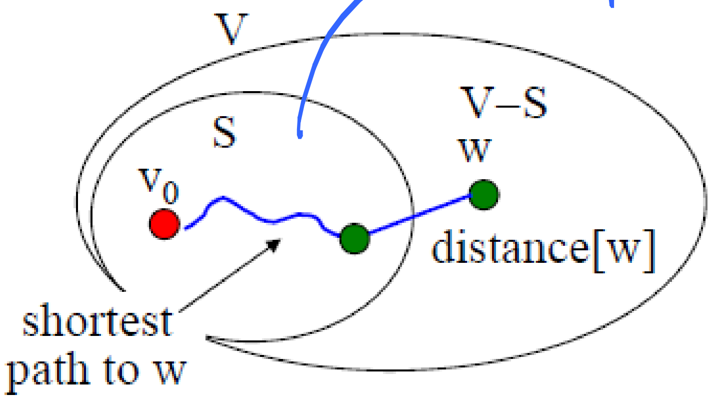
◆ Dijkstra's Shortest Path Algorithm

- ✦ Single source all destinations shortest path problem.
- ✦ Find the shortest path from the source v0 to every other vertices. Let length[v] be the shortest path length.
- ✦ Let S denote the set of vertices, including v0, whose shortest path has been found.
- ✦ Let distance[w] be the length of the shortest path starting from v0, going through vertices only in S, and ending in w.

shortes path의 길이를 찾은 vertex set.



Dijkstra's Shortest Path Algorithm

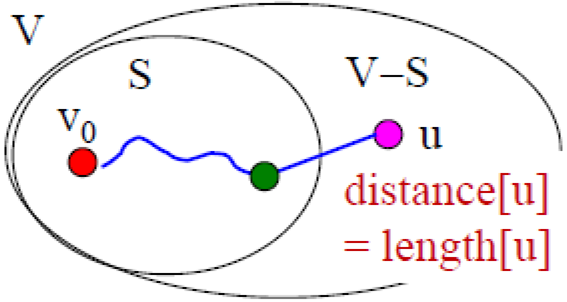
Initialization

```
S = {v0}, distance[v0] = 0
for w ∈ V – S {
  if ( (v0, w) ∈ E ) distance[w] = cost(v0, w)
  else distance[w] = ∞
}
```

distance[u] 가 length[u] 가 된다.
즉, shortest path from s to u length

Algorithm

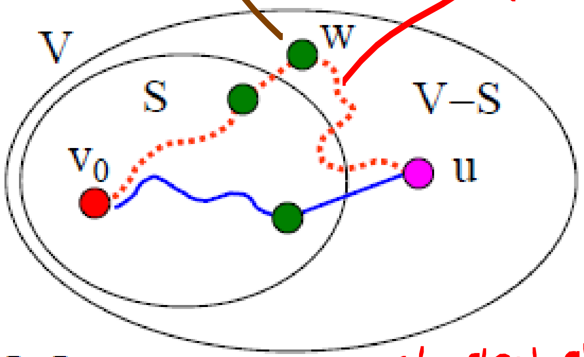
```
1. S = {v0}
2. while V – S != empty set {
3.    u = min{distance[w], w ∈ V – S}
4.    S = S ∪ {u}, V – S = (V – S) – {u}   // u는 S에 포함. V-S에서는 제거.
5.    for every vertex w in V – S {
6.       update distance[w]
7.    }
8. }
```



- At line 3, length[u] = distance[u]

✦ A proof that length[u] = distance[u] at Line 3. distance[w] ?

P



1. Suppose that distance[u] > length[u].
2. Let P be a shortest path from v0 to u.
3. The length of P = length[u].
4. P has at least one vertex $\notin$ S.

   Otherwise, the length of P = distance[u].

s shortest 이면 x

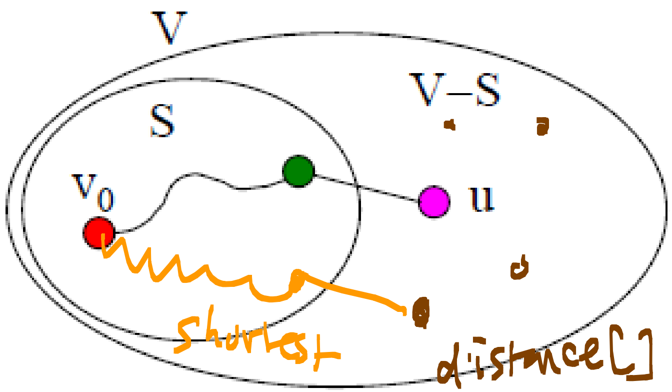
shortest 이다.

