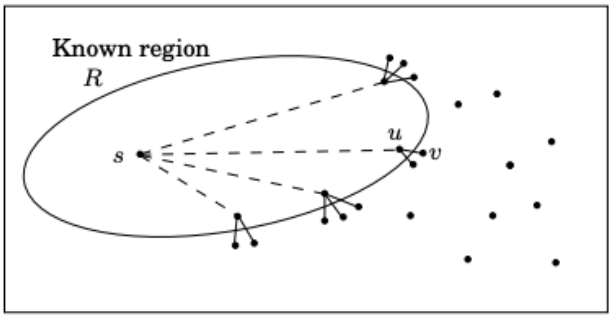


: 모든 edge length 가 양수라는 가정하에 u 는 v 보다 s 에 더 가까울 거다 = u 는 R 에 포함되어 있음

: 그래서, s 에서 v 로 가는 최단 경로는 a known shortest path extended by a single edge

: 근데 아래 그림처럼 많은 single-edge extensions 이 있을 수도 있다

Figure 4.10 Single-edge extensions of known shortest paths.



→ 이럴땐, the shortest of these extended paths! (뻗어 나가는 edge 들 중에 가장 짧은걸로)

: 현재 shortest path set 의 확장을 통해 R 을 확장시키는 알고리즘

```
Initialize dist(s) to 0, other dist(·) values to ∞
R = { } (the ''known region'')
while R ≠ V:
    Pick the node v ∉ R with smallest dist(·)
    Add v to R
    for all edges (v, z) ∈ E:
        if dist(z) > dist(v) + l(v, z):
            dist(z) = dist(v) + l(v, z)
```

4.4.3 Running time

Figure 4.8 Dijkstra's shortest-path algorithm.

```
procedure dijkstra(G, l, s)
Input:    Graph G = (V, E), directed or undirected;
          positive edge lengths {l_e : e ∈ E}; vertex s ∈ V
Output:   For all vertices u reachable from s, dist(u) is set
          to the distance from s to u.

for all u ∈ V:
   dist(u) = ∞
   prev(u) = nil
dist(s) = 0

H = makequeue(V)   (using dist-values as keys)
while H is not empty:
   u = deletemin(H)
   for all edges (u, v) ∈ E:
      if dist(v) > dist(u) + l(u, v):
         dist(v) = dist(u) + l(u, v)
         prev(v) = u
         decreasekey(H, v)
```

: Dijkstra algorithm 은 구조적으로 BFS 와 동일하지만, 우선순위 queue 가 계산량을 더 많이 요구하기 때문에 BFS 보다 느림

: total operation = |V| deletemin and |V|+|E| insert/decreasekey
(makequeue 에서 최대 |V|insert 연산)

: 시간복잡도는 어떻게 구현하냐에 따라 다른데, binary heap 을 사용하면 O((|V|+|E|)log|V|)

4.5 Priority queue implementations

4.5.1 Array

: priority queue 의 가장 간단한 구현방법

: 모든 vertices 의 key 값들의 배열(unordered)

: 처음에 이 배열값들은 ∞로 초기화된다.

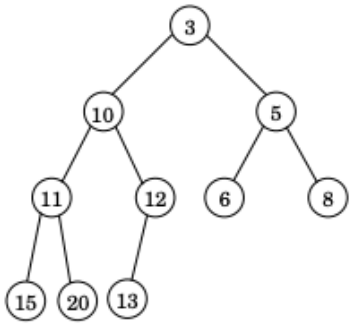
: insert, decreasekey 는 O(1)로 속도가 빠름
→ key 값만 조정하면 되니까

: deletemin 은 리스트의 linear-time scan 이 필요하다

4.5.2 Binary heap

Elements(node)은 각 lavel 에서 왼쪽에서 오른쪽으로 값들이 채워지는 complete binary tree 에 저장된다

* 이때, tree 의 모든 node 의 key 값이 하위노드의 key 값보다 작거나 같아야 한다.

