



Shorted Paths on Weighted Digraphs

최단경로 탐색 방법과 활용도 이해

01. 퀴즈 풀이 & 예습 내용 복습 (이번 주 #1~3차 답안 공개)
02. 최단경로 문제의 기본 세팅 + 최단경로 탐색 방법의 공통점
03. Bellman-Ford 알고리즘
04. Dijkstra 알고리즘
05. Acyclic Shortest Path
06. Seam Carving
07. 실습: Seam Carving 구현

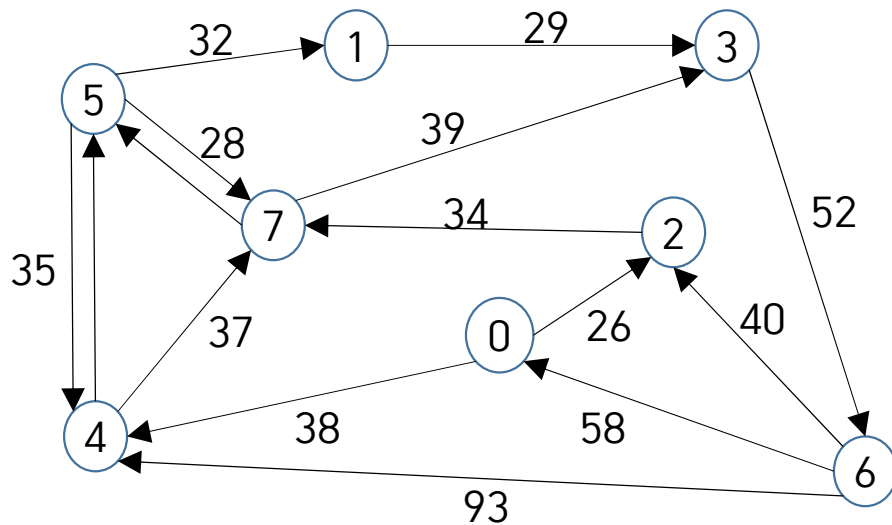


Edge-Weighted Digraph에서의 최단경로(Shortest Path)

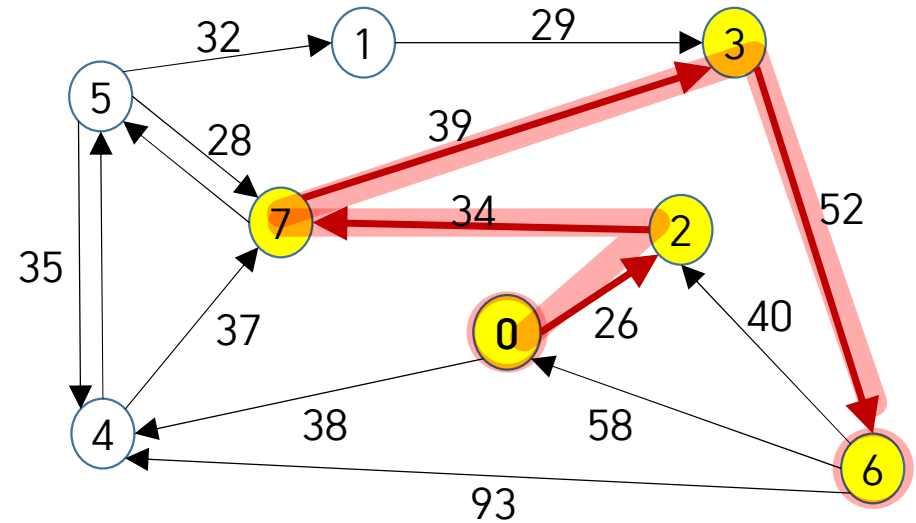
2

▪ <입력>

- 간선에 **weight** 있는
- **directed** graph G



- <출력: 출발지 s에서 도착지 t까지 **최단경로**>
- s에 t까지 연결하는 **경로** 중 (연결된 간선의 집합)
- **간선의 weight 합이 최소**인 경로
가장 짧게

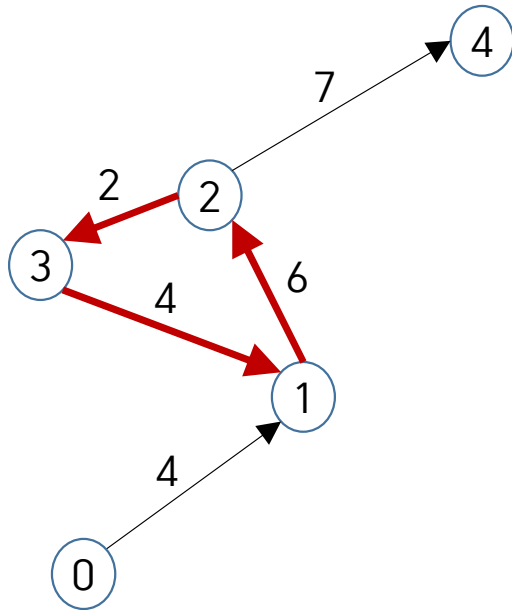


정점 0에서 6까지의 최단경로

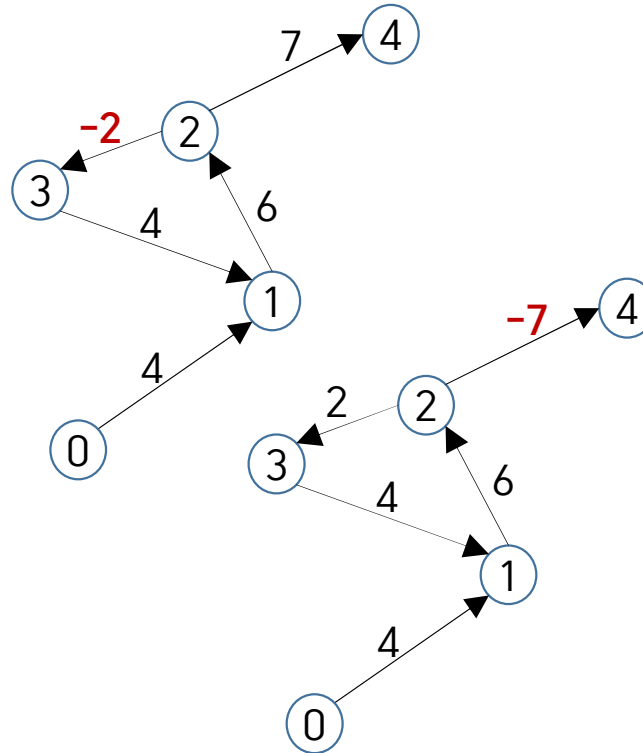


Weight 합 < 0인 사이클 있는 그래프는 최단경로 존재하지 않음

3



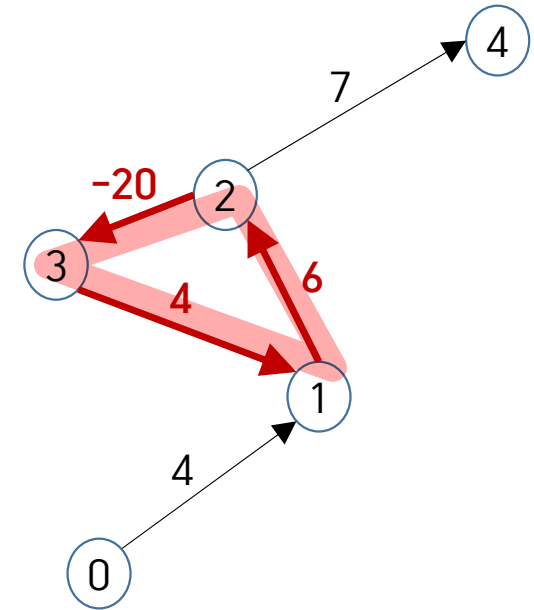
- 사이클 있어도 최단경로 존재



- 간선 weight < 0 이어도
- 최단경로 존재

양수/음수 구분하는 예?
양수 weight은 penalty
음수 weight은 reward(보상)

해미들 다 가능한데,
들이 뛸면 불가능



- 간선의 weight 모두 더한 총 합이
- 음수인 사이클 (negative cycle)
- 있으면 최단경로 존재할 수 없음

(\therefore 임계에서 제리)



최단경로 활용 예

4

<인터넷 지도(카카오맵, 네이버맵, 구글맵, ...)>



<자동차 네비게이션 시스템>



... 이 외에도 매우 많음 ...

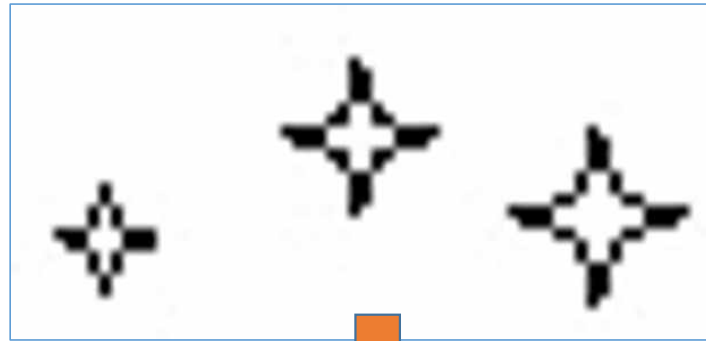


최단경로 활용 예

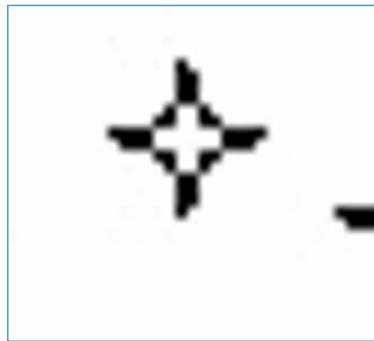
5

- Seam Carving: 이미지의 크기 조절 시 중요한 부분 자동 인식해 최대한 보존하며 조절하는 방식 (web browser, 휴대폰 등에서 활용)

<원본>



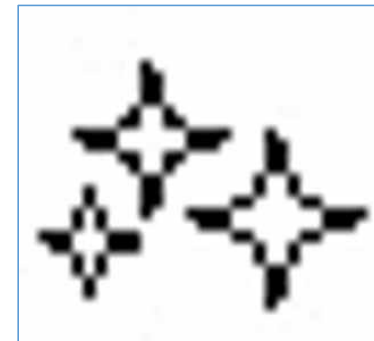
<Resize 후>



① Resize 시
잘라내는 방식
(crop)

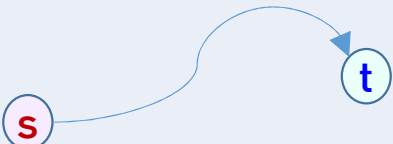
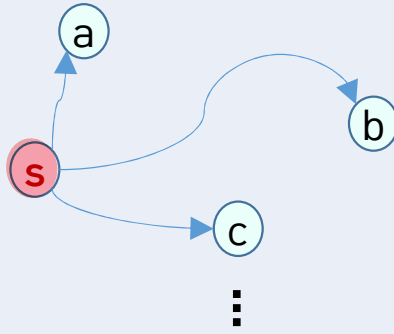
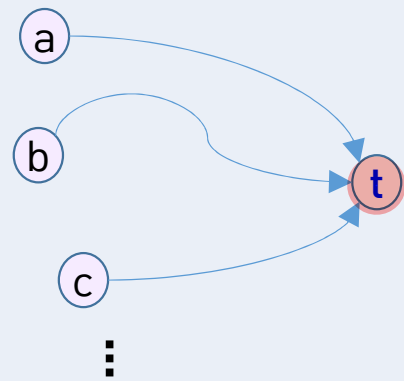
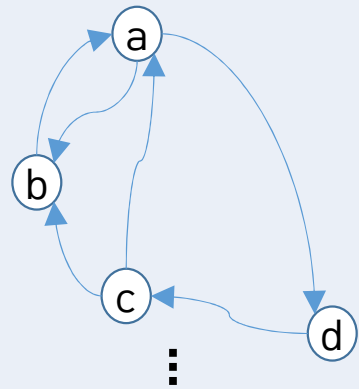


② Resize 시
찌그러뜨리는 방식



③ Seam Carving
(중요한 부분 최대한 보존)

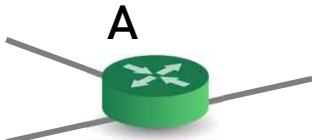
어떤 정점 간 최단경로를 구할 것인가?

	① Source-Sink	② Single Source	③ Single Sink	④ All Pairs
출발지	정점 s 하나	정점 s 하나	모든 정점	모든 정점
목적지	정점 t 하나	다른 모든 정점	정점 t 하나	모든 정점
설명	s~t까지 경로 찾기	s에서 다른 모든 정점까지 경로 찾기	모든 정점에서 t까지 경로 찾기	모든 정점 간 경로 찾기
예제				

6.11.19

6-임새 77

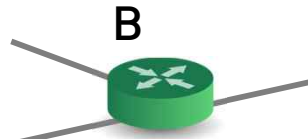
어떤 정점 간 최단경로를 구할 것인가? ② Single Source



라우터 A의 라우팅 테이블

목적지 IP	전송 링크 #
1.*.*.*	1
2.*.*.*	2
3.*.*.*	3
4.*.*.*	4
5.*.*.*	5
...	..

라우터 A를 출발지로 해서
모든 목적지까지 경로 계산



라우터 B의 라우팅 테이블

목적지 IP	전송 링크 #
1.*.*.*	3
2.*.*.*	2
3.*.*.*	1
4.*.*.*	1
5.*.*.*	3
...	

라우터 B를 출발지로 해서
모든 목적지까지 경로 계산



경북대를 출발지로
경로 계산

자동차 1의
네비게이션 시스템

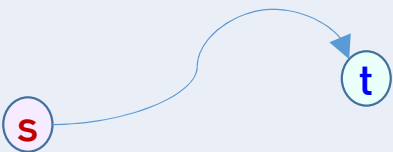
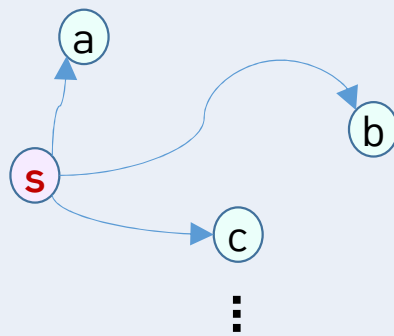
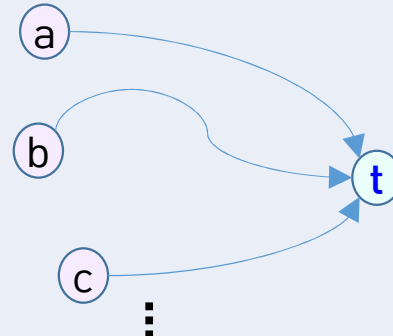
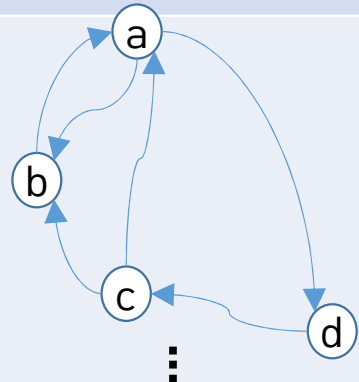


동대구역을 출발지로
경로 계산

자동차 2의
네비게이션 시스템

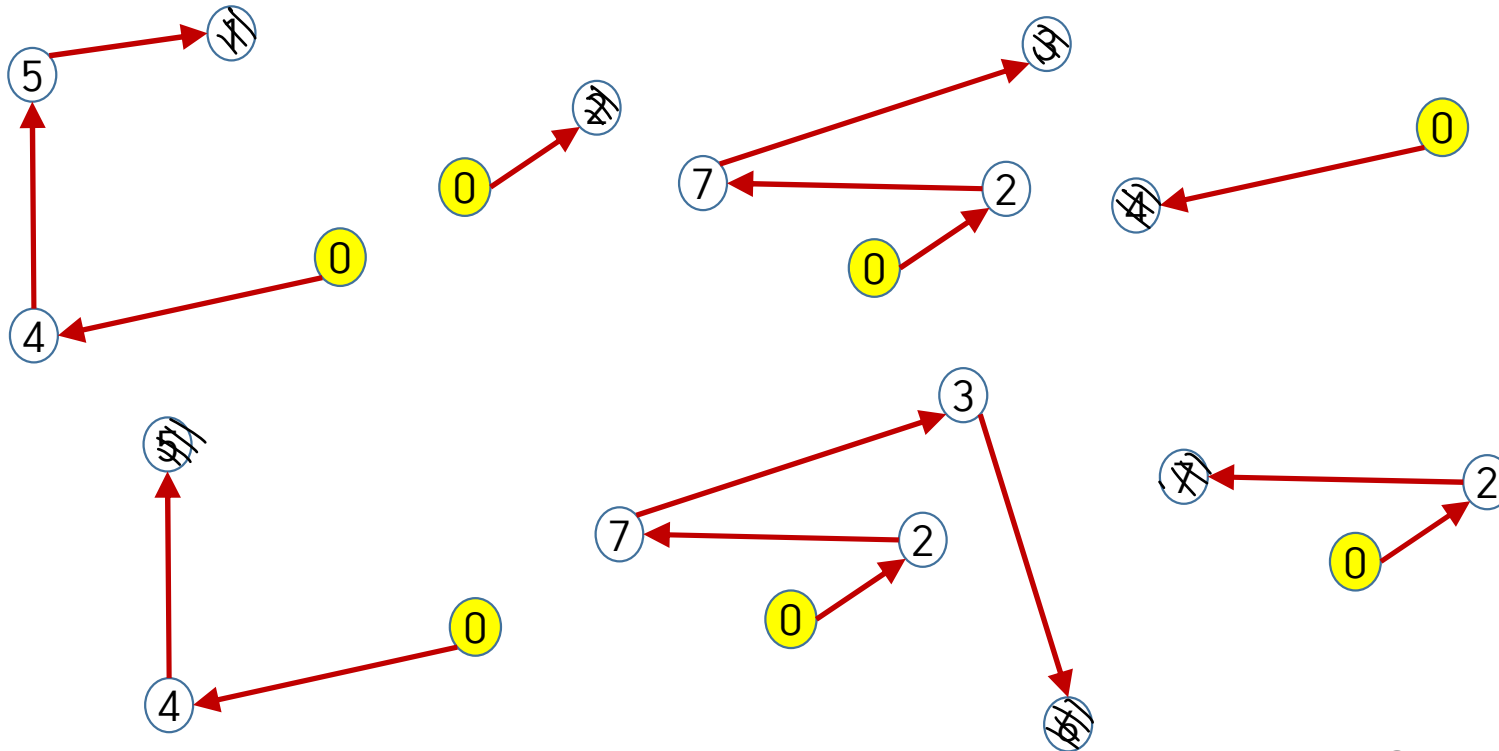


[Q] ① Source-Sink와 ④ All Pairs 문제는 ② Single Source 방법 사용해 풀 수 있다.
그렇다면 ③ Single Sink는 Single Source로 풀 수 있는가?

	① Source-Sink	② Single Source	③ Single Sink	④ All Pairs
출발지	정점 s 하나	정점 s 하나	모든 정점	모든 정점
목적지	정점 t 하나	다른 모든 정점	정점 t 하나	모든 정점
설명	s~t까지 경로 찾기	s에서 다른 모든 정점까지 경로 찾기	모든 정점에서 t까지 경로 찾기	모든 정점 간 경로 찾기
예제				
Single Source와 관계	Single Source 해로부터 Source-Sink해 얻을 수 있음		?? <i>reverse graph</i>	Single Source를 여러 출발지에 적용하여 풀이 가능



- Single Source 문제 풀이
- Single Source s 는 정점 0이라 가정
- 즉 0에서 다른 모든 정점까지 최단경로 구하는 상황

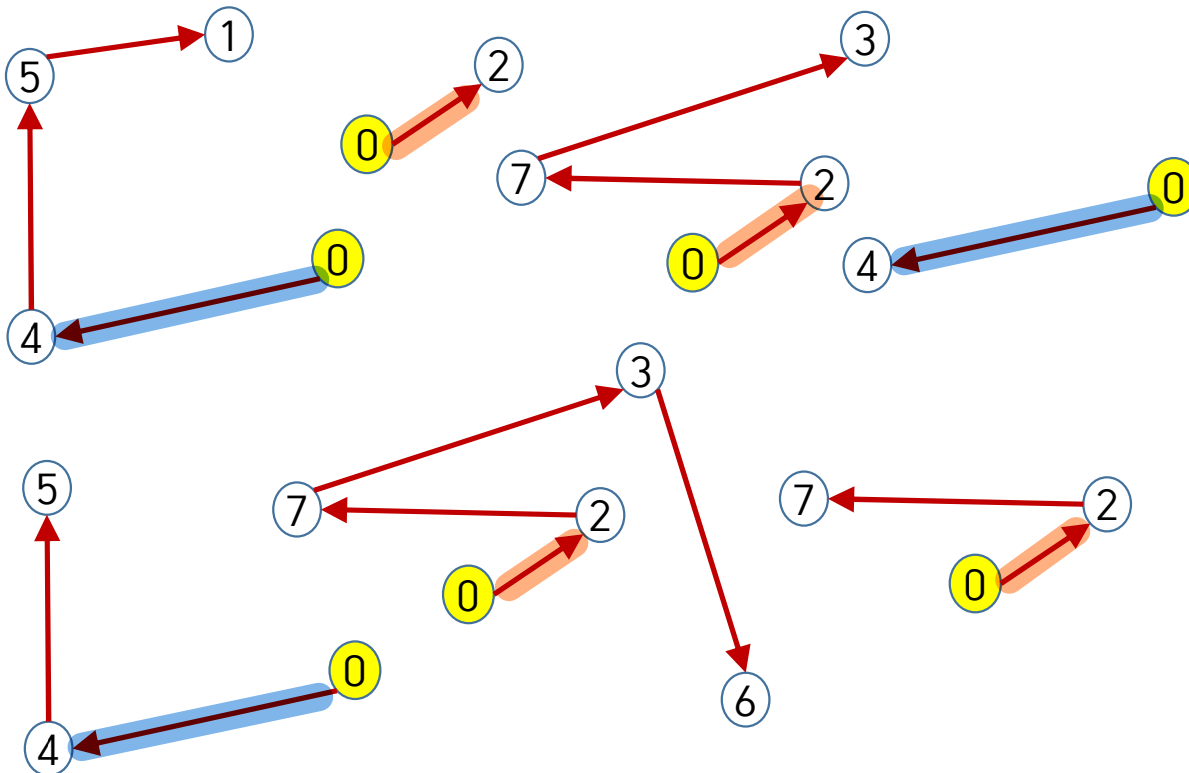




Single Source 문제의 해는 어떤 자료구조에 저장할 것인가?

목표: compact 하면서 원하는 해에 접근도 쉬운 구조

- (방법 1) 각 목적지마다 최단 경로 저장 (거쳐가는 정점 or 간선의 리스트 형태로)



목적지 t	0으로부터 최단경로
1	0→4→5→1
2	0→2
3	0→2→7→3
4	0→4
5	0→4→5
6	0→2→7→3→6
7	0→2→7

[Q] 최단 경로들을 비교 관찰해 보자.
크기 더 줄여 저장할 수는 없을까?

☐처럼 마지막 간선만 저장! (점차 가능 함)

1	5→1
2	0→2
3	7→3
4	0→4
5	4→5
6	3→6
7	2→7

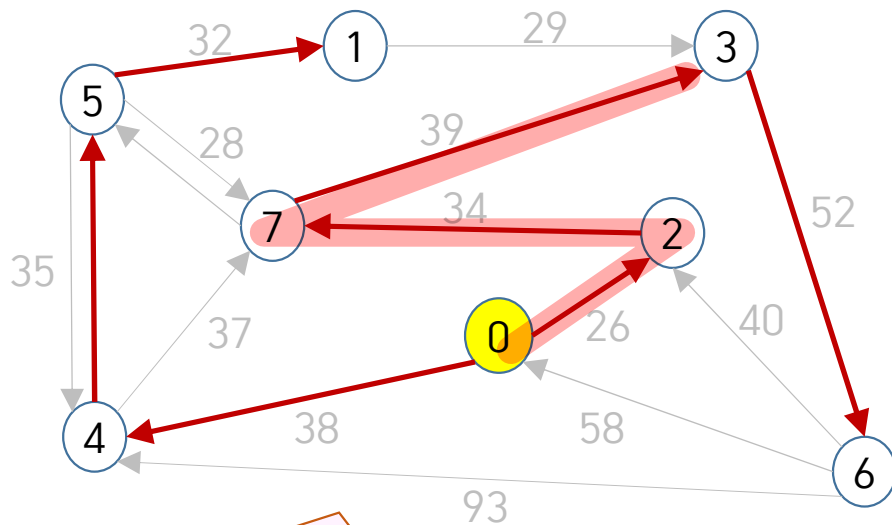
Copyright © by Sihyung Lee - All rights reserved.



Single Source 문제의 해는 어떤 자료구조에 저장할 것인가?

목표: compact 하면서 원하는 해에 접근도 쉬운 구조

- (방법 2) **SPT (Shortest Path Tree)** 형태로 저장 : *최단 경로 트리*
- Why? Single source 문제의 해는 하나의 트리로 표현됨



정점 0에서 다른 모든 정점까지의 SPT

- 최단+최단+최단...*
- 정점 $s \sim t$ 까지 **최단 경로의 일부 역시 최단 경로**
 - ($s \sim t$ 까지 최단 경로가 k 를 거친다면,
 - $s \sim k$ 경로 역시 k 까지의 최단 경로)

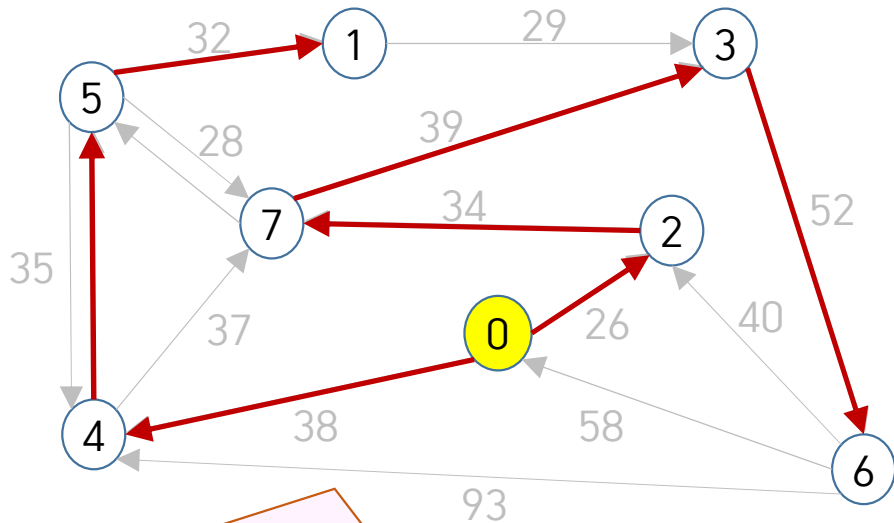
[Q] 위 명제가 항상 참임을 증명 하시오.



SPT (Shortest Path Tree) 저장하는 자료 구조: edgeTo[], distTo[]

edgeTo[v]: s~v 최단경로에
사용된 **마지막 간선**

distTo[v]:
s~v 최단경로의 길이



정점 0에서 다른 모든 정점까지의 SPT

v	edgeTo[v]	distTo[v]
0	None	0
1	5→1	105
2	0→2	26
3	7→3	99
4	0→4	38
5	4→5	73
6	3→6	151
7	2→7	60

[Q] 왼쪽 그래프와 비교해 edgeTo[v]와 distTo[v]가 올바른지 검증해 보시오.

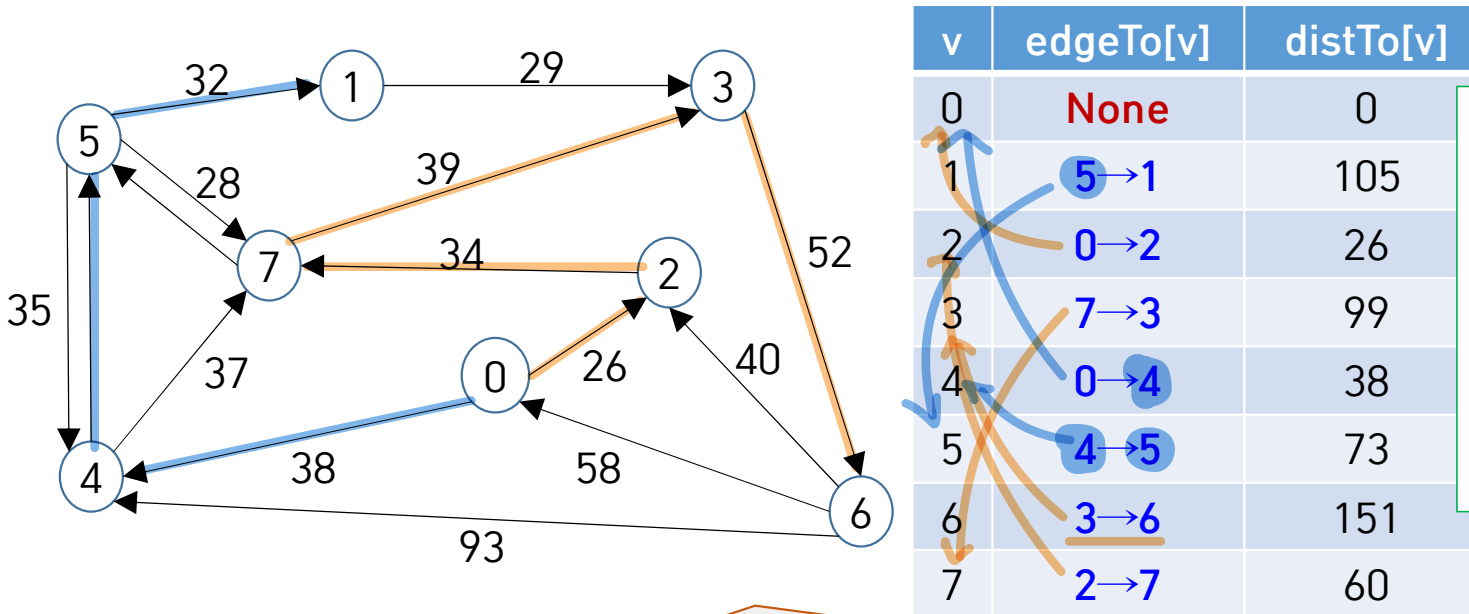
- 각 경로 저장하는 방법에 비해 작은 공간 차지하면서
- 임의의 목적지 t에 대한 최단 경로와 재구성도 간단



SPT를 edgeTo[]와 distTo[]에 저장했다면 목적지 t에 대한 경로는 어떻게 재구성?

13

- edge[t]에서 시작, s 방향으로 거꾸로 돌아가며 s~t 경로 재구성



```
def pathTo(self, v): (s → v)의 경로를 리턴
    path = []
    e = self.edgeTo[v]
    while e != None:
        path.append(e)
        e = self.edgeTo[e.v]
    path.reverse()
    return path
```

[Q] pathTo(1)을 호출하였다. 어떤 경로가 반환되는가?

0 → 4 → 5 → 1

[Q] pathTo(6)을 호출하였다. 어떤 경로가 반환되는가?

0 → 2 → 7 → 3 → 6



간선

간선의 길이

edgeTo[]와 distTo[]는 어떻게 만들어가면 될까?

14

① 초기화

- (모든 정점에 대해) $\text{edgeTo}[] = \text{None}$
- (출발지 s) $\text{distTo}[s] = 0$
- (그 외) $\text{distTo}[t] = \infty$

초기화 해볼 시작

v	edgeTo[v]	distTo[v]
0	None	0
1	None	∞
2	None	∞
3	None	∞
4	None	∞
5	None	∞
6	None	∞
7	None	∞

출발지만 아니냐
여기부터 시작해야함.
(∵ 알고리즘은 끝까지 실행해야함)

② 탐색

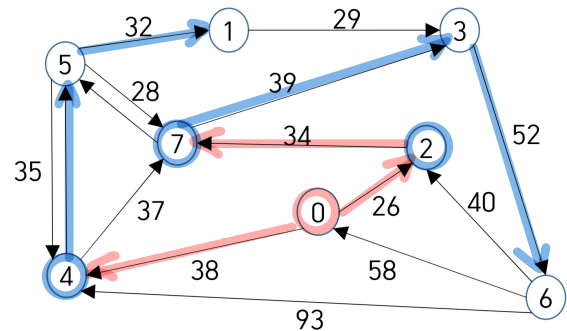
- 기존에 알던 s-t 경로보다 **더 짧은 경로 발견하면**
- $\text{edgeTo}[t]$, $\text{distTo}[t]$ **업데이트**
- $\text{distTo}[t]$: **현재까지 발견한 t까지 최단거리**
- $\text{edgeTo}[t]$: **현재까지 발견한 t까지 최단경로에서 마지막 간선**

v	edgeTo[v]	distTo[v]
0	None	0
1	None	∞
2	0→2	26
3	None	∞
4	0→4	38
5	None	∞
6	None	∞
7	0→2→7	60

③ 완료

- 모든 가능한 경로 탐색 끝났다면
- $\text{distTo}[t]$: t까지 최단거리
- $\text{edgeTo}[t]$: t까지 최단경로에서 마지막 간선

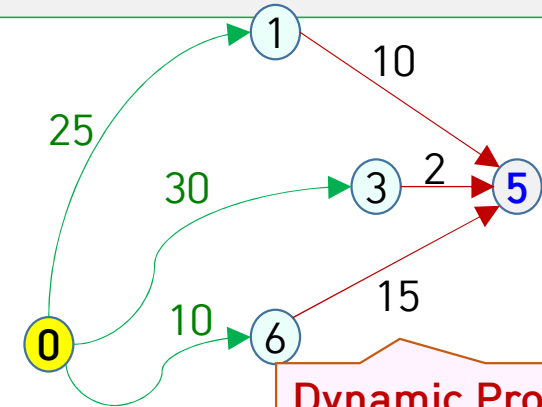
v	edgeTo[v]	distTo[v]
0	None	0
1	5→1	105
2	0→2	26
3	7→3	99
4	0→4	38
5	4→5	73
6	3→6	151
7	2→7	60





서로 다른 경로 탐색은 어떻게 하나? 기 발견한 경로 + 간선

- t 까지 경로 탐색: s 부터 t 까지 처음부터 경로 구성하는 대신
- *Dynamic Programming*
기존에 알고 있던 경로 $s \sim v$ 에 간선 $e=v \rightarrow t$ 더해 새 경로 $P=s \sim t$ 구성
- 기존에 알고 있던 경로 $P':s \sim t$ 보다 P 가 더 짧다면 P' 를 P 로 대체



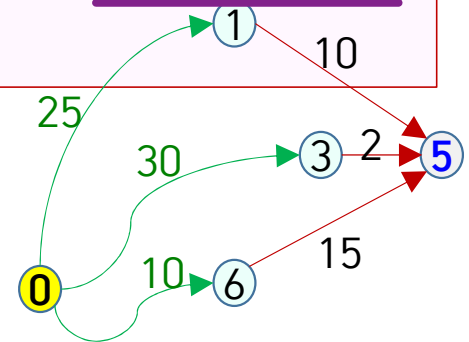
Dynamic Programming

v	edgeTo[v]	distTo[v]		v	edgeTo[v]	distTo[v]		v	edgeTo[v]	distTo[v]		v	edgeTo[v]	distTo[v]
0	None	0		0	None	0		0	None	0		0	None	0
1	...	25		1	...	25		1	...	25		1	...	25
2		2		2		2
3	...	30	→	3	...	30	→	3	...	30	→	3	...	30
4		4		4		4
5	None	∞		5	1→5	35		5	3→5	32		5	6→5	25
6	...	10		6	...	10		6	...	10		6	...	10
7		7		7		7



relax(e): 기존에 알던 $s \rightarrow v$ 경로에 간선 $e=v \rightarrow t$ 더했을 때 t 까지 더 짧은 경로 나온다면 이 경로를 $\text{edgeTo}[t]$, $\text{distTo}[t]$ 에 저장

```
def relax(self, e):
    if self.distTo[e.t] > self.distTo[e.v] + e.weight:
        self.distTo[e.t] = self.distTo[e.v] + e.weight
        self.edgeTo[e.t] = e
```



relax(1→5): 간선 1→5 사용해
5까지 더 짧은 경로 나오면
이 경로 기억

relax(3→5): 간선 3→5 사용해
5까지 더 짧은 경로 나오면
이 경로 기억

relax(6→5): 간선 6→5 사용해
5까지 더 짧은 경로 나오면
이 경로 기억

v	edgeTo[v]	distTo[v]
0	None	0
1	...	25
2
3	...	30
4
5	None	∞
6	...	10
7

v	edgeTo[v]	distTo[v]
0	None	0
1	...	25
2
3	...	30
4
5	1→5	35
6	...	10
7

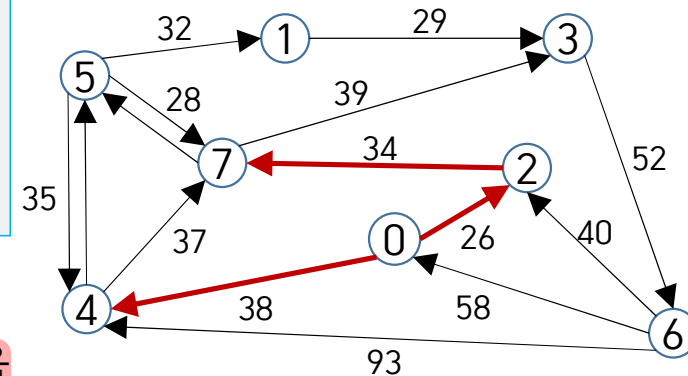
v	edgeTo[v]	distTo[v]
0	None	0
1	...	25
2
3	...	30
4
5	3→5	32
6	...	10
7

v	edgeTo[v]	distTo[v]
0	None	0
1	...	25
2
3	...	30
4
5	6→5	25
6	...	10
7

아직 알지 못하는 경로에 간선 더해 새로운 경로 만들지는 않음
현재까지 찾은 경로에 간선 더해 새로운 경로 만들어냄

```
def relax(self, e):
    if self.distTo[e.t] > self.distTo[e.v] + e.weight:
        self.distTo[e.t] = self.distTo[e.v] + e.weight
        self.edgeTo[e.t] = e
```

- $e=v \rightarrow t$ relax할 때, v 까지 경로 아직 모른다면($\text{distTo}[v]=\infty$)
- 지금까지 발견된 t 까지 경로보다 짧아질 수 없어 $\text{edgeTo}[], \text{distTo}[]$ 변화 없음



v	edgeTo[v]	distTo[v]
0	None	0
1	None	∞
2	0→2	26
3	None	∞
4	0→4	38
5	None	∞
6	None	∞
7	2→7	60

relax(1→3)

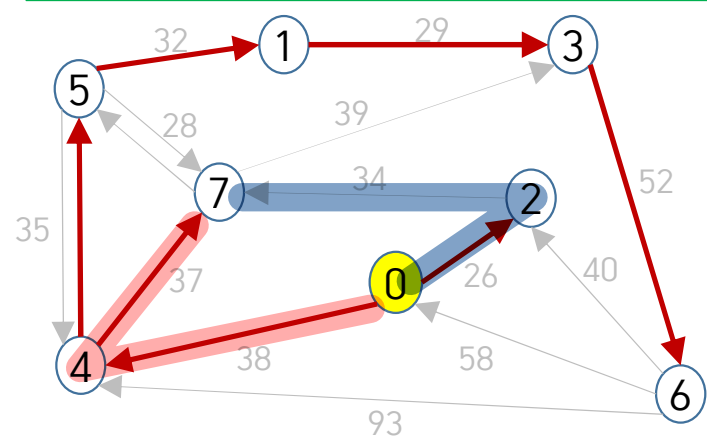
v	edgeTo[v]	distTo[v]
0	None	0
1	None	∞
2	0→2	26
3	None	∞
4	0→4	38
5	None	∞
6	None	∞
7	2→7	60

relax(5→1)

이 모든 경로에
다해봐서
의미가 없다.

v	edgeTo[v]	distTo[v]
0	None	0
1	None	∞
2	0→2	26
3	None	∞
4	0→4	38
5	None	∞
6	None	∞
7	2→7	60

[Q] 아래 왼쪽 상태에서 2개 간선 $2 \rightarrow 7$, $7 \rightarrow 3$ 을 차례로 relax했을 때 edgeTo[]와 distTo[]에 저장된 값이 어떻게 변하는지 써 보시오.



v	edgeTo[v]	distTo[v]
0	None	0
1	5→1	105
2	0→2	26
3	1→3	134
4	0→4	38
5	4→5	73
6	3→6	186
7	4→7	75

$0 \rightarrow 4 \rightarrow 7$

$0 \rightarrow 4 \rightarrow 5 \rightarrow 1 \rightarrow 3$



relax(2→7): 간선 $2 \rightarrow 7$ 사용해
7까지 더 짧은 경로 나오면
이 경로 기억

$0 \rightarrow 2 \rightarrow 7$

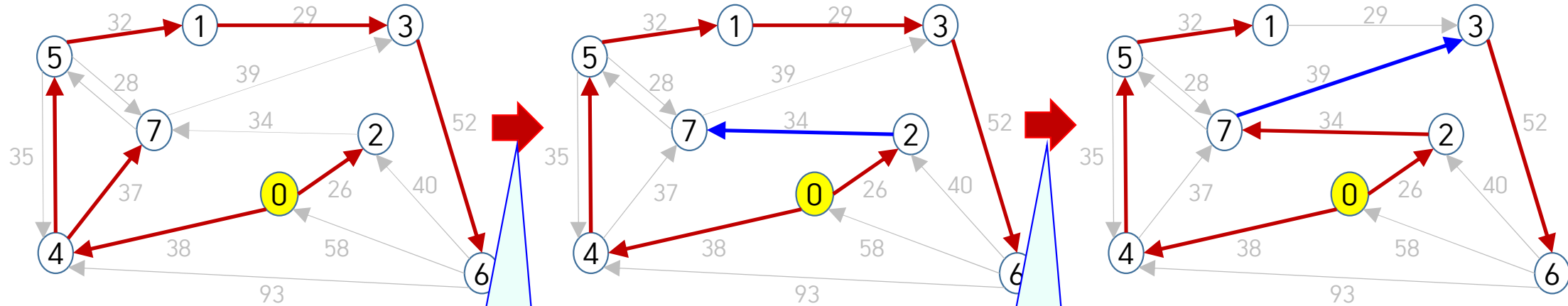
v	edgeTo[v]	distTo[v]
0	None	0
1	5→1	105
2	0→2	26
3	1→3	134
4	0→4	38
5	4→5	73
6	3→6	186
7	2→7	60

$0 \rightarrow 2 \rightarrow 7 \rightarrow 3$



relax(7→3): 간선 $7 \rightarrow 3$ 사용해
3까지 더 짧은 경로 나오면
이 경로 기억

v	edgeTo[v]	distTo[v]
0	None	0
1	5→1	105
2	0→2	26
3	7→3	99
4	0→4	38
5	4→5	73
6	3→6	186
7	2→7	60



relax(2→7)

v	edgeTo[v]	distTo[v]
0	None	0
1	5→1	105
2	0→2	26
3	1→3	134
4	0→4	38
5	4→5	73
6	3→6	186
7	4→7	75

relax(7→3)

v	edgeTo[v]	distTo[v]
0	None	0
1	5→1	105
2	0→2	26
3	1→3	134
4	0→4	38
5	4→5	73
6	3→6	186
7	2→7	60

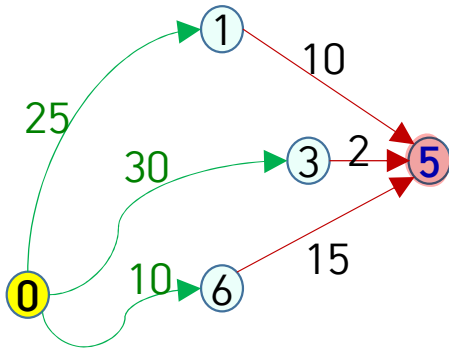
v	edgeTo[v]	distTo[v]
0	None	0
1	5→1	105
2	0→2	26
3	7→3	99
4	0→4	38
5	4→5	73
6	3→6	186
7	2→7	60

[Q] 앞의 두 relax 결과가 제대로 반영되지 않아 여전히 relax 필요한 행이 있는가?