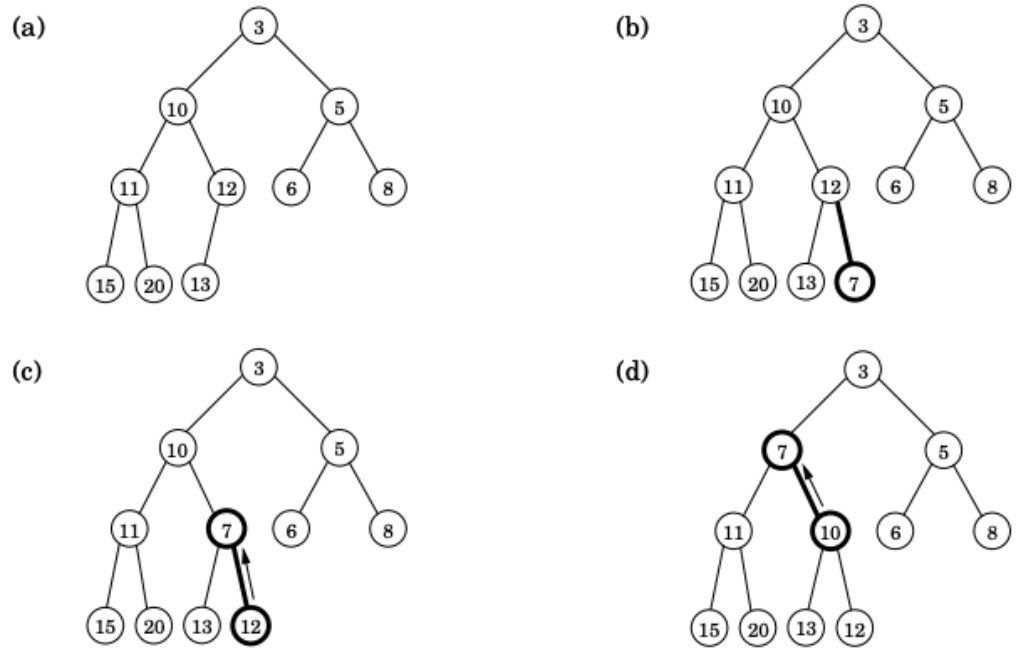


→ root 는 언제나 가장 작은 값을 가진다

- insert 연산

: 새로운 element 를 tree 의 가장 아래쪽에 위치 시킨 후 bubble up 을 수행한다

* bubble up : 부모보다 큰 값을 가지면 부모와 자신을 swap 하는 것 → swap 되지 않을 때까지 반복



→ 새로운 element 7 이 들어왔을 때, bubble up 하는 과정

→ swap 하는 횟수는 최대 $\lfloor \log_2 n \rfloor$ (tree height, n=element 개수)

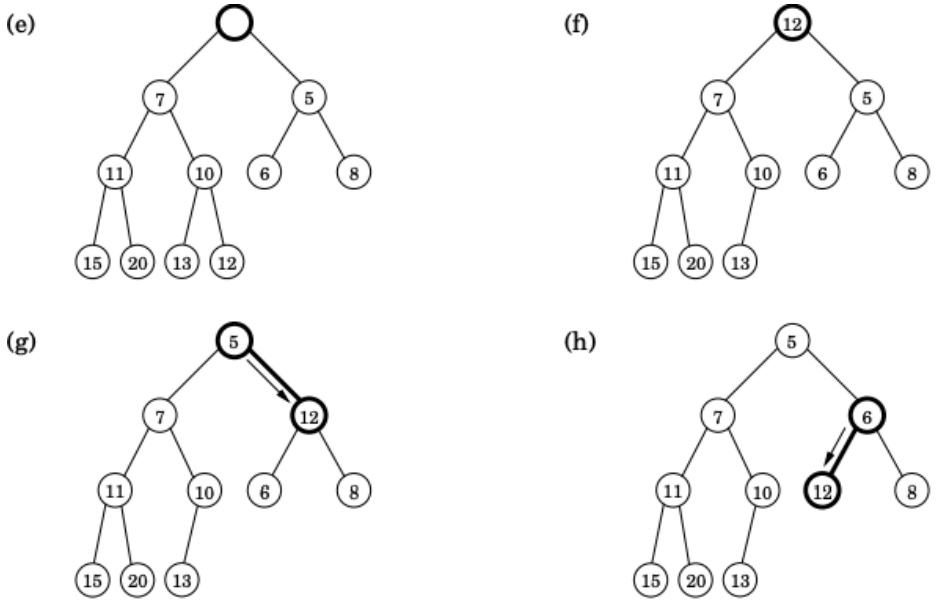
- deletemin 연산

: root value return (root 에는 항상 최소값이 있으니까)