다시 relax 해야 하는 것 같아

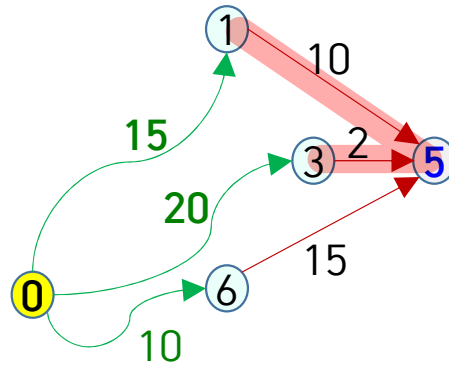


경로 앞쪽도 같은 방식으로 변경되면 그 뒤 경로 우선순위도 바뀌어야 함
따라서 **간선 고려하는 (relax) 순서 잘 정해야 relax 횟수 최적화 가능**



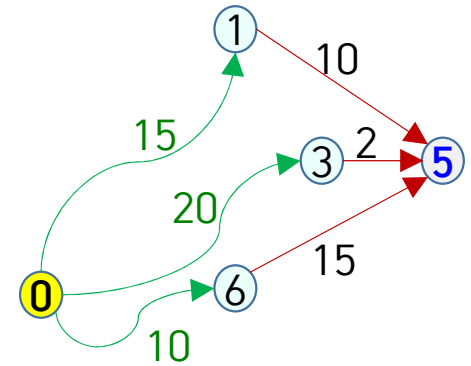
v	edgeTo[v]	distTo[v]
0	None	0
1	...	25
2
3	...	30
4
5	6→5	25
6		10
7		...

relax 1→5, 3→5, 6→5 한 결과



v	edgeTo[v]	distTo[v]
0	None	0
1	...	15
2
3	...	20
4
5	6→5	25
6	...	10
7

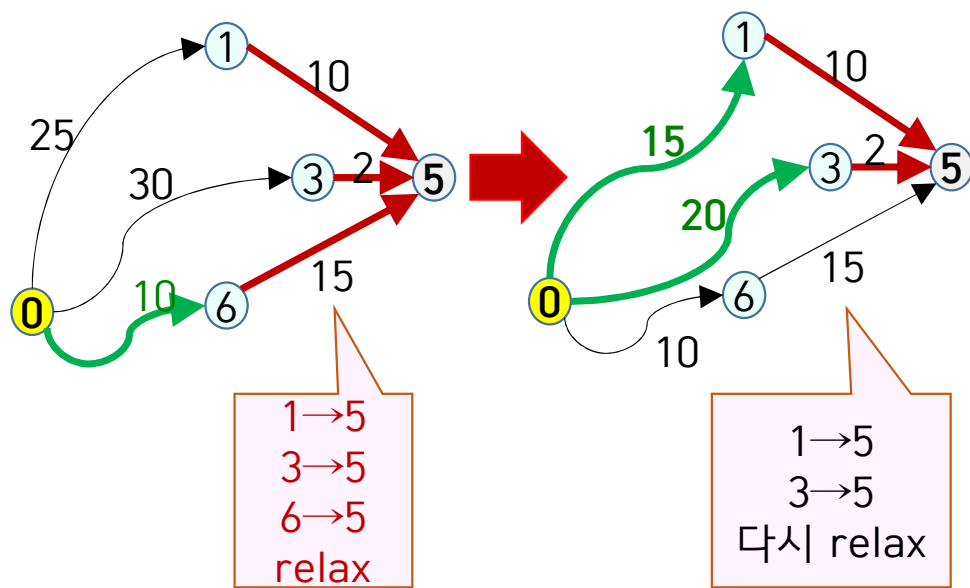
다른 relax에 의해 정점 1, 3
으로의 최단 경로 변경됨



v	edgeTo[v]	distTo[v]
0	None	0
1	...	15
2
3	...	20
4
5	3→5	22
6	...	10
7

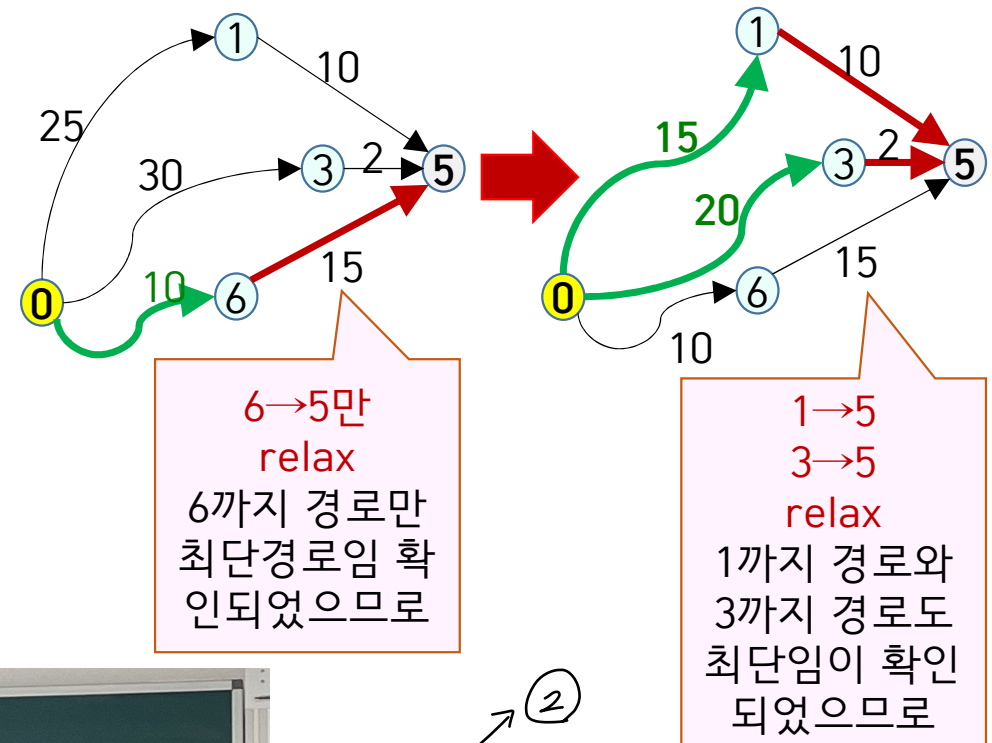
정점 5로 가는 간선 1→5, 3→5, 6→5
다시 relax 필요 (했던 일 다시 반복)

- $\text{relax}(x \rightarrow v)$ 에 의해
- v 까지 경로가 더 짧은 경로로 변경되었다면
- v 에서 나가는 각 간선 $e=v \rightarrow w$ 를 다시 relax해서
- w 까지의 경로도 수정해주어야 함

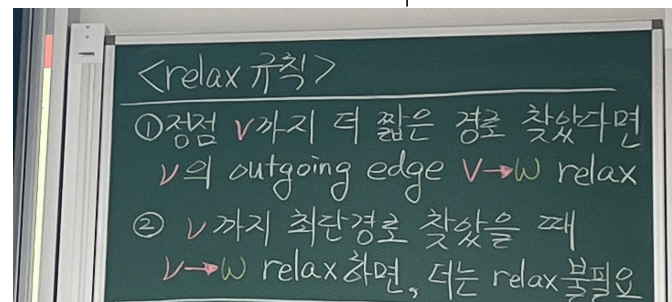


1→5
3→5
다시 relax

- v 까지의 최단 경로 계산 끝난 후에
- v 에서 나가는 각 간선 $e=v \rightarrow w$ 를 relax 한다면
- e 를 단 한 번만 relax하면 되어 효율적일 것임

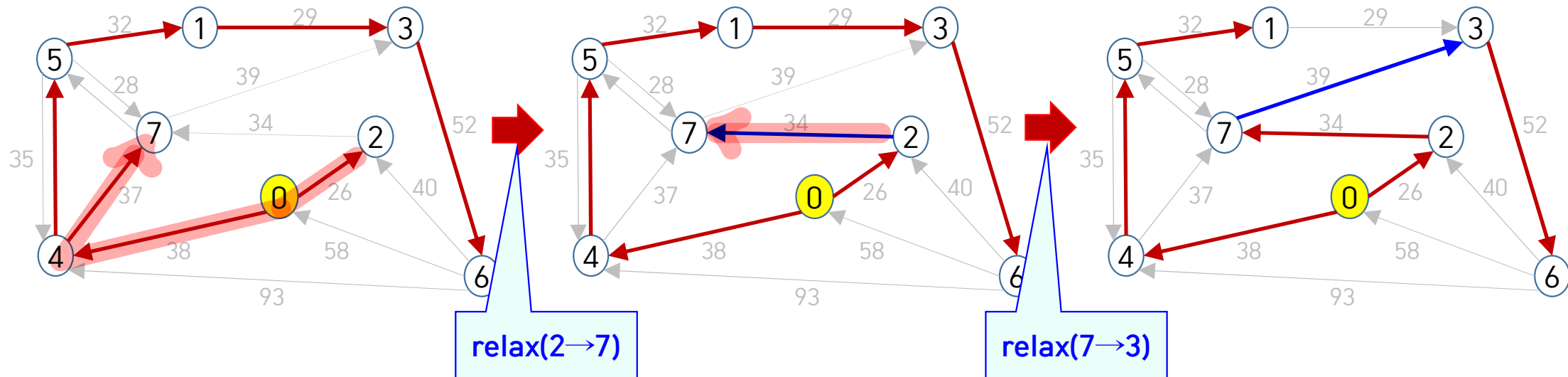


1→5
3→5
relax
1까지 경로와
3까지 경로도
최단임이 확인
되었으므로





정리: (1) 최단경로는 **트리(SPT)** 형태로 저장,
(2) SPT 얻기 위해 적절한 순서로 **간선 relax**하는 일 반복



Bellman
Dijkstra
Acyclic → **Dijkstra**
Acyclic



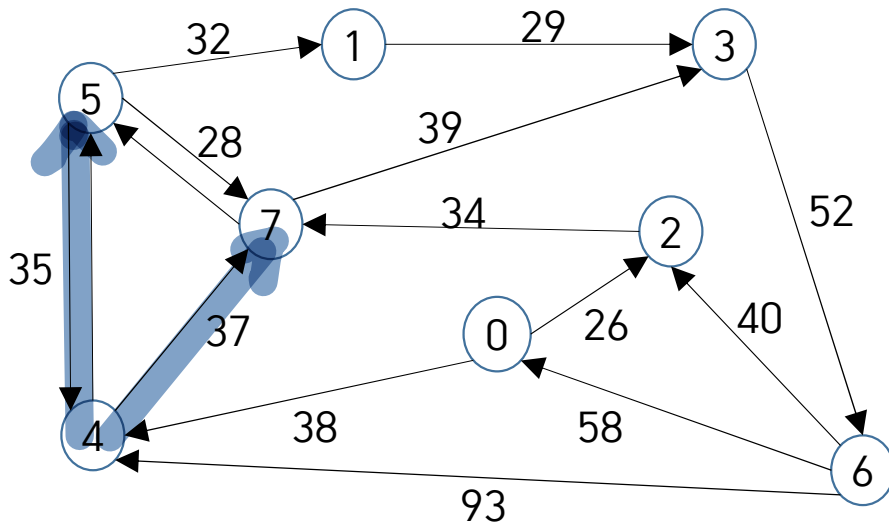
Shorted Paths on Weighted Digraphs

최단경로 탐색 방법과 활용도 이해

01. 퀴즈 풀이 & 예습 내용 복습 (이번 주 #1~3차 답안 공개)
02. 최단경로 문제의 기본 세팅 + 최단경로 탐색 방법의 공통점
03. Bellman-Ford 알고리즘
04. Dijkstra 알고리즘
05. Acyclic Shortest Path
06. Seam Carving
07. 실습: Seam Carving 구현



relax(v): 정점 v 에서 ^{나가는 방향} outgoing하는 모든 간선 $e=v \rightarrow w$ 를 relax
 (not 간선, yes 정점)



- 왼쪽 그래프에서
- relax(0): 간선 $0 \rightarrow 2$, $0 \rightarrow 4$ relax
- relax(1): 간선 $1 \rightarrow 3$ relax
- relax(2): 간선 $2 \rightarrow 7$ relax
- relax(3): 간선 $3 \rightarrow 6$ relax
- relax(4): 간선 $4 \rightarrow 5$, $4 \rightarrow 7$ relax
- relax(5): 간선 $5 \rightarrow 1$, $5 \rightarrow 4$, $5 \rightarrow 7$ relax
- relax(6): 간선 $6 \rightarrow 0$, $6 \rightarrow 2$, $6 \rightarrow 4$ relax
- relax(7): 간선 $7 \rightarrow 5$, $7 \rightarrow 3$ relax



Bellman-Ford 알고리즘 (매우 간단)

- 초기화: (모든 정점에 대해) $edgeTo[] = None$, (출발지 s) $distTo[s]=0$, (그 외) $distTo[t]=\infty$
- 다음을 **$V-1$ 회 반복** (V : 정점 개수)
 - **모든 정점 relax (=모든 간선 relax)**

```
# g: EdgeWeightedDigraph 객체
for _ in range(g.V)-1:
    for v in range(g.V):
        for e in g.adj[v]: relax(e)
```

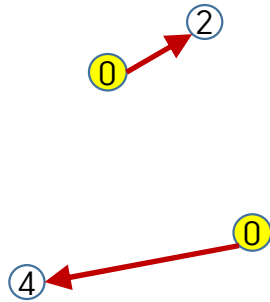
최대 **$\sim V \times E$** 시간 소요
(V : 정점 개수
 E : 간선 개수)



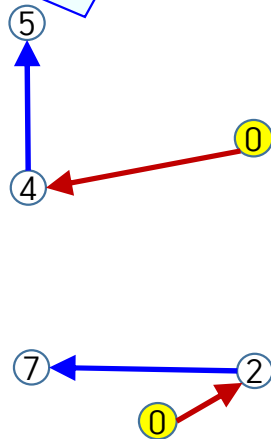
Bellman-Ford 알고리즘

- 다음을 **V-1회 반복** (V: 정점 개수)
 - 모든 간선 relax

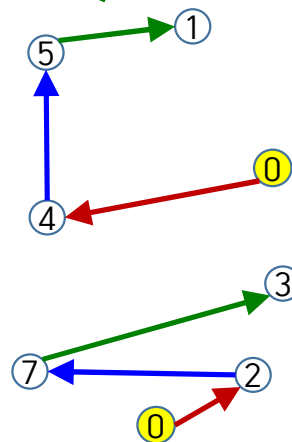
iteration 1
간선 1개 쓰는
최단경로 다 찾음



iteration 2
간선 2개 쓰는
최단경로 다 찾음



iteration 3
간선 3개 쓰는
최단경로 다 찾음



...

iteration V-1
간선 V-1개 쓰는
최단경로 다 찾음

정점 V개 쓰는 경로
0 - 0 - 0
= 정복되어야 함

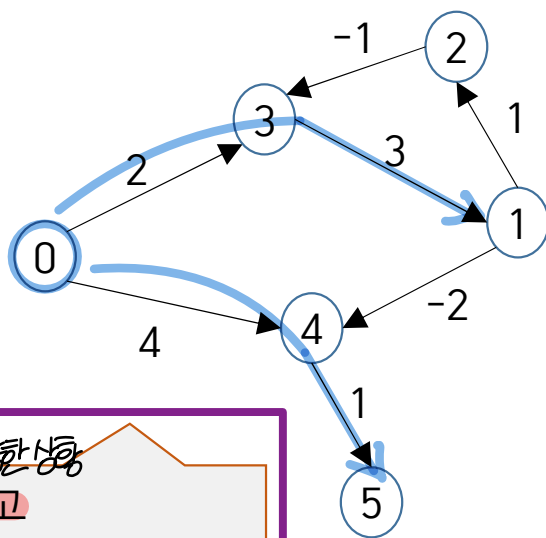
[Q] 간선 V-1개보다 많이 사용하는 최단경로는 나올 수 없는가? (즉, V-1회의 iteration으로 모든 목적지에 대한 최단경로 다 찾을 수 있나?)

정점이 V개에 따라서 최대 (V-1)번



Bellman-Ford 알고리즘

- 다음을 **V-1회 반복** (V: 정점 개수)
 - 모든 간선 relax



v	edgeTo[v]	distTo[v]	iteration1	iteration2	iteration3
0	None	0			
1	None	∞	3→1 5		
2	None	∞		1→2 6	
3	None	∞	0→3 2		
4	None	∞	0→4 4	더 짧아짐 1→4 3	
5	None	∞	4→5 5	4→5 4	

아무런 변화 X

Bellman-Ford는 가중치가 음수인 간선도 있는 그래프에 사용 가능함

cycle도 있고

음수 간선도 있는 그래프

[Q] 위 그래프에 Bellman-Ford를 수행해 edgeTo[], distTo[]의 변화를 기록하시오. 각 iteration에서는 정점 0→1→...→5 차례로 outgoing 간선을 relax 한다고 가정 하시오.

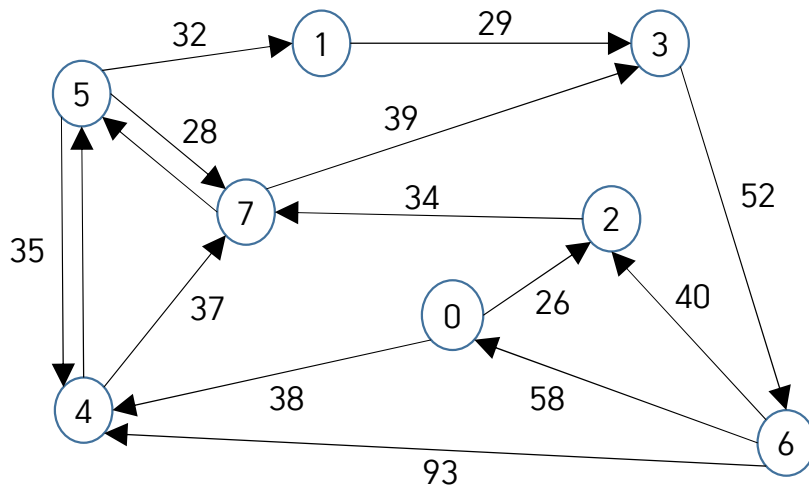
→ Dijkstra는 음의 간선이 있으면 답을 제대로 구할 수 X

Async의 cycle이 있으면 "



Bellman-Ford 알고리즘

- 다음을 **V-1회 반복** (V: 정점 개수)
 - 모든 간선 **relax**



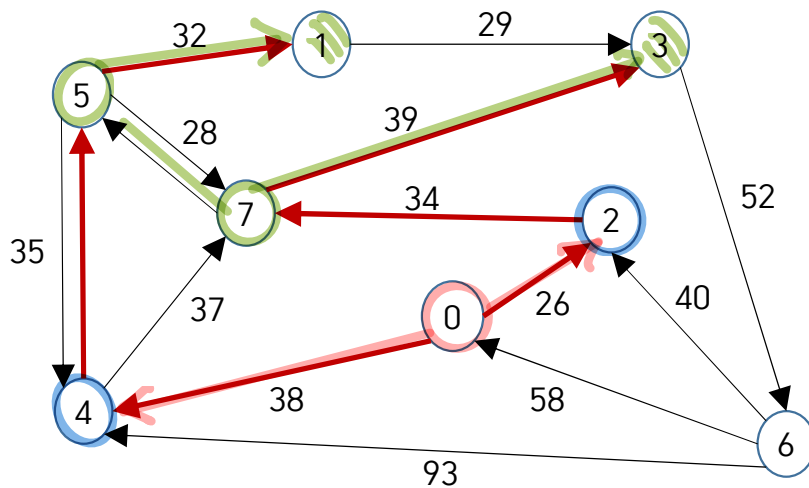
v	edgeTo[v]	distTo[v]
0	None	0
1	None	∞
2	None	∞
3	None	∞
4	None	∞
5	None	∞
6	None	∞
7	None	∞

[Q] 위 그래프에 **iteration 1**을 수행해 edgeTo[], distTo[]의 변화를 기록하고, 간선 1개 사용하는 최단 경로는 모두 찾음을 확인하시오. 각 iteration에서는 정점 0→1→...→7 차례로 outgoing 간선을 relax 한다고 가정 하시오.