: root value 를 삭제하고 나서, 트리의 가장 아래쪽 가장 오른쪽에 있는 값을 root 자리에 넣는다.

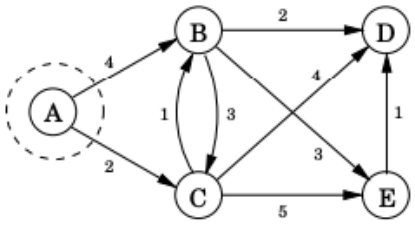
: 그리고 sift down 수행

* sift down: 자식보다 내가 더 크면 자식 중 가장 작은 것과 swap → swap 되지 않을 때까지 반복



→ $O(\log n)$ time

* 질문



→어떻게 binary heap 으로..?

4.5.3 d-ary heap

: binary heap 은 자식을 2 개 가질 수 있지만 d-ary heap 은 d 개 가질 수 있다

: 위의 차이를 빼고는 binary heap 과 동일

: binary heap 에 비해 height 가 감소 = $\Theta(\log_d n) = \Theta((\log n)/(\log d))$

- insert 연산

  : 더 빨라짐 = $\Theta(\log d)$

- deletemin 연산

  : 봐야할 자식이 많아졌으므로 더 느려진다 = $O(d \log_d n)$

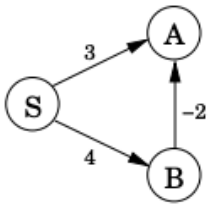
4.6 Shortest paths in the presence of negative edges

4.6.1 Negative edges

Edge length 가 음수인 경우 Dijkstra 알고리즘은 부분적으로 동작한다

: 아래의 그림에서 S to B 의 shortest path 가 더 멀리 돌아가는 경우

Figure 4.12 Dijkstra's algorithm will not work if there are negative edges.



- 이런 새로운 상황을 어떻게 해결할 수 있을까? → Dijkstra's 알고리즘을 활용하여

dist value 는 항상 해당 노드까지의 거리와 정확히 동일하거나 더 크거나 둘 중 하나다

dist value 는 ∞에서 시작되고, 그 값이 바뀌는 경우는 아래와 같이 update 될때 뿐

procedure update $((u,v) \in E)$

$\texttt{dist}(v) = \min\{\texttt{dist}(v), \texttt{dist}(u) + l(u,v)\}$

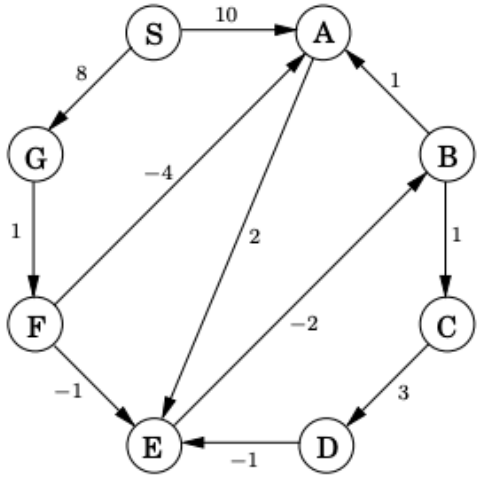
- 위의 update 연산은
  : “v 까지의 거리는 u 까지의 거리+/(u, v)보다 더 클 가능성이 없다”는 사실을 이용한 것
  : 2 가지 특성이 있음
  1) v 까지의 shortest path 에서 u 가 second-last node 일 때 + dist(u)가 정확한 경우 → update 연산은 정확한 distance 를 줄 수 있음
  2) dist(v)의 값을 매우 작게 만들지 않기 때문에 safe 하다고 할 수 있음(많은 수의 update 문에 영향을 받지 않음)

  → Dijkstra’s 알고리즘은 사실 update 를 단순히 나열한 것으로 볼 수 있음
  → 하지만, 이런 단순한 나열은 negative edges 에선 동작하지 못한다

* Bellman-Ford algorithm

: negative edge 에서도 동작함
: 모든 edge 들에 대해 |v|-1 번의 update

Figure 4.14 The Bellman-Ford algorithm illustrated on a sample graph.