Bellman-Ford 알고리즘

- 다음을 **V-1회 반복** (V: 정점 개수)
 - 모든 간선 **relax**



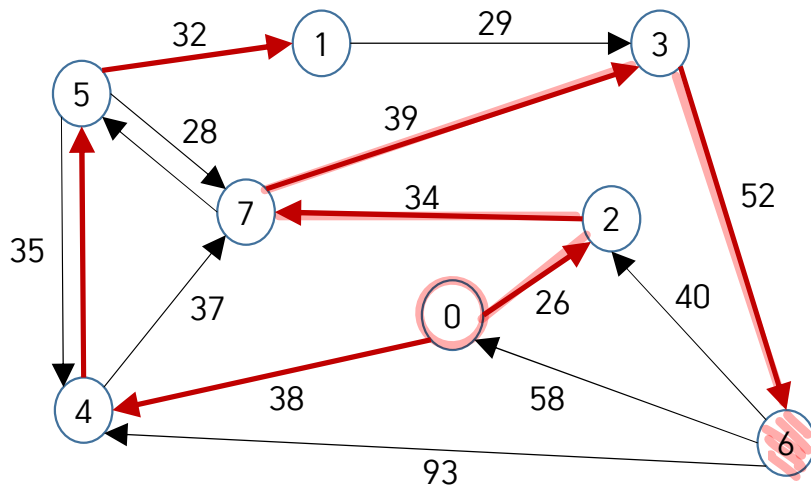
v	edgeTo[v]	distTo[v]
0	None	0
② 1	5→1	105
① 2	0→2	26
③ 3	7→3	99
① 4	0→4	38
② 5	4→5	73
6	None	∞
② 7	2→7	60

[Q] 위 그래프에 **iteration 2**를 수행해 edgeTo[], distTo[]의 변화를 기록하고, 간선 2개 사용하는 최단 경로는 모두 찾음을 확인하시오. 각 iteration에서는 정점 0→1→...→7 차례로 outgoing 간선을 relax 한다고 가정 하시오.



Bellman-Ford 알고리즘

- 다음을 **V-1회 반복** (V: 정점 개수)
 - 모든 간선 **relax**



v	edgeTo[v]	distTo[v]
0	None	0
1	5→1	105
2	0→2	26
3	7→3	99
4	0→4	38
5	4→5	73
6	3→6	151
7	2→7	60

[Q] 위 그래프에 **iteration 3**를 수행해 edgeTo[], distTo[]의 변화를 기록하고, 간선 3개 사용하는 최단 경로는 모두 찾음을 확인하시오. 각 iteration에서는 정점 0→1→...→7 차례로 outgoing 간선을 relax 한다고 가정 하시오.

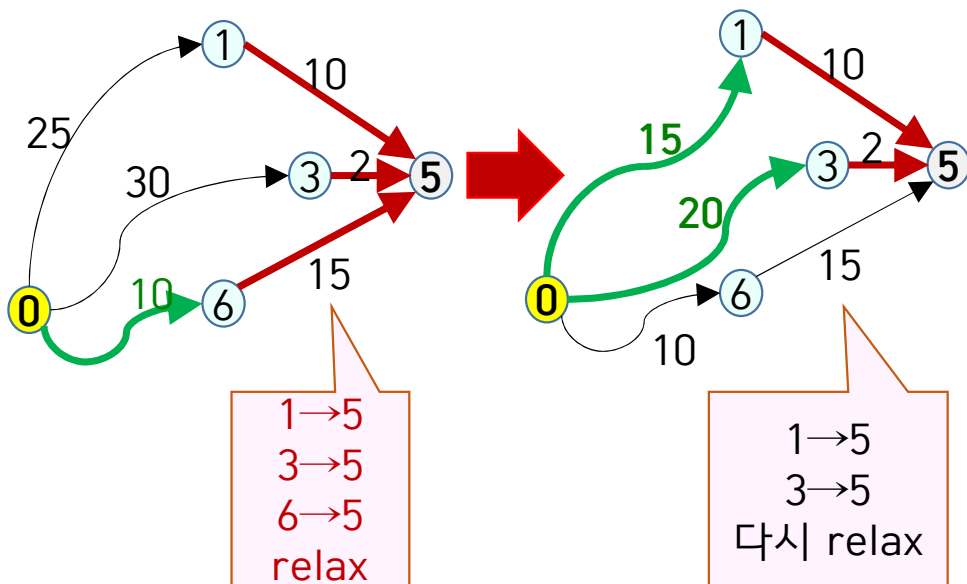


Shorted Paths on Weighted Digraphs

최단경로 탐색 방법과 활용도 이해

01. 퀴즈 풀이 & 예습 내용 복습 (이번 주 #1~3차 답안 공개)
02. 최단경로 문제의 기본 세팅 + 최단경로 탐색 방법의 공통점
03. Bellman-Ford 알고리즘
04. Dijkstra 알고리즘
05. Acyclic Shortest Path
06. Seam Carving
07. 실습: Seam Carving 구현

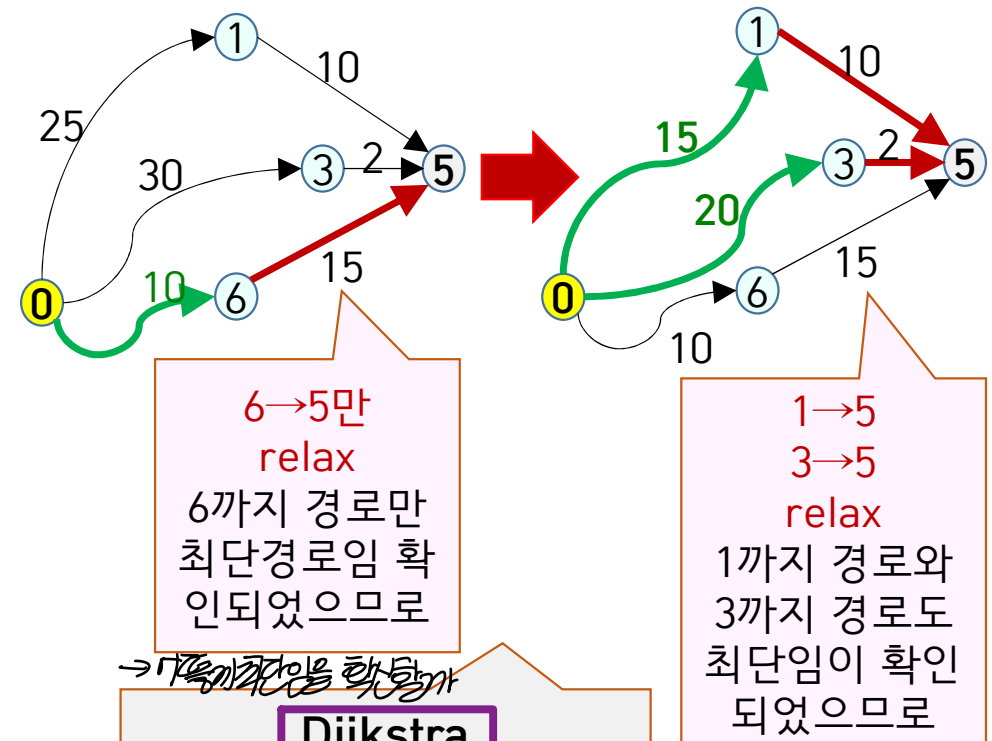
- $\text{relax}(x \rightarrow v)$ 에 의해
- v 까지 경로가 더 짧은 경로로 변경되었다면
- v 에서 나가는 각 간선 $e=v \rightarrow w$ 를 다시 relax해서
- w 까지의 경로도 수정해주어야 함



Bellman Ford

같은 간선 여러 차례 relax

- v 까지의 최단 경로 계산 끝난 후에
- v 에서 나가는 각 간선 $e=v \rightarrow w$ 를 relax 한다면
- e 를 단 한 번만 relax하면 되어 효율적일 것임



Dijkstra

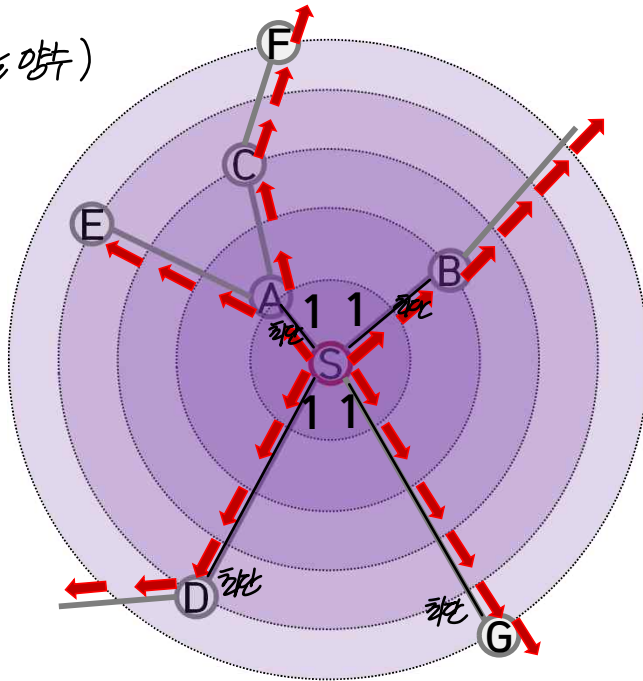
각 간선 한번씩만 relax



① 모든 방향으로 ② 같은 속도로 탐색

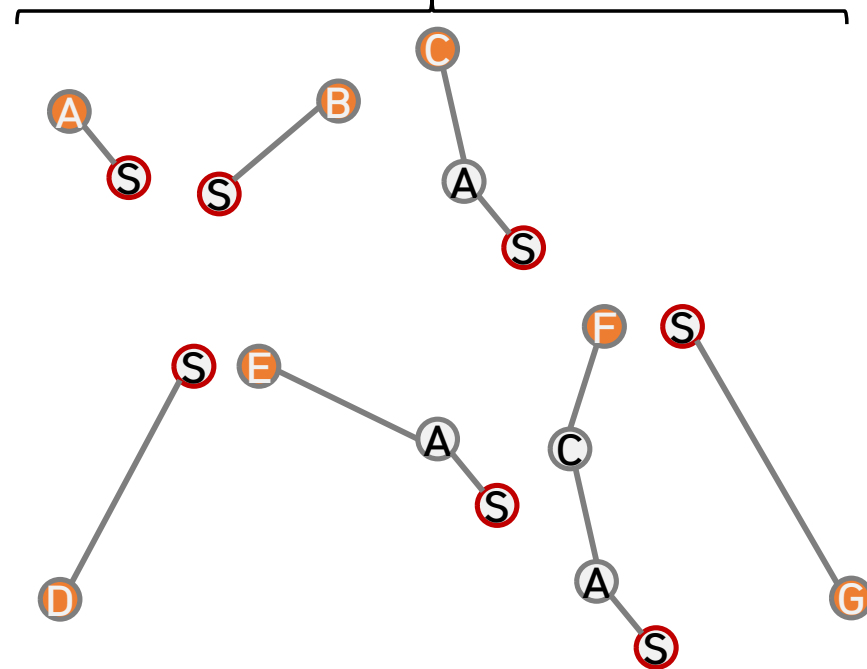
- 탐색 중 기존에 못 본 새로운 목적지 α 발견하면, 그때까지 거쳐 온 경로를 α 까지 최단경로로 기록

(모든 weight는 양수)



다음 길이의 모든 경로 탐색: 1 2 3 4 5

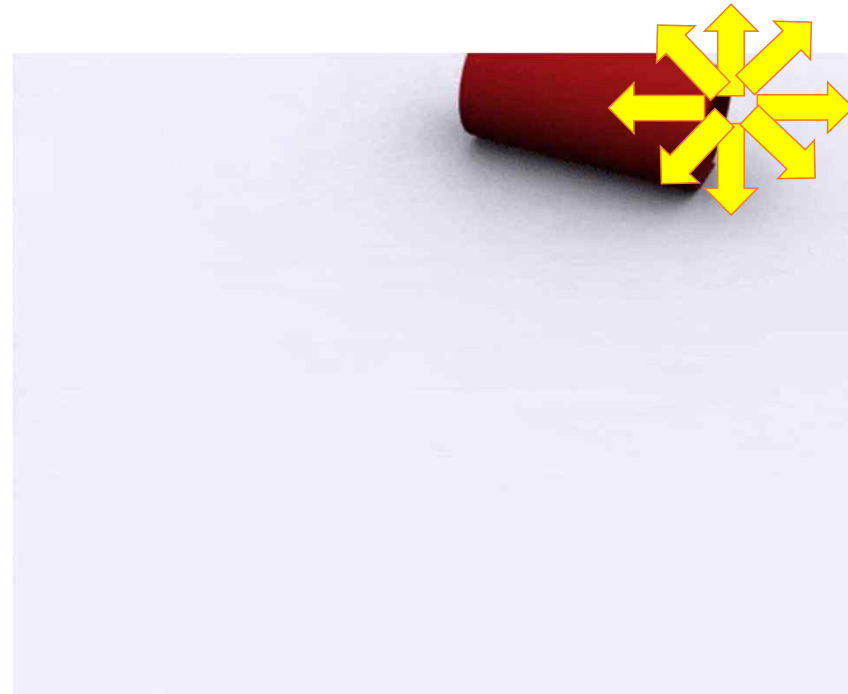
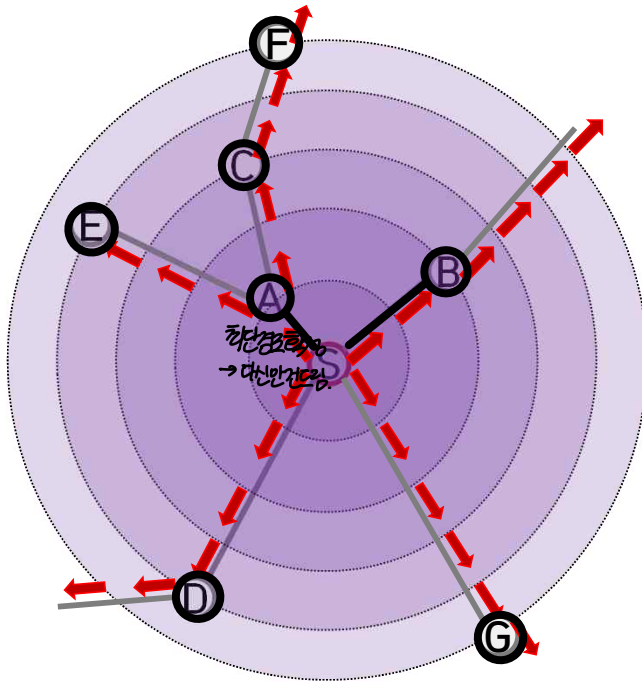
지금까지 발견한 최단 경로





① 모든 방향으로 ② 같은 속도로 카펫을 펼치는 것(혹은 구슬 굴리는 것)과 유사

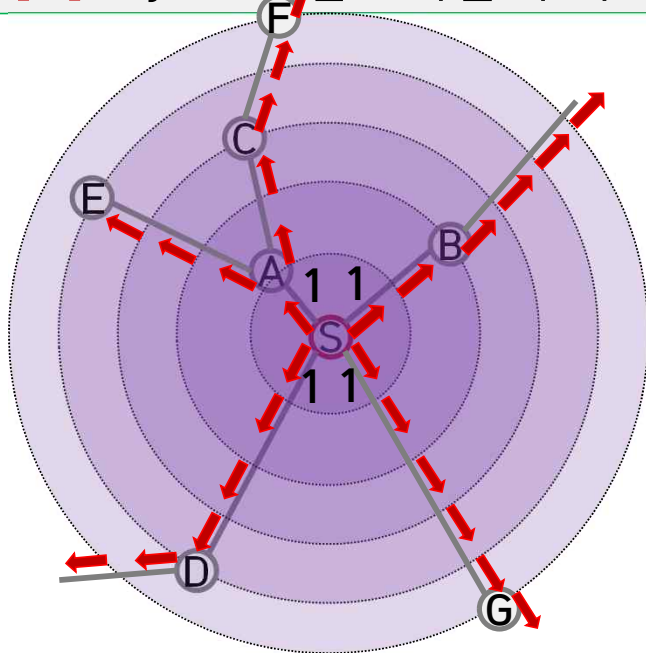
- 점진적으로 S로부터 더 멀리 있는 (더 긴) 최단 경로 찾게 됨 (예: 길이 1→2→3→4→...)



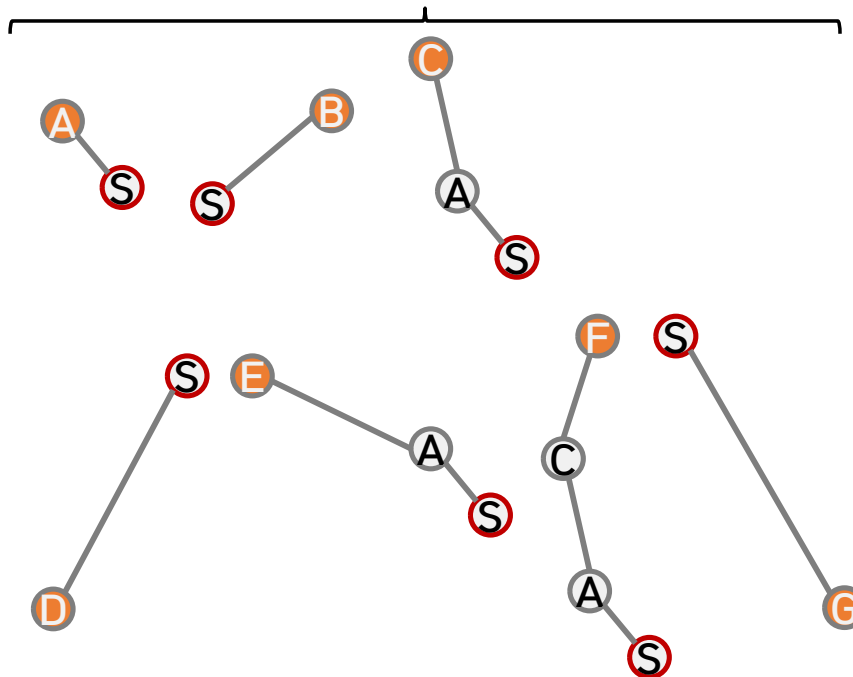
다음 길이의 모든 경로 탐색: 1 2 3 4 5

이 방식으로 진행해서 만나는 정점 v 까지 경로는 반드시 v 까지 최단경로라 확신할 수 있음
따라서 v 에서 outgoing하는 간선들을 한 번만 relax 해주고 다시 안 해도 됨

[Q] Dijkstra 알고리즘에 따라 찾은 경로는 왜 최단경로인가?



지금까지 발견한 최단 경로



다음 길이의 모든 경로 탐색: 1 2 3 4 5

- 간선 weight가 양의 정수라고 가정 (즉 모든 거리는 양의 정수)

[Q1] 모든 방향으로 1만큼 전진했을 때 찾은 경로는 최단 경로인가?

[Q2] [Q1]에서 못 도달한 목적지를 거리 2에서 만났다. 이 경로는 최단 경로인가?



Dijkstra 알고리즘을 edgeTo[], distTo[], relax() 사용해 표현

- 초기화: (모든 정점에 대해) $\text{edgeTo}[] = \text{None}$, (출발지 s) $\text{distTo}[s]=0$, (그 외) $\text{distTo}[t]=\infty$

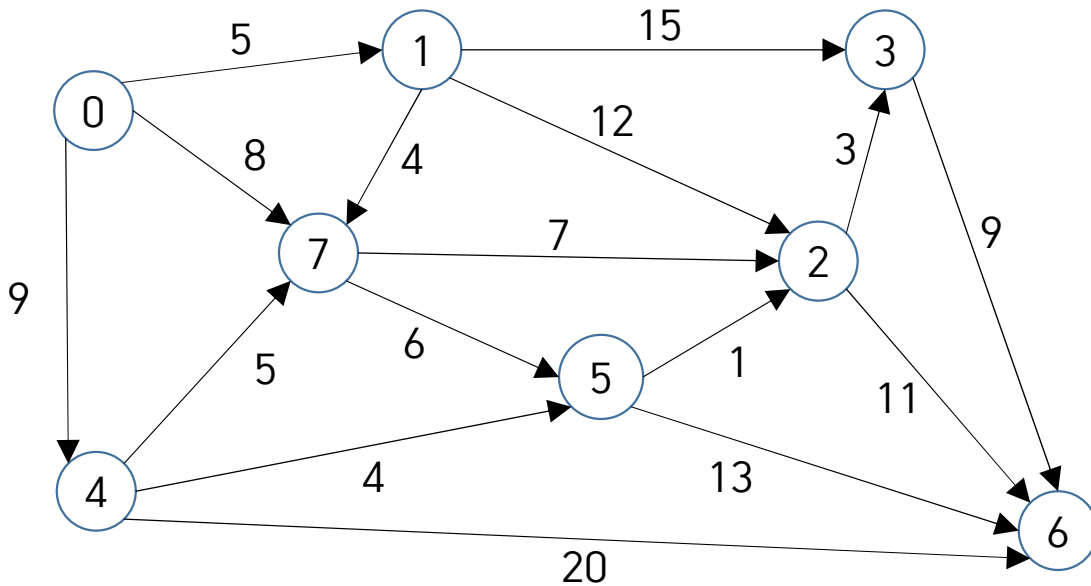
- 출발지 s 에서 먼저 도달할 수 있는 정점 순으로 선정

- 선정한 정점 v 를 SPT에 포함
- v 로부터 outgoing하는 각 간선 $e=v \rightarrow t$ 에 대해
- t 가 아직 SPT에 포함되지 않았다면 e 를 relax

한 번 relax한 간선은
더는 relax하지 않음
따라서 Bellman-Ford보다 효율적



- 초기화: (모든 정점에 대해) $\text{edgeTo}[] = \text{None}$, (출발지 s) $\text{distTo}[s]=0$, (그 외) $\text{distTo}[t]=\infty$
- 출발지 s 에서 먼저 도달할 수 있는 정점 순 선정 ($\text{distTo}[]$ 가장 작은 정점)
 - 선정한 정점 v 를 SPT에 포함하면서 이로부터 outgoing하는 모든 간선 $e=v \rightarrow t$ relax



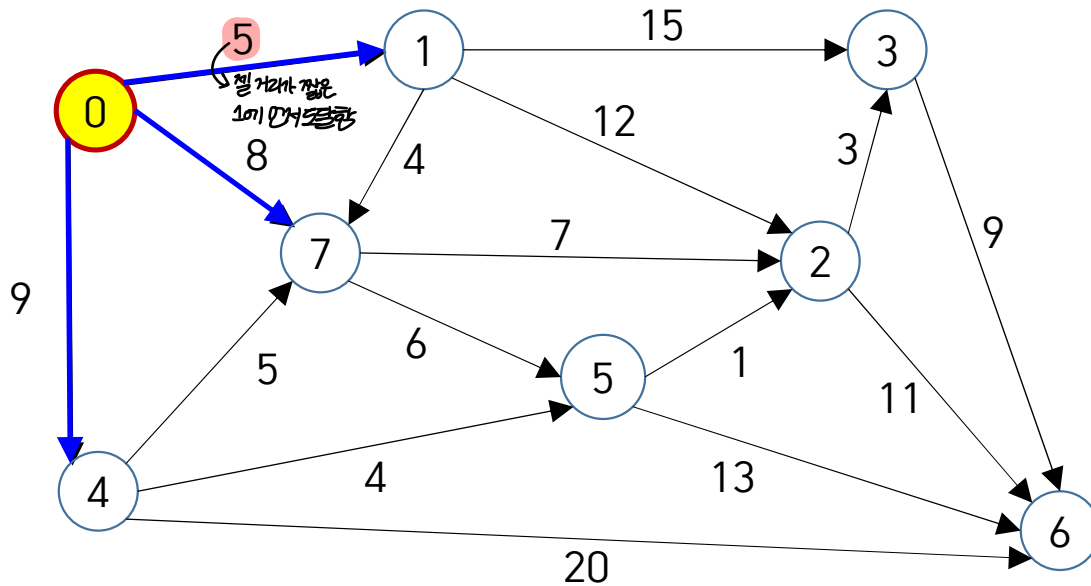
v	distTo[v]	edgeTo[v]
0	0	
1	∞	
2	∞	
3	∞	
4	∞	
5	∞	
6	∞	
7	∞	

None은 비어 있는 것으로 표시



Dijkstra 알고리즘

- 출발지 s 에서 먼저 도달할 수 있는 정점 순 선정 ($\text{distTo}[]$ 가장 작은 정점)
 - 선정한 정점 v 를 SPT에 포함하면서 이로부터 **outgoing**하는 모든 간선 중
 - 이미 SPT에 포함한 정점 t 로 가지 않는 간선 $e=v \rightarrow t$ relax



v	distTo[v]	edgeTo[v]
0	0	
1	5	0→1
2	∞	
3	∞	
4	9	0→4
5	∞	
6	∞	
7	8	0→7

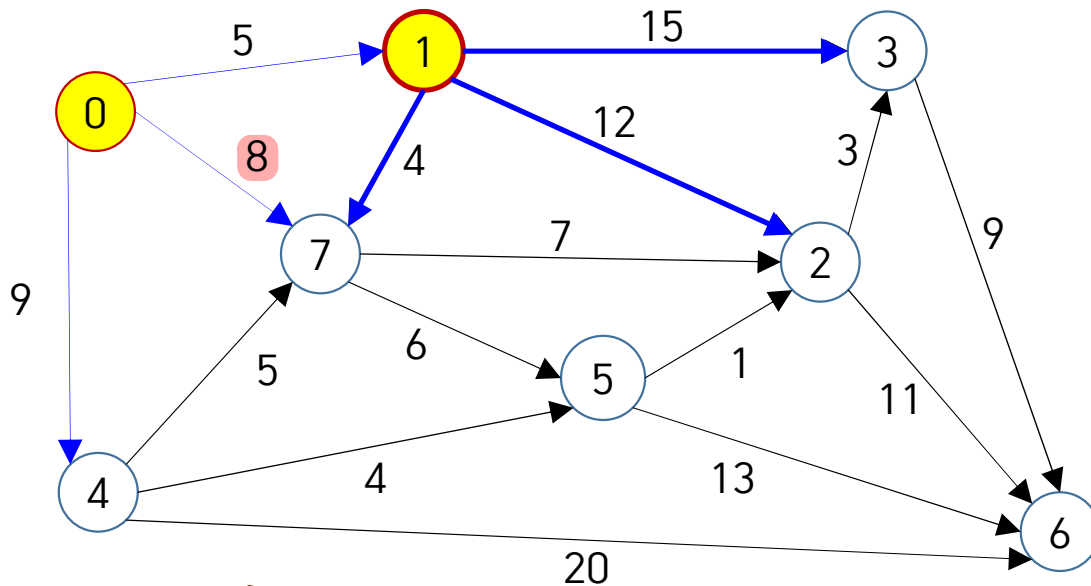
확인 fix

SPT에 포함 안 된 정점 중
 $\text{distTo}[] \neq \infty$ 목적지:
 0에서 같은 속도로 나아갔을 때
 다음 도달할 수 있는 후보들



Dijkstra 알고리즘

- 출발지 s 에서 먼저 도달할 수 있는 정점 순 선정 ($\text{distTo}[]$ 가장 작은 정점)
 - 선정한 정점 v 를 SPT에 포함하면서 이로부터 **outgoing**하는 모든 간선 중
 - 이미 SPT에 포함된 정점 t 로 가지 않는 간선 $e=v \rightarrow t$ relax



정점 1을 SPT에 포함함으로써
1까지 최단경로는 $0 \rightarrow 1$ 로 확정

v	distTo[v]	edgeTo[v]
0	0	
1	5	$0 \rightarrow 1$
2	17	$1 \rightarrow 2$
3	20	$1 \rightarrow 3$
4	9	$0 \rightarrow 4$
5	∞	
6	∞	
7	8	$0 \rightarrow 7$

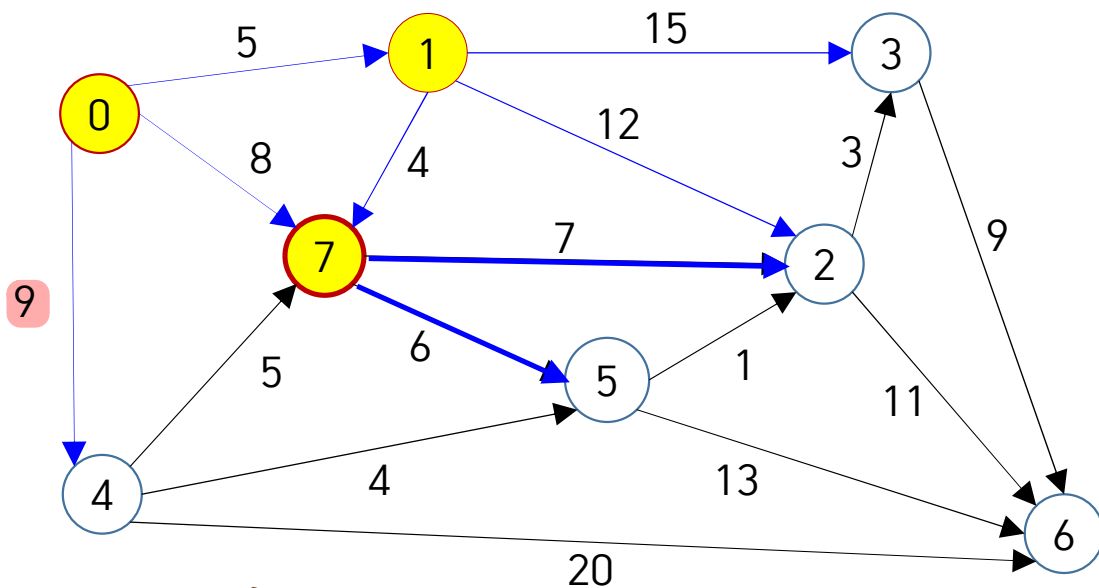
fix

SPT에 포함 안 된 정점 중
 $\text{distTo}[] \neq \infty$ 목적지:
거리 5에서 같은 속도로 나아갔을
때 다음 도달할 수 있는 후보들



Dijkstra 알고리즘

- 출발지 s 에서 먼저 도달할 수 있는 정점 순 선정 ($\text{distTo}[]$ 가장 작은 정점)
 - 선정한 정점 v 를 SPT에 포함하면서 이로부터 **outgoing**하는 모든 간선 중
 - 이미 SPT에 포함한 정점 t 로 가지 않는 간선 $e=v \rightarrow t$ relax



정점 7을 SPT에 포함함으로써
7까지 최단경로는 $0 \rightarrow 7$ 로 확정

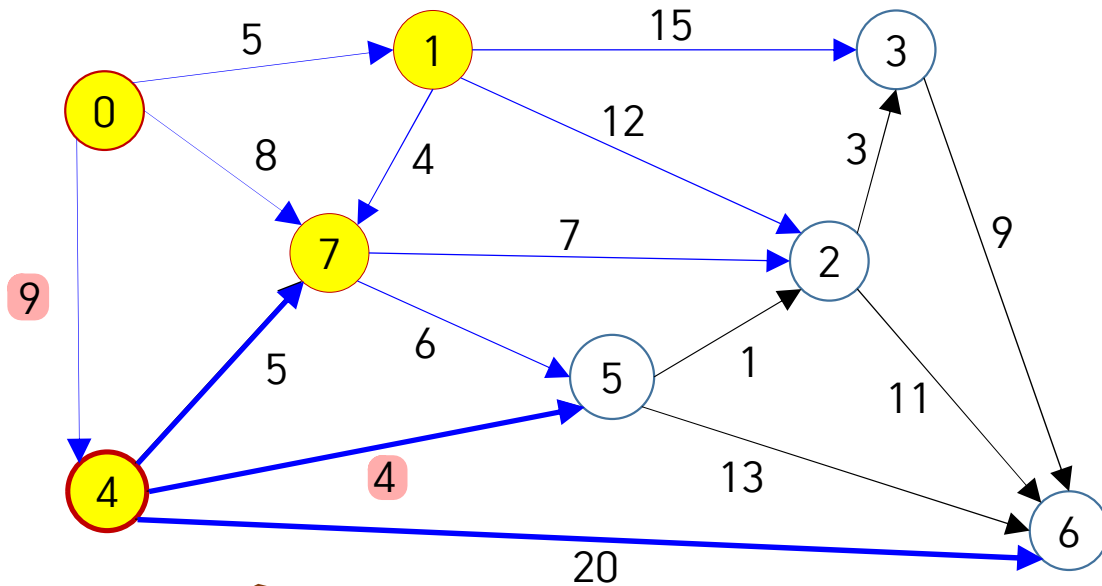
v	distTo[v]	edgeTo[v]
0	0	
1	5	$0 \rightarrow 1$
2	15	$7 \rightarrow 2$
3	20	$1 \rightarrow 3$
4	9	$0 \rightarrow 4$
5	14	$7 \rightarrow 5$
6	∞	
7	8	$0 \rightarrow 7$

SPT에 포함 안 된 정점 중
 $\text{distTo}[] \neq \infty$ 목적지:
거리 8에서 같은 속도로 나아갔을
때 다음 도달할 수 있는 후보들



Dijkstra 알고리즘

- 출발지 s 에서 먼저 도달할 수 있는 정점 순 선정 ($\text{distTo}[]$ 가장 작은 정점)
 - 선정한 정점 v 를 SPT에 포함하면서 이로부터 **outgoing**하는 모든 간선 중
 - 이미 SPT에 포함한 정점 t 로 가지 않는 간선 $e=v \rightarrow t$ relax



정점 4를 SPT에 포함함으로써
4까지 최단경로는 $0 \rightarrow 4$ 로 확정

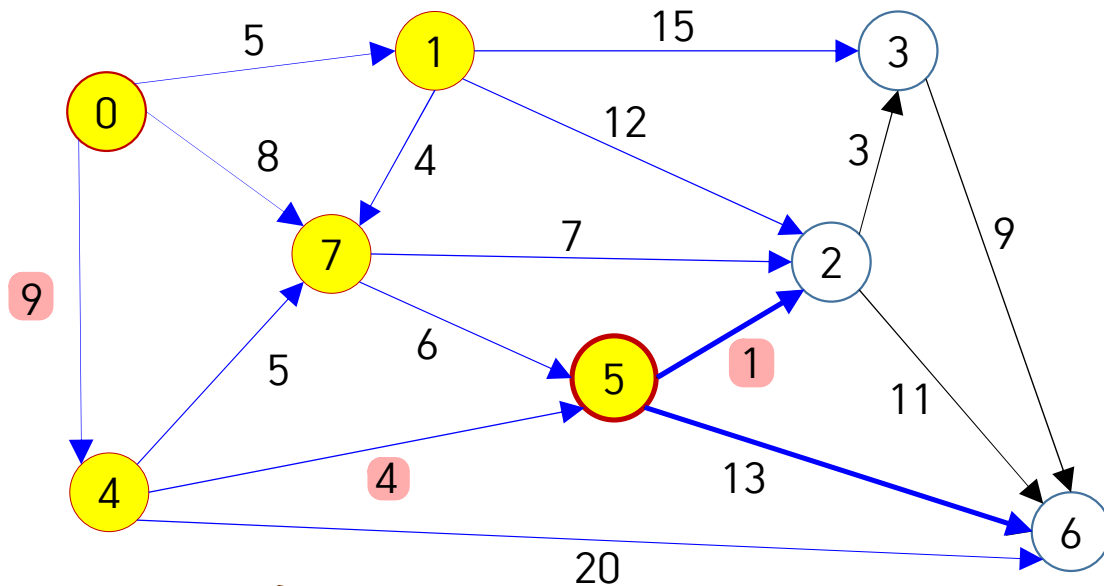
v	distTo[v]	edgeTo[v]
0	0	
1	5	$0 \rightarrow 1$
2	15	$7 \rightarrow 2$
3	20	$1 \rightarrow 3$
4	9	$0 \rightarrow 4$
5	13	$4 \rightarrow 5$
6	29	$4 \rightarrow 6$
7	8	$0 \rightarrow 7$

SPT에 포함 안 된 정점 중
 $\text{distTo}[] \neq \infty$ 목적지:
거리 9에서 같은 속도로 나아갔을
때 다음 도달할 수 있는 후보들



Dijkstra 알고리즘

- 출발지 s 에서 먼저 도달할 수 있는 정점 순 선정 ($\text{distTo}[]$ 가장 작은 정점)
 - 선정한 정점 v 를 SPT에 포함하면서 이로부터 **outgoing**하는 모든 간선 중
 - 이미 SPT에 포함한 정점 t 로 가지 않는 간선 $e=v \rightarrow t$ relax



정점 5를 SPT에 포함함으로써
5까지 최단경로는 $0 \rightarrow 4 \rightarrow 5$ 로 확정

v	distTo[v]	edgeTo[v]
0	0	
1	5	$0 \rightarrow 1$
2	14	$5 \rightarrow 2$
3	20	$1 \rightarrow 3$
4	9	$0 \rightarrow 4$
5	13	$4 \rightarrow 5$
6	26	$5 \rightarrow 6$
7	8	$0 \rightarrow 7$

PQ에 저장된 정점의 key
($\text{distTo}[]$) 변경될 수 있으므로
Indexed minPQ 사용

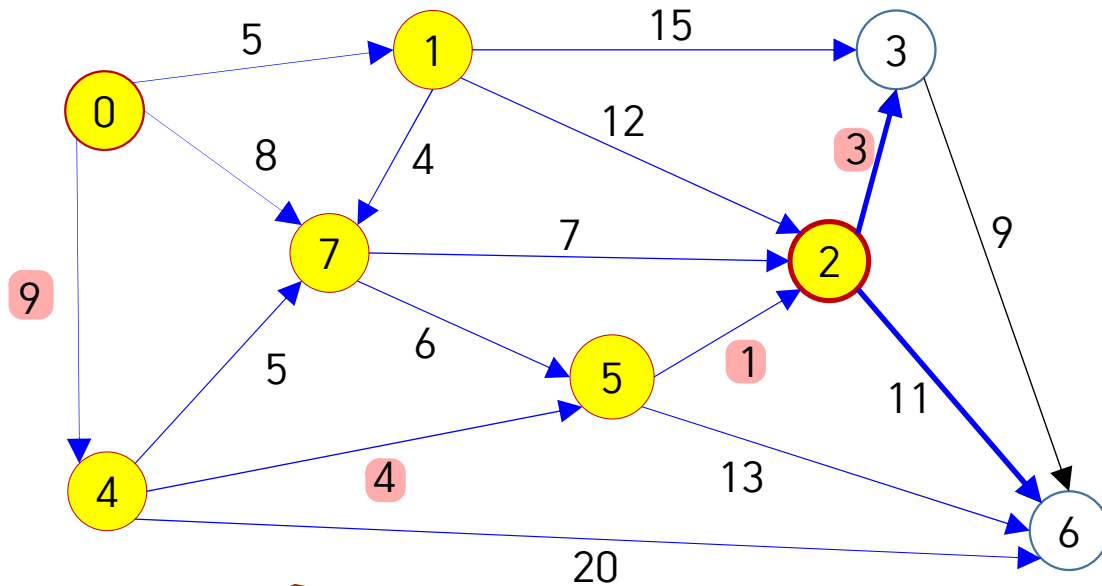
매 iteration마다
(SPT에 포함 안 된 정점 중)
 $\text{distTo}[]$ 가장 작은 정점
선정하므로
minPQ 사용해 구현

SPT에 포함 안 된 정점 중
 $\text{distTo[]} \neq \infty$ 목적지:
거리 13에서 같은 속도로 나아갔을
때 다음 도달할 수 있는 후보들



Dijkstra 알고리즘

- 출발지 s 에서 먼저 도달할 수 있는 정점 순 선정 ($\text{distTo}[]$ 가장 작은 정점)
 - 선정한 정점 v 를 SPT에 포함하면서 이로부터 **outgoing**하는 모든 간선 중
 - 이미 SPT에 포함된 정점 t 로 가지 않는 간선 $e=v \rightarrow t$ relax



정점 2를 SPT에 포함함으로써
2까지 최단경로는 $0 \rightarrow 4 \rightarrow 5 \rightarrow 2$ 로 확정

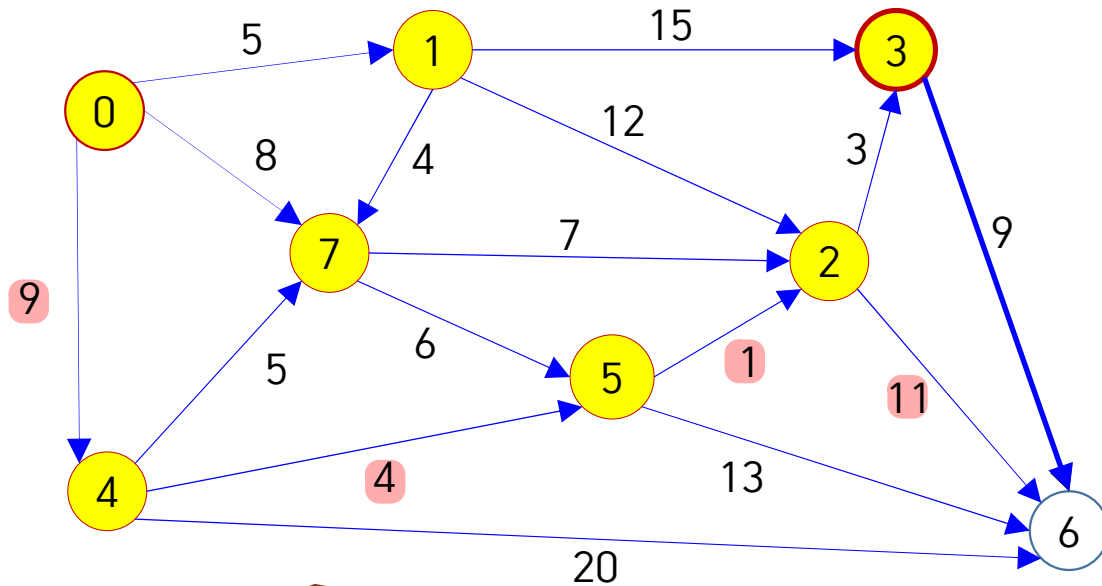
v	distTo[v]	edgeTo[v]
0	0	
1	5	$0 \rightarrow 1$
2	14	$5 \rightarrow 2$
3	17	$2 \rightarrow 3$
4	9	$0 \rightarrow 4$
5	13	$4 \rightarrow 5$
6	25	$2 \rightarrow 6$
7	8	$0 \rightarrow 7$

SPT에 포함 안 된 정점 중
 $\text{distTo}[] \neq \infty$ 목적지:
거리 14에서 같은 속도로 나아갔을
때 다음 도달할 수 있는 후보들



Dijkstra 알고리즘

- 출발지 s 에서 먼저 도달할 수 있는 정점 순 선정 ($\text{distTo}[]$ 가장 작은 정점)
 - 선정한 정점 v 를 SPT에 포함하면서 이로부터 **outgoing**하는 모든 간선 중
 - 이미 SPT에 포함한 정점 t 로 가지 않는 간선 $e=v \rightarrow t$ relax



정점 3을 SPT에 포함함으로써
3까지 최단경로는 $0 \rightarrow 4 \rightarrow 5 \rightarrow 2 \rightarrow 3$ 으로 확정

v	distTo[v]	edgeTo[v]
0	0	
1	5	$0 \rightarrow 1$
2	14	$5 \rightarrow 2$
3	17	$2 \rightarrow 3$
4	9	$0 \rightarrow 4$
5	13	$4 \rightarrow 5$
6	25	$2 \rightarrow 6$
7	8	$0 \rightarrow 7$