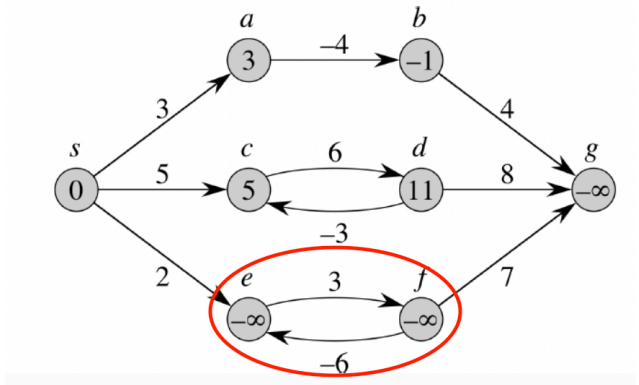


→

	Iteration							
Node	0	1	2	3	4	5	6	7
S	0	0	0	0	0	0	0	0
A	∞	10	10	5	5	5	5	5
B	∞	∞	∞	10	6	5	5	5
C	∞	∞	∞	∞	11	7	6	6
D	∞	∞	∞	∞	∞	14	10	9
E	∞	∞	12	8	7	7	7	7
F	∞	∞	9	9	9	9	9	9
G	∞	8	8	8	8	8	8	8

4.6.2 Negative cycles

: graph 에서 negative cycle 이 존재하는 경우 사이클을 돌면 돌수록 거리가 작아져 벨만-포드 알고리즘으로 최단경로를 구하는 것에 의미가 없어짐
* 여기서 말하는 negative cycle



→ e,f 가 negative cycle(c,d 는 아님!)

: 모든 엣지에 대해서 edge relaxation 을 시작노드를 제외한 전체 노드에 대해 진행한 후, 마지막으로 그래프 모든 엣지에 대해 edge relaxation 을 한번 더 수행한다. 이때 한번이라도 업데이트가 일어나면 negative edge 가 있다는 것!

```
BELLMAN-FORD(G, w, s)
INIT-SINGLE-SOURCE(G, s)
for i = 1 to |G.V| − 1
    for each edge (u, v) ∈ G.E
        RELAX(u, v, w)
for each edge (u, v) ∈ G.E
    if v.d > u.d + w(u, v)
        return FALSE
return TRUE
```

4.7 Shortest paths in dags

Negative cycles 의 가능성을 제외하면, 그래프는 2 가지의 subclass 를 가짐

1) negative edges 가 없는 그래프

2) cycles 가 없는 그래프

1)은 이미 배웠고

2) cycles 가 없는 그래프 = single-source shortest-path in DAG

: linear time 내에 풀 수 있다

→ DAG 를 DFS 를 통해 linearize 할 수 있음

→ 정렬된 순서로 노드 방문

→ 각 노드들마다 노드의 edges 를 update

Figure 4.15 A single-source shortest-path algorithm for directed acyclic graphs.

```
procedure dag-shortest-paths(G, l, s)
Input:    Dag G = (V, E);
          edge lengths {l_e : e ∈ E}; vertex s ∈ V
Output:   For all vertices u reachable from s, dist(u) is set
          to the distance from s to u.

for all u ∈ V:
   dist(u) = ∞
   prev(u) = nil

dist(s) = 0
Linearize G
for each u ∈ V, in linearized order:
   for all edges (u, v) ∈ E:
      update(u, v)
```

→ 이를 이용하여 가장 longest path 를 찾을 수 있음: 모든 길이에 -부호로 뒤집어주면 됨