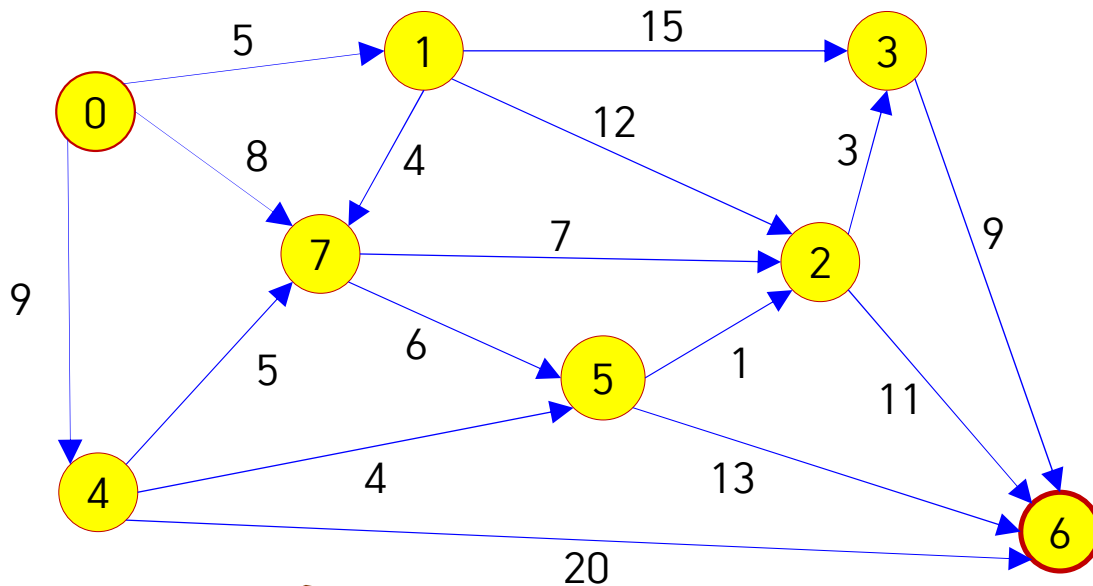
간선 $3 \rightarrow 6$ 을 relax 시도해도
기존 경로보다 낮지 않으므로
변화 없음

SPT에 포함 안 된 정점 중
 $\text{distTo}[] \neq \infty$ 목적지:
거리 17에서 같은 속도로 나아갔을
때 다음 도달할 수 있는 후보들



Dijkstra 알고리즘

- 출발지 s 에서 먼저 도달할 수 있는 정점 순 선정 ($\text{distTo}[]$ 가장 작은 정점)
 - 선정한 정점 v 를 SPT에 포함하면서 이로부터 **outgoing**하는 모든 간선 중
 - 이미 SPT에 포함한 정점 t 로 가지 않는 간선 $e=v \rightarrow t$ relax



정점 6을 SPT에 포함함으로써
6까지 최단경로는 $0 \rightarrow 4 \rightarrow 5 \rightarrow 2 \rightarrow 6$ 으로 확정

각 간선을 단 한번씩만 relax 했음에 유의
(따라서 Bellman-Ford보다 효율적)

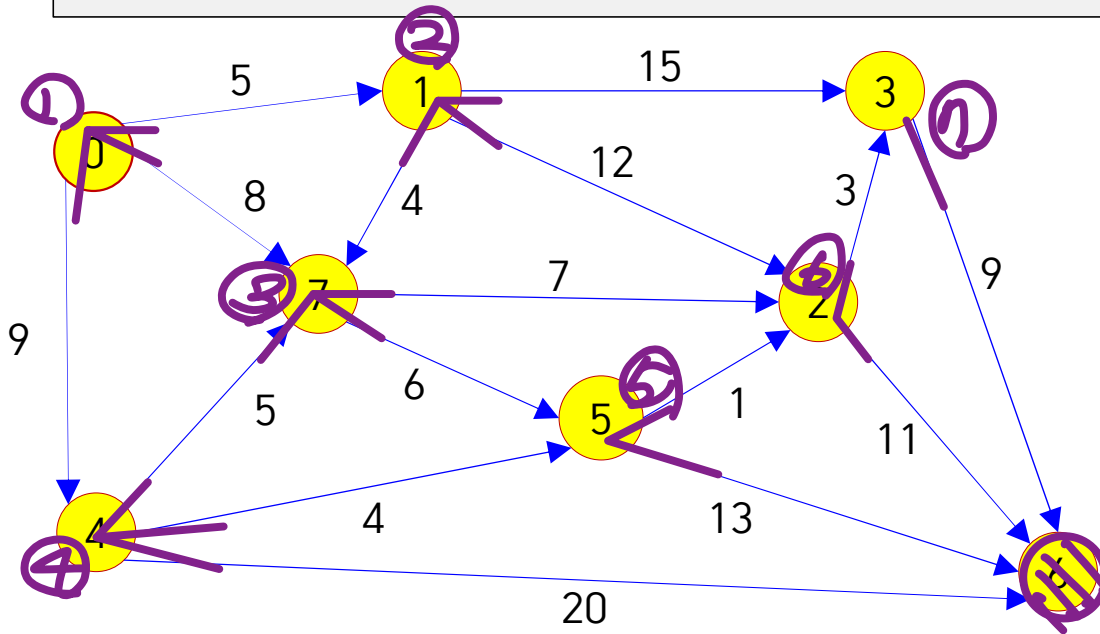
v	distTo[v]	edgeTo[v]
0	0	
1	5	$0 \rightarrow 1$
2	14	$5 \rightarrow 2$
3	17	$2 \rightarrow 3$
4	9	$0 \rightarrow 4$
5	13	$4 \rightarrow 5$
6	25	$2 \rightarrow 6$
7	8	$0 \rightarrow 7$

모든 정점이
SPT에 포함되었으므로
알고리즘 종료



Dijkstra 알고리즘

- 출발지 s 에서 먼저 도달할 수 있는 정점 순 선정 ($\text{distTo}[]$ 가장 작은 정점)
 - 선정한 정점 v 를 SPT에 포함하면서 이로부터 **outgoing**하는 모든 간선 중
 - 이미 SPT에 포함한 정점 t 로 가지 않는 간선 $e=v \rightarrow t$ relax



v	distTo[v]	edgeTo[v]
0	0	
1	5	0→1
2	14	5→2
3	17	2→3
4	9	0→4
5	13	4→5
6	25	2→6
7	8	0→7



Prim 알고리즘과 유사함.



```
class DijkstraSP(SP): # Inherit SP class
```

```
def __init__(self, g, s):
```

```
    super().__init__(g, s) # run the constructor of the parent class
```

```
    self.pq = IndexMinPQ(g.V)
```

```
    self.pq.insert(s, 0)
```

Indexed minPQ에 출발지 s (거리 0) 추가

```
    while not self.pq.isEmpty():
```

```
        # select vertices in order of distance from s
```

```
        dist, v = self.pq.delMin()
```

```
        for e in self.g.adj[v]:
```

```
            self.relax(e)
```

매 iteration마다
(SPT에 포함 안 되었으면서)
distTo[] 가장 작은 정점 v 선정해
v의 모든 outgoing 간선 relax

Indexed minPQ에는
아직 SPT에 추가 안되었으면서
distTo[] $\neq \infty$ 인 정점
(즉 다음에 도달 가능한 목적지)
담겨 있음

```
def relax(self, e):
```

```
    if self.distTo[e.w] > self.distTo[e.v] + e.weight:
```

```
        self.distTo[e.w] = self.distTo[e.v] + e.weight
```

```
        self.edgeTo[e.w] = e
```

```
        if self.pq.contains(e.w): self.pq.decreaseKey(e.w, self.distTo[e.w])
```

```
        else: self.pq.insert(e.w, self.distTo[e.w])
```

relax해서 distTo[] 변한 목적지 w 있다면
(w가 이미 minPQ에 있다면) key 변경
(w가 아직 minPQ에 없다면) 추가



Dijkstra 알고리즘 성능: minPQ 성능에 의존

- V: 정점(Vertex) 개수
- E: 간선(Edge) 개수
- Indexed minPQ로 **binary heap** 사용하는 경우, 아래와 같이 ($E \gg V$ 라면) $\sim E \log V$

Operation	1회 비용	필요한 횟수	
PQ, insert	$\log v$	V	각 정점은 단 1번 PQ에 추가 or 삭제 되므로 (\cdot 정점 개수)
PQ, deleteMin	$\log v$	V	
PQ, decreaseKey	$\log v$	E	relax를 간선 개수 E만큼 하므로

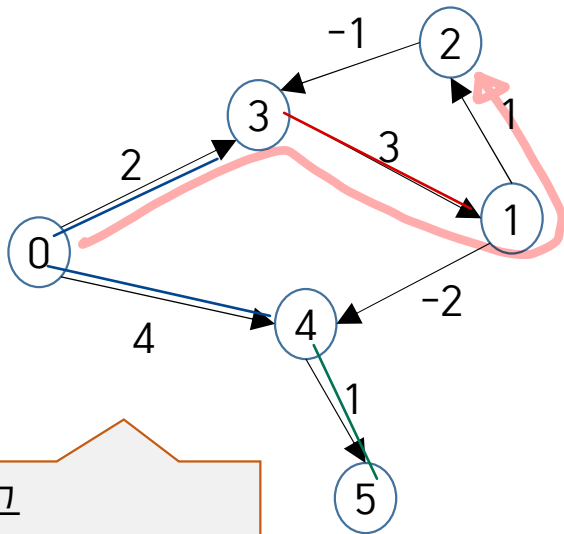


	Bellman-Ford	Dijkstra
방법	E개 간선 모두를 relax하는 것 정점 수 $V-1$ 회 반복	출발지 s 에서 가까운 정점 v 순으로 v 의 모든 간선 relax
시간 복잡도	$\sim E \times V$	$\sim E \times \log V$
간선에 negative weight 허용?	(한 번 거친 곳도 계속 다시 relax 하므로) 음수 weight인 간선 있어도 동작	(한 번 지나간 곳은 다시 보지 않으므로) 간선 weight ≥ 0 인 경우만 최단경로 찾을 수 있음 음수 weight 있다면 iteration 지날 때마다 더 먼 경로 발견한다는 가정 어긋남
Cycle 허용?	Yes. Cycle 있는 directed graph에서도 최단 경로 찾을 수 있음	Yes. Cycle 있는 directed graph에서도 최단 경로 찾을 수 있음

최단경로 찾기를 실패한 Case

Dijkstra 알고리즘

- 출발지 s 에서 먼저 도달할 수 있는 정점 순 선정 ($distTo[]$ 가장 작은 정점)
 - 선정한 정점 v 를 SPT에 포함하면서 이로부터 **outgoing**하는 모든 간선 중
 - 이미 SPT에 포함한 정점 t 로 가지 않는 간선 $e=v \rightarrow t$ relax



v	edgeTo[v]	distTo[v]
①0	None	0
1	None	∞
2	None	∞
③3	None	∞
④4	None	∞
5	None	∞

3 → 1 | 5

1 → 2 | 6

0 → 3 | 2

0 → 4 | 4

4 → 5 | 5

→ 최단이 아님!

(Dijkstra 작동 FX)

cycle도 있고
~~음수 간선~~도 있는 그래프

[Q] 위 그래프에 Dijkstra를 수행해 $edgeTo[]$, $distTo[]$ 의 변화를 기록하시오.
일부 정점에 대해서는 최단 경로를 찾지 못함을 확인하고, 이유를 생각해 보시오.



Shorted Paths on Weighted Digraphs

최단경로 탐색 방법과 활용도 이해

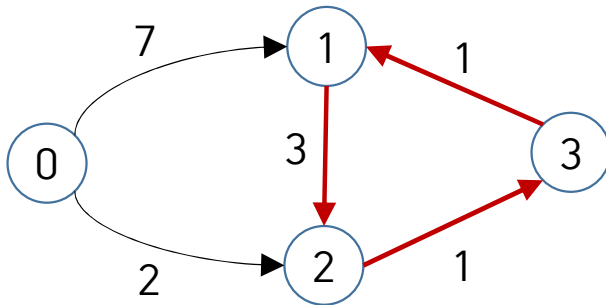
01. 퀴즈 풀이 & 예습 내용 복습 (이번 주 #1~3차 답안 공개)
02. 최단경로 문제의 기본 세팅 + 최단경로 탐색 방법의 공통점
03. Bellman-Ford 알고리즘
04. Dijkstra 알고리즘
05. Acyclic Shortest Path
06. Seam Carving
07. 실습: Seam Carving 구현

Cycle 없는 간단한 그래프에 대한
효율적인 최단경로 탐색 방법

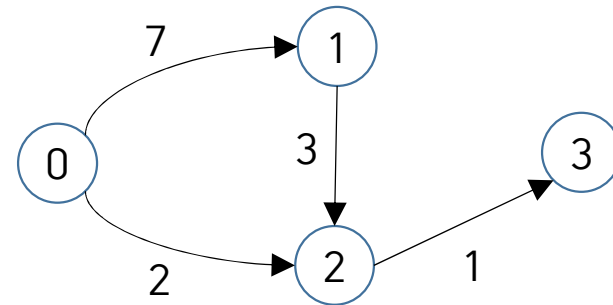


cycle 없는 그래프도 많음. 입력 그래프에 **cycle 없다면** Bellman-Ford나 Dijkstra보다 **더 간단**하게 (더 효율적으로) 최단경로 찾을 수 있는가? **Yes!**

- cycle **있으면** 더 다양한 경로 가능
- 이들 다 고려하기 위해 복잡도 ↑



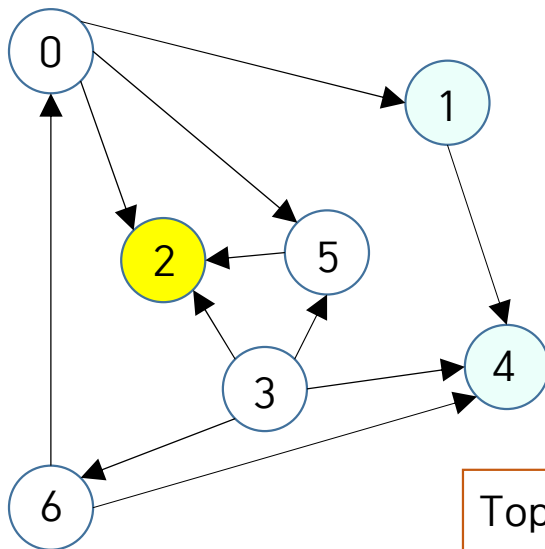
- cycle **없으면** 가능한 경로 수 ↓
- 더 **간단**한 탐색 방법 존재



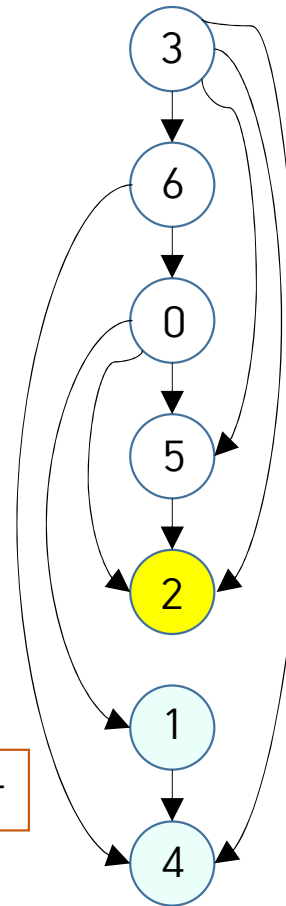


Cycle 없는 그래프에는 topological order 존재

- Cycle 없는 digraph에서
- Topological order: $v \rightarrow w$ 경로 있다면 v 후에 w 가 오도록 하는 순서
- Topological order 순으로 정점 나열하는 것을 topological sort라 함



Topological Sort한 결과



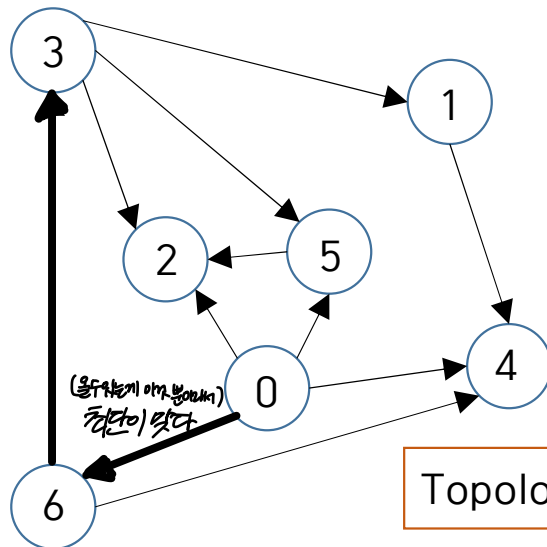
DFS 사용하면
~E+V 시간에
찾을 수 있음



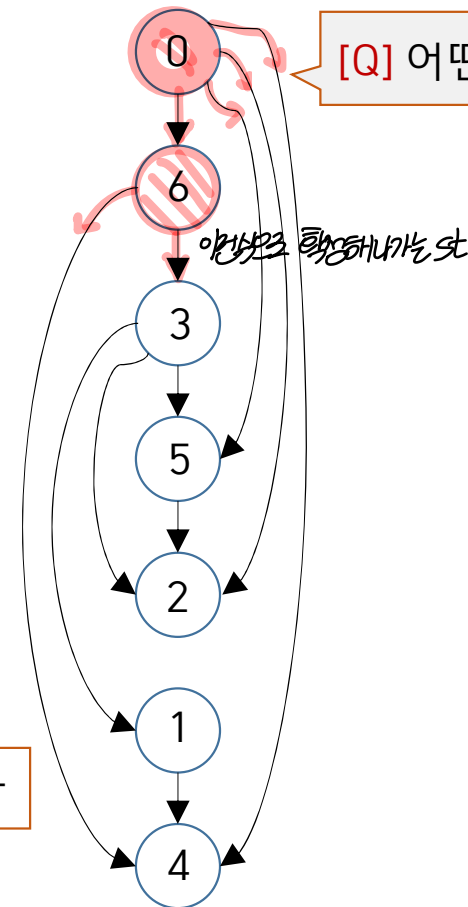
Acyclic Shorted Path 알고리즘: topological order 순으로 relax

- 초기화: (모든 정점에 대해) $\text{edgeTo}[] = \text{None}$, (출발지 s) $\text{distTo}[s]=0$, (그 외) $\text{distTo}[t]=\infty$

- Topological order 순으로 정점 v 선정
 - v 로부터 outgoing하는 모든 간선 $e=v \rightarrow t$ relax



Topological Sort한 결과





```
class AcyclicSP(SP):  
    def __init__(self, g, s):  
        super().__init__(g, s) # run the constructor of the parent class  
        tpOrder = topologicalSort(g)  
        for v in tpOrder:  
            for e in self.g.adj[v]:  
                self.relax(e)
```

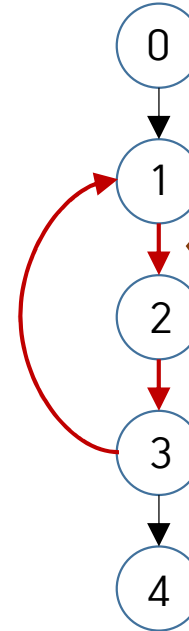
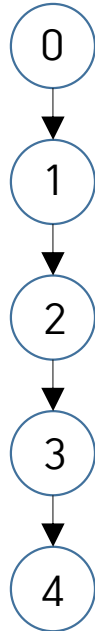
topological order 구하기

topological order 순으로 v 선정해
v의 모든 outgoing 간선 relax



[Q] 왜 Topological order 순으로 relax하면 항상 최단 경로 찾을 수 있나?

새로운 경로는
기존에 찾은 경로에
topological order 순으로
간선 더해서만 나올 수 있으므로
이 순서로 고려하면
모든 가능한 경로 다 고려



Cycle 있다면
topological order
존재하지 않으므로
다른 가능한 순서 사용해야 함
(예: Dijkstra 알고리즘처럼 출발
지로부터 가까운 순서)



Bellman-Ford

- 다음을 **V-1회 반복** (V: 정점 개수)
 - 모든 간선 relax

각 간선을 $\sim V$ 번 relax
 $\sim (E \times V)$

Acyclic Shortest Path

- **Topological order** 순 정점 v 선정
- v로부터 outgoing하는 모든 간선 $e=v \rightarrow t$ relax

Topological sort &
각 간선을 한 번만 relax
 $\sim (E + V)$
따라서 Bellman-Ford보다 효율적



Dijkstra

- 출발지에서 가까운 순 정점 v 선정
- v 로부터 outgoing하는 모든 간선 $e=v \rightarrow t$ relax

각 간선을 한 번만 relax
relax마다 PQ 접근하므로 $\log V$ 비용
 $\sim E \times \log V$

Acyclic Shortest Path

- Topological order 순 정점 v 선정
- v 로부터 outgoing하는 모든 간선 $e=v \rightarrow t$ relax

각 간선 한 번만 relax &
① Topological sort (DFS)
 $\sim (E + V)$
따라서 Dijkstra보다 효율적

<relax 규칙>

- ① 정점 v 까지 더 짧은 경로 찾았다면
 v 의 outgoing edge $v \rightarrow w$ relax
- ② v 까지 최단경로 찾았을 때
 $v \rightarrow w$ relax하면, 더는 relax 불필요

Bellman Ford	EV	
Dijkstra	$E \log V$	비중 ≥ 0
Acyclic	$E + V$	사이클 없을 때

② Topological order 모든 정점 relax
(가벼워)
 $\sim (V + E)$



- V: 정점(Vertex) 개수
- E: 간선(Edge) 개수
- Indexed minPQ로 **binary heap** 사용하는 경우, 아래와 같이 ($E \gg V$ 라면) $\sim E \log V$

Operation	1회 비용	필요한 횟수
PQ, insert	$\log V$	V <small>각 정점은 단 1번 PQ에 추가 or 삭제 되므로</small>
PQ, deleteMin	$\log V$	V <small>(- 정점 개수)</small>
PQ, decreaseKey	$\log V$	E <small>relax를 간선 개수 E만큼 하므로</small>

	Bellman-Ford	Dijkstra	Acyclic SP
방법	E개 간선 모두를 relax하는 것 정점 수 V-1 회 반복	출발지 s에서 가까운 정점 v 순으로 v의 모든 간선 relax	Topological order 순으로 정점 v 선택해 v의 모든 간선 relax
시간 복잡도	$\sim E \times V$	$\sim E \times \log V$	$\sim E + V$
간선에 negative weight 허용?	(한 번 거친 곳도 계속 다시 relax 하므로) 음수 weight인 간선 있어도 동작	(한 번 지나간 곳 다시 보지 않으므로) 간선 weight ≥ 0인 경우만 최단경로 찾음 음수 weight 있다면 iteration 지날 때마다 더 먼 경로 발견 한다는 가정 어긋남	(한 번 지나간 곳은 다시 돌아 오는 길 없으므로) 음수 weight인 간선 있어도 동작
Cycle 허용?	Yes Cycle 있는 directed graph에서도 최단경로 찾음	Yes Cycle 있는 directed graph에서도 최단경로 찾음	No. Cycle 있으면 topological order 존재 안 함

Bellman-Ford

- 다음을 **V-1회 반복** (V: 정점 개수)
- 모든 간선 relax

각 간선을 $\sim V$ 번 relax
 $\sim (E \times V)$

Acyclic Shortest Path

- **Topological order** 순 정점 v 선정
- v로부터 **outgoing**하는 모든 간선 $e=v \rightarrow t$ relax

Topological sort &
각 간선을 한 번만 relax
 $\sim (E + V)$
따라서 Bellman-Ford보다 효율적



Shorted Paths on Weighted Digraphs

최단경로 탐색 방법과 활용도 이해

01. 퀴즈 풀이 & 예습 내용 복습 (이번 주 #1~3차 답안 공개)

02. 최단경로 문제의 기본 세팅 + 최단경로 탐색 방법의 공통점

03. Bellman-Ford 알고리즘

04. Dijkstra 알고리즘

05. Acyclic Shortest Path

06. Seam Carving

지금까지 배운 내용 기반으로
어플리케이션 상황에 맞는 최단경로 알고리즘 설계해 보기

07. 실습: Seam Carving 구현



Seam Carving: 이미지의 크기 조절 시 중요한 부분 자동 인식해 최대한 보존하며 조절하는 방식 (web browser, 휴대폰 등에서 활용)

62



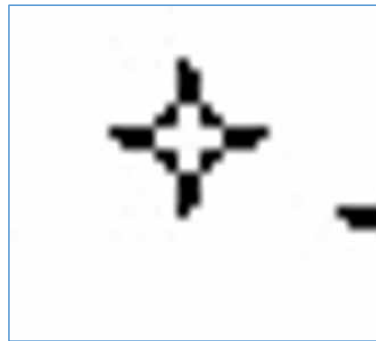


Seam Carving: 이미지의 크기 조절 시 중요한 부분 자동 인식해 최대한 보존하며 조절하는 방식 (web browser, 휴대폰 등에서 활용)

<원본>



<Resize 후>



① Resize 시
잘라내는 방식
(crop)



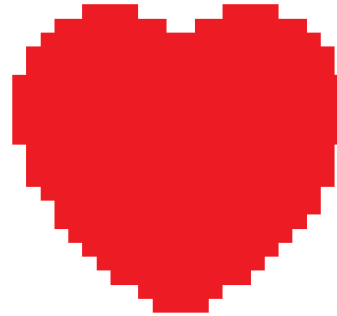
② Resize 시
찌그러뜨리는 방식



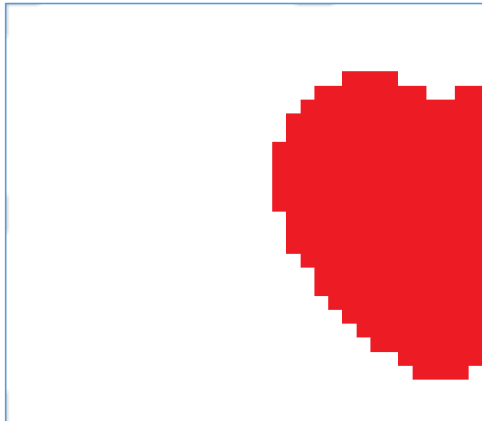
③ Seam Carving
(중요한 부분 최대한 보존)



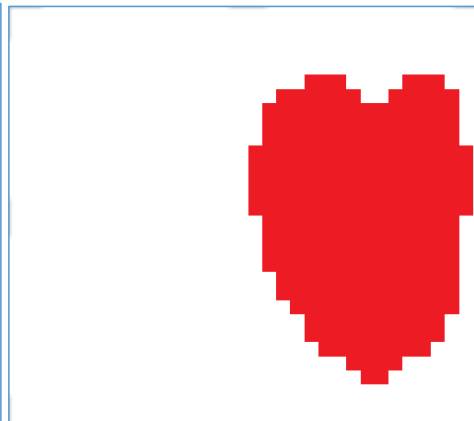
<원본>



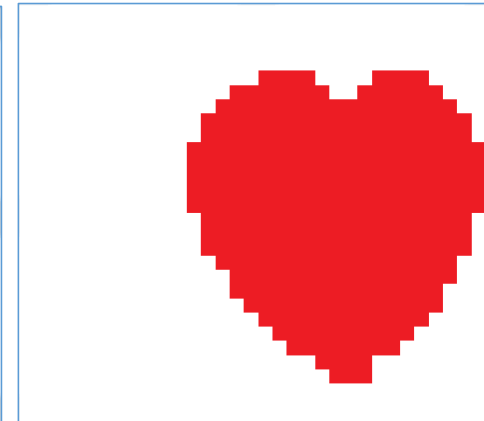
<오른쪽 10개 pixel 만큼 slide해서 창 크기 줄일 때>



① 오른쪽 crop



② 같은 비율로 줄임
(임의 10개 수직선 삭제)



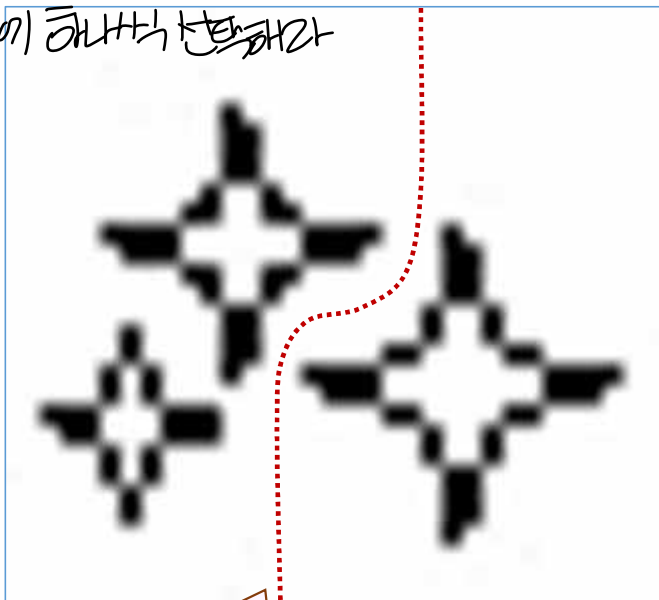
③ Seam Carving
(중요한 부분 최대한 보존)

주요 콘텐츠 아닌 부분 위주로 (예: 비어 있는 배경) 삭제함으로써 주요 콘텐츠 우선적으로 보존. Content-aware 압축으로 볼 수도 있음

양옆 밀어 1 pixel 축소:

너비 1 pixel인 위아래 연결선 중 중요도 합 가장 낮은 선 (seam) 찾아 제거 (carve)

→ 한쪽이 하나씩 변형해라

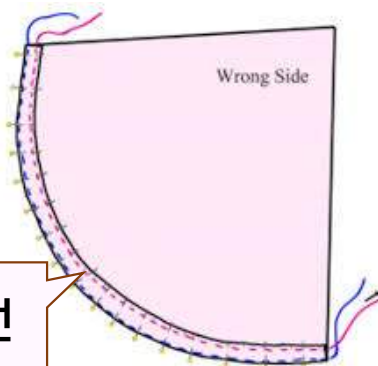


연결선에 속한 pixel의
중요도 합



연결되었다면
곡선도 가능

seam: 꺾맨 선, 재봉 선

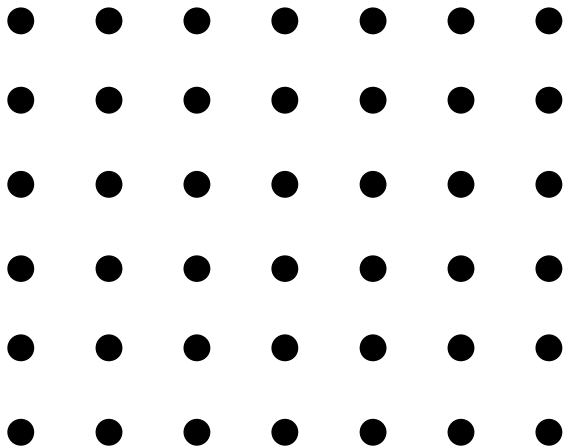




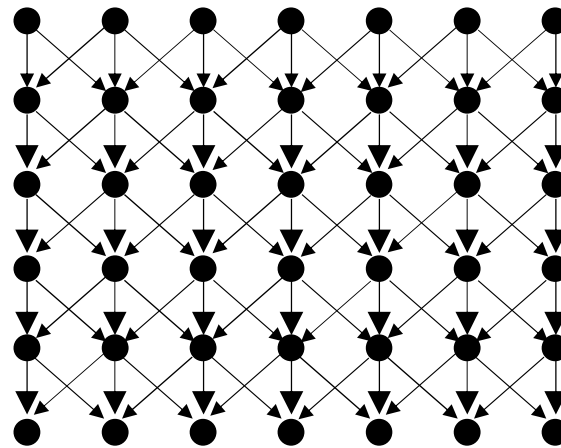
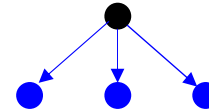
Seam(중요도 합 가장 낮은 선) 찾는 방법

66

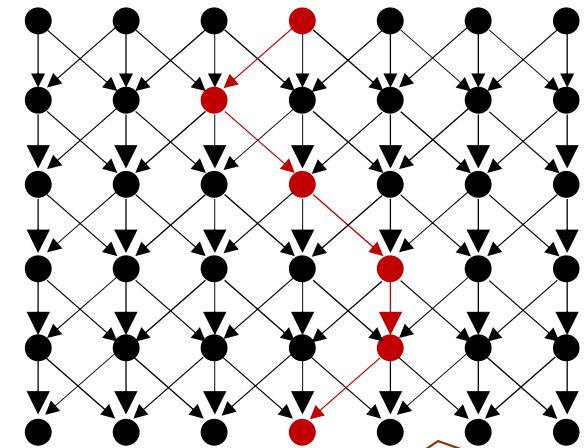
① 각 픽셀을 정점으로 봄



② 각 픽셀이 아래 3개 정점으로 연결되었다고 봄



③ 위아래 연결선 중
중요도 합 최소인 연결선
(seam) 찾기



위→아래로 끊어짐 없이 연결된
경로 찾기 위함

각 행마다 하나의 열 선정
 r 번째 행에 선정된 열이 c 라면
 $r+1$ 번째 행에는 $c-1, c, c+1$
중 하나의 값 선정됨



픽셀의 중요도(energy): 픽셀 주변 색깔 변화 클수록 중요도 커짐



반대로 색깔 변화 거의 없는
곳이 중요도 가장 낮음



[Q] 위 두 그림에서 중요도 가장 낮은 곳(주변의 색깔 변화 가장 작은 곳)은 어디일까?
Seam carving은 중요도 합이 가장 작은 위아래 연결선을 제거함을 기억 하시오.

Background (흰배경)

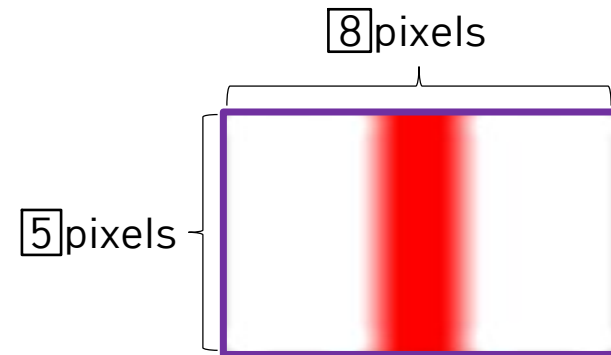


픽셀의 중요도(energy): 픽셀 주변 색깔 변화 클수록 중요도 커짐

- 픽셀 별로 중요도 배정 (정점에 비중). 높을수록 더 중요
- 픽셀 주변 색깔 변화 정도에 따라 $0 \leq \text{energy} < 1000$ 범위 숫자 배정
- 경계선 상 픽셀: (외부와 경계라 중요한 부분으로 보고) 최대치 1000 배정

픽셀 번호가 아래로 증가.

y/x	0	1	2	3	4	5	6	7
0	1000	1000	1000	1000	1000	1000	1000	1000
1	1000	0	0	360	360	0	0	1000
2	1000	0	0	360	360	0	0	1000
3	1000	0	0	360	360	0	0	1000
4	1000	1000	1000	1000	1000	1000	1000	1000



한 픽셀 작을까.

이미지의 픽셀은 보통 (x,y) 순으로 indexing 하고
 왼쪽 위 index가 (0,0)이고,
 오른쪽 아래가 (w-1, h-1) 임에 유의
 w: 그림의 너비(가로 픽셀 수)
 h: 높이(세로 픽셀 수)



(RGB 형식) 각 픽셀의 색은 0~255 범위 정수 3개로 나타냄

(255,101,51)	(255,101,153)	(255,101,255)
(255,153,51)	(255,153,153)	(255,153,255)
(255,203,51)	(255,204,153)	(255,205,255)
(255,255,51)	(255,255,153)	(255,255,255)

12개 pixel의 RGB 값

- (R, G, B)
- R: Red
- G: Green
- B: Blue
- 위 3개 색의 조합으로 색깔을 나타냄



픽셀의 중요도(energy): 픽셀 주변 색깔 변화 클수록 중요도 커짐

- 경계선 상 픽셀: (외부와 경계라 중요한 부분으로 보고) 최대치 1000 배정
- 그 외 픽셀 $(x,y): \sqrt{\Delta(x-1, y, x+1, y)^2 + \Delta(x, y-1, x, y+1)^2}$ (x,y) 주변 4개 픽셀의 색깔 차이
- $\Delta(x-1, y, x+1, y)^2 =$ (x,y) 좌우 2개 픽셀 색깔 차이
 $\{R(x-1, y) - R(x+1, y)\}^2 + \{G(x-1, y) - G(x+1, y)\}^2 + \{B(x-1, y) - B(x+1, y)\}^2$
- $\Delta(x, y-1, x, y+1)^2 =$ (x,y) 상하 2개 픽셀 색깔 차이
 $\{R(x, y-1) - R(x, y+1)\}^2 + \{G(x, y-1) - G(x, y+1)\}^2 + \{B(x, y-1) - B(x, y+1)\}^2$

(255,101,51)	(255,101,153)	(255,101,255)
(255,153,51)	(255,153,153)	(255,153,255)
(255,203,51)	(255,204,153)	(255,205,255)
(255,255,51)	(255,255,153)	(255,255,255)

12개 pixel의 RGB 값

1000	1000	1000
1000	$\sqrt{52225}$	1000
1000	$\sqrt{52024}$	1000
1000	1000	1000



새로운 접근법 → 양방향 → Digraph 쓸 수도
 사이클 X → Asyclic 쓸 수도

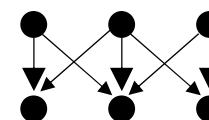
중요도 낮음

[Q] Pixel의 중요도가 아래와 같을 때 위아래 연결선 중 **seam**을 찾으시오.

1000	1000	1000	1000	1000	1000
1000	237	151	234	107	1000
1000	138	228	133	211	1000
1000	153	174	284	194	1000
1000	1000	1000	1000	1000	1000

1000	1000	1000	1000	1000	1000
1000	1	3	0	0	1000
1000	2	0	5	4	1000
1000	7	3	0	0	1000
1000	1000	1000	1000	1000	1000

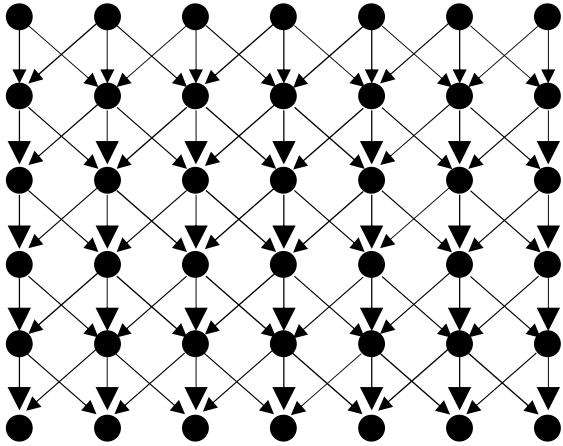
1000	1000	1000	1000	1000	1000	1000
1000	0	7	5	1	2	1000
1000	3	0	10	9	3	1000
1000	21	21	100	1	1	1000
1000	45	8	4	0	2	1000
1000	90	8	13	2	7	1000
1000	1000	1000	1000	1000	1000	1000



- ① 각 행마다 하나의 열 선정
- ② 위 → 아래로 한 pixel 내려갈 때 좌우로 최대 1 pixel 만큼만 이동 가능
 즉 r번째 행에 선정된 열이 c라면 r+1번째 행에는 c-1, c, c+1 중 하나의 값 선정



Pixel의 중요도 얻었다면 위아래 연결선 중 **seam(중요도 합 가장 낮은 선)** 어떻게 찾나? 3가지 알고리즘 중 무엇에 기반해야 할까?

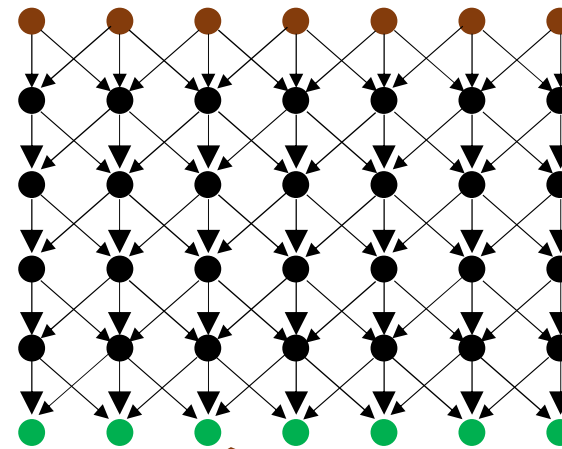


[Q] Topological Order 구해야 하나?

Seam(중요도 합 가장 낮은 선) 찾기: 유의사항 및 기본 가정

y/x	0	1	2	3	4	5
0	1000	1000	1000	1000	1000	1000
1	1000	1	3	0	0	1000
2	1000	2	0	5	4	1000
3	1000	7	3	0	0	1000
4	1000	1000	1000	1000	1000	1000

이미지에서는
 왼쪽 위 index가 (0,0)이고,
 오른쪽 아래가 (w-1, h-1) 임에 유의
 w: 그림의 너비(가로 픽셀 수)
 h: 높이(세로 픽셀 수)



가장 위 행에서 ($y=0$)
 가장 아래 행 ($y=h-1$) 각 정점으로 가는
 최단 경로(energy 합 최소인 경로)
 구한 후
 이 중 최단 경로를 seam으로 선택할 것임

- Energy(중요도) 합 최소인 seam 여럿이라면
- 이 중 어느 것이든 report 하면 됨

↓ 순서로 (topological order) (x,y)까지 최단경로 구해가기

<너비 w=6, 높이 h=5인 이미지에서
각 pixel의 중요도(energy)>

y/x	0	1	2	3	4	5
0	1000	1000	1000	1000	1000	1000
1	1000	1	3	0	0	1000
2	1000	2	0	5	4	1000
3	1000	7	3	0	0	1000
4	1000	1000	1000	1000	1000	1000

<distTo[]: 가장 위 행에서
x, y까지 최단경로 거리
(energy 합, (x,y)의 energy도 포함)>

y/x	0	1	2	3	4	5
0	1000	1000	1000	1000	1000	1000
1	2000	1001	1003	1000	1000	2000
2	2001	1002	1006	1005	1004	2000
3	2003	1001	1003	1000	1004	2004
4	2001	2003	2000	2000	2000	2004

<edgeTo[]: 가장 위 행에서
x, y까지 최단경로에서 마지막에 거쳐온 열>

y/x	0	1	2	3	4	5
0	None	None	None	None	None	None
1	↓ ₀	↓	↓	↓	↓	↓ ₅
2	←	↓	←	↓	←	→
3	←	2	2	2	4	4
4	←	2	3	3	3	4

(x-1,y-1), (x,y-1), (x+1,y-1) 중
energy 합 최소인 경우 여럿이라면
x → x-1 → x+1 순으로 선정한다고 가정



<너비 w=6, 높이 h=5인 이미지에서
각 pixel의 중요도(energy)>

1000	1000	1000	1000	1000	1000	1000
1000	0	7	5	1	2	1000
1000	3	0	10	9	3	1000
1000	21	21	100	1	1	1000
1000	45	8	4	0	2	1000
1000	90	8	13	2	7	1000
1000	1000	1000	1000	1000	1000	1000

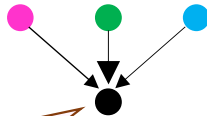
<distTo[]: 가장 위 행에서
x, y까지 최단경로 거리
(energy 합, (x,y)의 energy도 포함)>

1000	1000	1000	1000	1000	1000	1000
2000	1000	1000	1005	1001	1002	2000
2000	1003	1000	1011	1010	1004	2002

<edgeTo[]: 가장 위 행에서
x, y까지 최단경로에서 마지막에 거쳐온 열>

None	None	None	None	None	None	None

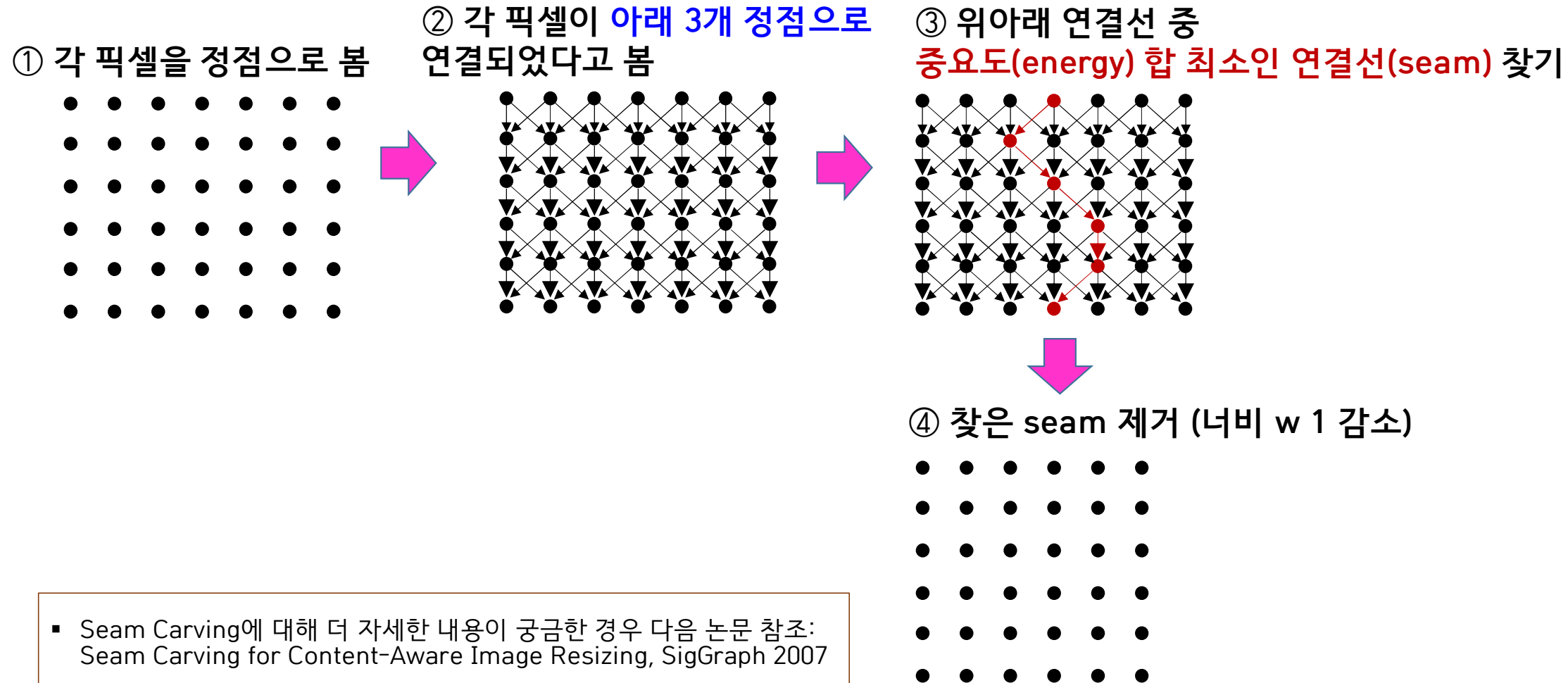
(x-1,y-1), (x,y-1), (x+1,y-1) 중
energy 합 최소인 경우 여럿이라면
x → x-1 → x+1 순으로 선정한다고 가정





Seam Carving 단계 정리

76



- Seam Carving에 대해 더 자세한 내용이 궁금한 경우 다음 논문 참조:
Seam Carving for Content-Aware Image Resizing, SigGraph 2007



Shorted Paths on Weighted Digraphs

최단경로 탐색 방법과 활용도 이해

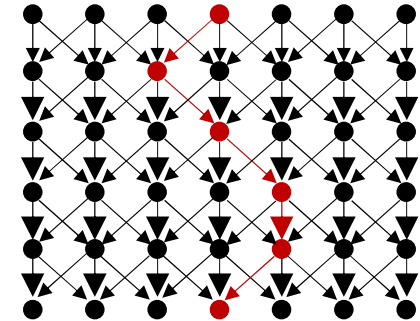
01. 퀴즈 풀이 & 예습 내용 복습 (이번 주 #1~3차 답안 공개)
02. 최단경로 문제의 기본 세팅 + 최단경로 탐색 방법의 공통점
03. Bellman-Ford 알고리즘
04. Dijkstra 알고리즘
05. Acyclic Shortest Path
06. Seam Carving
07. 실습: Seam Carving 구현

프로그램 구현 조건

- 위→아래 방향(vertical) seam 찾는 함수 구현


```
def findVerticalSeam(self):
```
- 입력 **self**: **SeamCarver** 클래스 객체로, 위 함수는 이 클래스의 멤버 함수임
 - 이 클래스에는 주요 멤버변수로 `self.image` 있으며
 - `Image` 클래스 객체로 seam carving할 이미지를 나타냄 (이어지는 페이지에서 더 자세히 설명)
- 반환 값
 - seam에 속하는 x값 저장하는 리스트
 - $y=0 \rightarrow y=h-1$ (h : 이미지의 높이) 순으로 seam에 선정된 x값 저장
 - 오른쪽 예제의 경우 `[3, 2, 3, 4, 4, 3]`

결과를 print하지 말고
반환하세요.



프로그램 구현 조건

- 최종 결과물로 SeamCarver.py 파일 하나만 제출하며, 이 파일만으로 코드가 동작해야 함
- import는 원래 SeamCarver.py 파일에서 하던 패키지 외에는 추가로 할 수 없음 (Path, Image, math, random, timeit)
- SeamCarver.py 내에 이미 구현되어 있던 코드는 제거하거나 수정하지 말 것
- 단 __main__ 아래의 코드는 테스트 위해 변경/추가해도 괜찮음
- 각자 테스트에 사용하는 모든 코드는 반드시 if __name__ == "__main__": 아래에 넣어
- 제출한 파일을 import 했을 때는 실행되지 않도록 할 것

구현된 API 정리 SeamCarver Class

seam carving 수행하고 결과 저장하는 클래스

```
# 이 클래스는 실습 과제로 구현할 findVerticalSeam() 외의 기능은 모두 구현된 클래스로
# 각 함수의 의미 이해하고 사용하기
class SeamCarver:
    # 멤버 변수, 상수
    self.image: Image class 객체로 seam carve하는 이미지 나타냄. Seam carving할 때마다 변경된 이미지 저장
    self.MAX_ENERGY: Energy(중요도)의 최대값 1000을 나타냄

    # 멤버 함수
    def __init__(self, image): # SeamCarver 생성자
        # image는 그림을 나타내는 Image class 객체로
        # 복사본을 멤버 변수 self.image에 저장 (seam carving 후에도 원본 이미지 보존하기 위함)

    def width(self): # self.image의 너비(가로 픽셀 수) 반환
    def height(self): # self.image의 높이(세로 픽셀 수) 반환

    def energy(self, x, y): # self.image에서 픽셀 (x,y)의 energy(중요도) 반환
        # **는 거듭제곱의 의미 (예: 4**2 == 16)
        # math.sqrt(x) 함수는 x의 제곱근을 구함
```


구현된 API 정리 SeamCarver Class

findVerticalSeam() 함수로 찾은 seam을 입력으로 받는 함수

```
# 이 클래스는 실습 과제로 구현할 findVerticalSeam() 외의 기능은 모두 구현된 클래스로
# 각 함수의 의미 이해하고 사용하기
class SeamCarver:
    # 멤버 함수
    def removeVerticalSeam(self, seam):
        # seam: findVerticalSeam() 함수로 찾은 vertical seam (위아래 방향으로 energy 합 최소인 경로)
        # self.image에서 seam을 제거 (그 결과 self.image에 저장된 이미지의 너비 1 감소)

    def isValidSeam(self, seam):
        # seam의 형식이 올바른지 검증하는 함수로
        # removeVerticalSeam() 등 seam을 입력으로 받는 함수 내부에서 입력 검증에 활용함
        # seam이 길이가 self.image의 높이와 같으며, 좌우로 1 pixel씩만 이동함 등을 검증

    def energySumOverVerticalSeam(self, seam):
        # seam이 나타내는 경로의 energy 합 구해 반환
```

구현된 API 정리 SeamCarver Class

debugging에 활용할 수 있는 함수 (Text Debugging)

이 클래스는 실습 과제로 구현할 findVerticalSeam() 외의 기능은 모두 구현된 클래스로
각 함수의 의미 이해하고 사용하기

```
class SeamCarver:
```

```
    # 멤버 함수
```

```
    def energyMap(self):
```

```
        # self.image 각 픽셀의 에너지를 문자열 형태로 반환
```

```
    def energyMapWithVerticalSeam(self, seam):
```

```
        # self.image 각 픽셀의 에너지를 문자열 형태로 반환하되
```

```
        # seam으로 선택된 픽셀은 에너지 값 뒤에 '*'를 붙여 반환
```

20×20 보다 큰 이미지는 이 방식으로 확인하기 불편합니다.

<10×10 이미지에 대한 energyMap() 출력 예>

```
1000 1000 1000 1000 1000 1000 1000 1000 1000 1000
1000    0    0 360 360 360 360    0    0 1000
1000    0    0 360 360 360 360    0    0 1000
1000    0    0 360 360 360 360    0    0 1000
1000    0    0 360 360 360 360    0    0 1000
1000    0    0 360 360 360 360    0    0 1000
1000    0    0 360 360 360 360    0    0 1000
1000    0    0 360 360 360 360    0    0 1000
1000    0    0 360 360 360 360    0    0 1000
1000    0    0 360 360 360 360    0    0 1000
1000 1000 1000 1000 1000 1000 1000 1000 1000 1000
```

<같은 이미지에 대한 energyMapWithVerticalSeam() 출력 예>

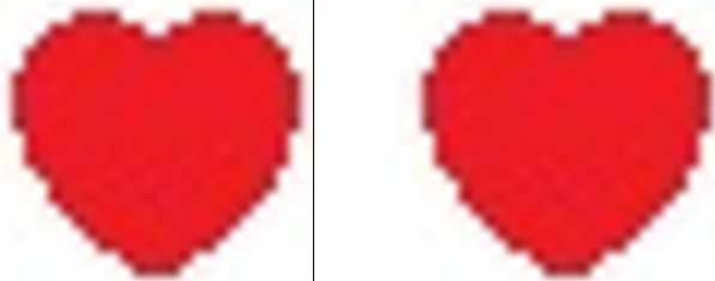
```
1000 1000* 1000 1000 1000 1000 1000 1000 1000 1000
1000    0*    0 360 360 360 360    0    0 1000
1000    0*    0 360 360 360 360    0    0 1000
1000    0*    0 360 360 360 360    0    0 1000
1000    0*    0 360 360 360 360    0    0 1000
1000    0*    0 360 360 360 360    0    0 1000
1000    0*    0 360 360 360 360    0    0 1000
1000    0*    0 360 360 360 360    0    0 1000
1000    0*    0 360 360 360 360    0    0 1000
1000    0*    0 360 360 360 360    0    0 1000
1000    0*    0 360 360 360 360    0    0 1000
1000* 1000 1000 1000 1000 1000 1000 1000 1000 1000
energy sum over vertical seam: 2000
```

구현된 API 정리

debugging에 활용할 수 있는 함수 (Graphical Debugging)

```
# 아래 함수는 SeamCarver 클래스 외부에 있는 함수로,
# SeamCarver 클래스 객체를 생성해 seam carving을 수행함
#
def showBeforeAfterSeamCarving(fileName, numCarve):
    # fileName: seam carving을 수행할 이미지 파일 이름으로 (jpg, png 등 가능)
    #           SeamCarver.py와 같은 디렉토리에 있는 파일이어야 함
    # numCarve: carving을 수행할 횟수
    #
    # fileName이 지정한 그림에 대해 numCarve만큼 seam carving을 수행해 결과를 좌우로 대비해 보여줌
```

showBeforeAfterSeamCarving("heartR.jpg", 10)



showBeforeAfterSeamCarving("stars.jpg", 10)



100×100 보다 큰 이미지는 Python 속도 때문에 처리에 시간이 걸리니 첨부된 4개 jpg 파일로 테스트 해보세요.

구현된 API 정리

이미지의 픽셀 접근 방법 알고 싶을 때 참고할 함수

```
# 아래 함수는 SeamCarver 클래스 외부에 있는 함수로,
# 주어진 컬러 이미지를 흑백(gray scale) 이미지로 변경해 반환함
# SeamCarver 클래스 내 energy() 함수의 동작 이해하고 싶다면 이 함수 참고
#
def convertToGrayscale(image) :
    # image: Image 클래스 객체
    # image.load(): pixel 나타내는 2차원 배열 pixels 반환
    # pixels[x,y]: 픽셀 (x,y)의 (r,g,b) 값을 3-tuple로 반환
```

```
# 위 함수 활용 예
image_color = Image.open(Path(__file__).with_name("heart.jpg")) # heart.jpg 파일 읽어 Image 객체 생성
image_gray = convertToGrayscale(image_color) # 이미지를 흑백으로 변환해 image_gray에 저장
image_color.show() # 변환 전 이미지 보이기
image_gray.show() # 변환 후 이미지 보이기
```



구현할 API 정리: findVerticalSeam()

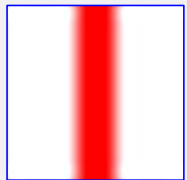
```
# 구현해야 할 함수. SeamCarver 클래스의 멤버 함수로, 멤버 변수 self.image에 저장된 이미지의 seam 구해 반환
def findVerticalSeam(self)
    # distTo[] 초기화: 첫 번째 행으로 (y=0) 이미지 너비 만큼 self.MAX_ENERGY 담은 리스트 추가
    # edgeTo[] 초기화: 첫 번째 행으로 (y=0) 이미지 너비 만큼 None 담은 리스트 추가
    #
    # distTo[], edgeTo[]를 마지막 행 y=h-1 까지 채우기 위해 아래 수행
    # for y = 1 ~ (h-1)에 대해 아래 반복:
    #     distTo[]에 새로운 행 추가 (이미지 너비 만큼 0 담은 리스트)
    #     edgeTo[]에 새로운 행 추가 (이미지 너비 만큼 0 담은 리스트)
    #     for x = 0 ~ (w-1)에 대해 아래 반복:
    #         distTo[y-1][x], distTo[y-1][x-1], distTo[y-1][x+1] 중 최솟값 찾기
    #         distTo[y][x] = 앞에서 찾은 최솟값 + (x,y)의 energy
    #         edgeTo[y][x] = 앞에서 찾은 최솟값의 x좌표
    #
    # 결과 리스트 (seam) 만들기 위해 아래 수행
    # 결과 리스트를 [0] * self.height()으로 초기화
    # 마지막 행 distTo[h-1]에 저장된 값 중 최솟값 찾기
    # 앞에서 찾은 최솟값의 x 좌표가 x_min 이라면
    # for y = (h-1) ~ 0에 대해 아래 반복:
    #     결과 리스트[row]에 x_min 기록
    #     x_min = edgeTo[y][x_min]
    #
    # 결과 리스트 반환
```

프로그램 입출력 예 (SeamCarver.py 파일 __main__ 아래에 있음)

```
image = Image.new("RGB", (10,10), "white") # 10×10 이미지 만들면서 픽셀 모두 흰색으로 초기화
pixels = image.load() # 이미지의 픽셀에 접근하기 위해 load() 호출해 픽셀 객체 얻음
for row in range(image.size[0]): # x=4~5 두 열에 속한 픽셀 모두를 붉은색으로 변경
    pixels[4,row] = (255,0,0)
    pixels[5,row] = (255,0,0)
sc = SeamCarver(image) # 앞에서 만든 이미지로 SeamCarver 클래스 객체 생성
vs = sc.findVerticalSeam() # seam 찾아서 vs에 저장
print(sc.energyMapWithVerticalSeam(vs), '\n') # 픽셀의 energy를 seam과 함께 출력
```

```
1000 1000* 1000 1000 1000 1000 1000 1000 1000 1000
1000 0* 0 360 360 360 360 0 0 1000
1000 0* 0 360 360 360 360 0 0 1000
1000 0* 0 360 360 360 360 0 0 1000
1000 0* 0 360 360 360 360 0 0 1000
1000 0* 0 360 360 360 360 0 0 1000
1000 0* 0 360 360 360 360 0 0 1000
1000 0* 0 360 360 360 360 0 0 1000
1000 0* 0 360 360 360 360 0 0 1000
1000* 1000 1000 1000 1000 1000 1000 1000 1000 1000
energy sum over vertical seam: 2000
```

Energy 합이 최소(2000)인
다른 경로가 출력되어도 괜찮음



프로그램 입출력 예 (SeamCarver.py 파일 __main__ 아래에 있음)

```

image = Image.new("RGB", (10,10), "white") # 10×10 이미지 만들면서 픽셀 모두 흰색으로 초기화
pixels = image.load() # 이미지의 픽셀에 접근하기 위해 load() 호출해 픽셀 객체 얻음
for row in range(image.size[0]): # x=4~5 두 열에 속한 픽셀 모두를 붉은색으로 변경
    pixels[4,row] = (255,0,0)
    pixels[5,row] = (255,0,0)
sc = SeamCarver(image) # 앞에서 만든 이미지로 SeamCarver 클래스 객체 생성
vs = sc.findVerticalSeam() # seam 찾아서 vs에 저장
print(sc.energySumOverVerticalSeam(vs)) # seam에 속한 픽셀의 energy 합 출력
sc.removeVerticalSeam(vs) # seam을 제거
print(sc.width()) # seam 제거 후 이미지의 너비 출력

```

2000

9

프로그램 입출력 예 (SeamCarver.py 파일 __main__ 아래에 있음)

```
image2 = Image.new("RGB", (3,10), "white") # 3×10 이미지 만들면서 픽셀 모두 흰색으로 초기화
sc2 = SeamCarver(image2) # 앞에서 만든 이미지로 SeamCarver 클래스 객체 생성
vs2 = sc2.findVerticalSeam() # seam 찾아서 vs에 저장
print(sc2.energyMapWithVerticalSeam(vs2)) # 픽셀의 energy를 seam과 함께 출력
```

```
1000 1000* 1000
1000  0* 1000
1000  0* 1000
1000  0* 1000
1000  0* 1000
1000  0* 1000
1000  0* 1000
1000  0* 1000
1000  0* 1000
1000* 1000 1000
```

Energy 합이 최소(2000)인
다른 경로가 출력되어도 괜찮음

```
energy sum over vertical seam: 2000
```




유의사항 정리

- `__main__` 아래 더 많은 테스트 케이스가 있습니다.
- Energy 합 최소인 경로 둘 이상이라면 그 중 하나만 반환하면 됨
- 이 문제에서는 간선과 그래프를 나타내는 객체는 별도로 안 만들어도 됩니다.
- 픽셀 간 연결상태 및 topological order를 이미 알고 있으므로
- 이에 따라 이미지 객체에 직접 탐색을 수행합니다.

프로그램 구현 조건 - 성능

- 앞에서 배운 방법에 따르면 vertical seam을 찾는 시간은 이미지의 픽셀 개수 $w \times h$ 에 비례합니다.
- Vertical seam을 찾는 과정은 `distTo[]`와 `edgeTo[]`에 값을 채워 넣는 과정인데
- 이들의 크기가 $w \times h$ 이며
- 한 값을 채워 넣는데 상수 시간이 걸리기 때문입니다. (3 값 중 최솟값 찾기 & 값 저장 등)
- 실행 시간은 `__main__` 아래 speed test를 통과하면 됩니다.
- 단 정확도 테스트를 모두 통과하지 않았다면 속도 테스트는 fail한 것으로 봅니다. (답이 올바르지 않다면 속도는 큰 의미 없음)

```
image3 = Image.open(Path(__file__).with_name("piplub.jpg")) # Use the location of the current .py file
sc3 = SeamCarver(image3)
n=20
tVerticalSeam = timeit.timeit(lambda: sc3.findVerticalSeam(), number=n)/n
tGrayScale = timeit.timeit(lambda: convertToGrayScale(image3), number=n)/n
print(f"Finding {n} vertical seams on a 100x100 image took {tVerticalSeam:.10f} sec on average")
print(f"Creating {n} gray scale images on a 100x100 image took {tGrayScale:.10f} sec on average")
if (tVerticalSeam < 12 * tGrayScale): print("pass for speed test")
else: print("fail for speed test")
```

```
Finding 20 vertical seams on a 100x100 image took 0.0440674250 sec on average
Creating 20 gray scale images on a 100x100 image took 0.0055430250 sec on average
pass for speed test
```



기말고사

- 화요일 분반
 - 날짜, 시간: **12월 13일(화)**
09:00~12:00 (수업 시간)
 - 장소: IT5-309 (수업 장소)
- 수요일 분반
 - 날짜, 시간: **12월 14일(수)**
09:00~12:00 (수업 시간)
 - 장소: IT5-309 (수업 장소)