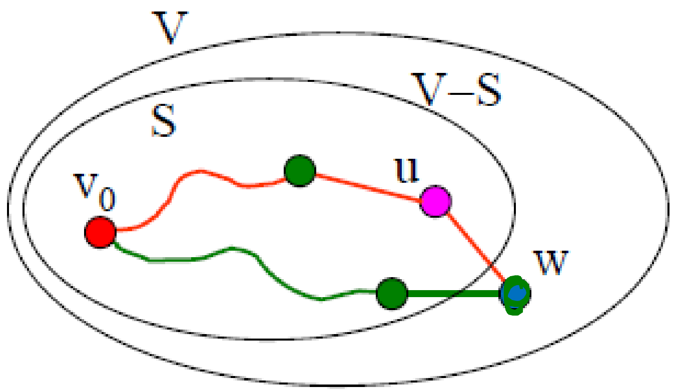
5. Let w ($\in$ P) be the vertex which is closest to v0.
6. distance[u] > length[u] $\geq$ length[w] (= distance[w])
7. This implies that distance[u] > distance[w] since length[w] = distance[w].

   why?
8. This is a contradiction since the Dijkstra's algorithm selects a vertex whose distance[ ] is minimum in V – S.
9. Therefore, distance[u] = length[u].

✦ Distance update at Line 6

```
for w ∈ V – S {
   distance[w] =                          length[u]
                                          //           w
            min{mistance[w], distance[u]+cost(u,v)}
                d
}
```





✦ After completion of the algorithm, distance[v]=length[v] for all $v \in V$.

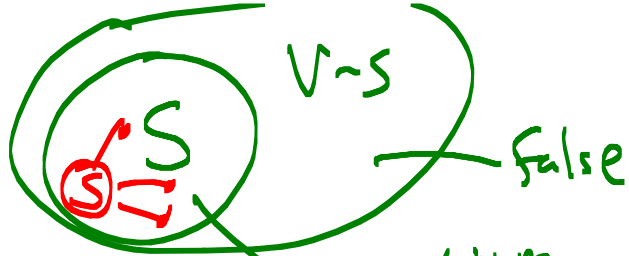
distance[] array 채우면 충분.

✦ An Algorithm with an Adjacency List

```
for( i = 0; i < n; i++ ) {
  found[i]=F; distance[i] = +∞;
}
found[s]=T; distance[s]=0;
for( every vertex u adjacent to s )
  distance[u] = cost(s,u);
for( i=0; i<n-1; i++ ) { // n-1 iterations
  Choose a vertex u such that found[u]=F and
    distance[u] is minimum;   // O(n)  n번 반복
  found[u] = T;   // u∈S
  for( every vertex w adjacent to u ) // O(|E|)
    if( found[w]==F &&                //   total
        distance[u]+cost(u,w) < distance[w] )
      distance[w]=distance[u]+cost(u,w);
}
```

if(u==t) break

Initialization



여기에 속하면 found[S]=true

포함시

각 vertex 마다 ~~ed~~ adj edge는 한번씩 검색. total |E|번 검색

$$\underbrace{n(n-1)}_{n^2} + \underbrace{|E|}_{O(n^2)} = O(n^2) \quad |V|^2$$

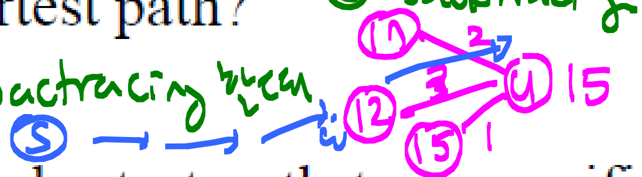
✦ $O(|V|^2)$ time.

◆ Remarks

① 추가 array 사용해서 distance[] update 할 때 u정보 parent[w]=u로 setting.

② backtracing

- Finding an actual shortest path?
- If the cost is real? → backtracing 할 때



if(distance[u] - cost(u,w) == distance[w]) {

실수인 경우 연산 때 오차가 발생할 수 있다. (절대값이 아주 작으면)

- If we want to find a shortest path to a specific target vertex, how can we modify the algorithm?
- Shortest path tree : The spanning tree which has the shortest path from the source vertex to every other vertex. We can easily create such a tree.

- May not work when some edges have negative cost.
- Time Complexity : Better implementation is possible.
  - O((|V|+|E|)log|V|) algorithm(see the next page).
  - O(|V|log|V| + |E|) algorithm using the Fibonacci heap.

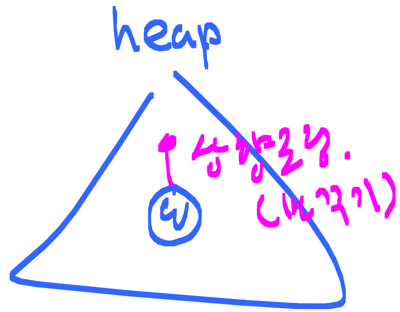
✦ An O((|V|+|E|) log|V|) Algorithm

```
for(i = 0; i < n; i++) {
   found[i] = F; distance[i] = -∞;}
found[s] = T;   distance[s] = 0;
for( every vertex u adjacent to s )
   distance[u] = cost(s,u);
construct min_heap(V-{s});
for(i=0; i < n-2; i++) { // n-1 iterations(=O(|V|))
   Choose a vertex u such that distance[u]
     is minimum; // O(1)
   found[u] = T;
   Delete u from the heap // O(log|V|)
   for( every vertex w adjacent to u ) // O(|E|)
     if ( found[w] == F &&                //   total
          distance[u] + cost(u,w) < distance[w] ) {
       distance[w] = distance[u] + cost(u,w);
       adjust heap(w);        // O(log|V|)
     }
}
```

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

Even if |E|<|V|, we still need n-1 iterations.

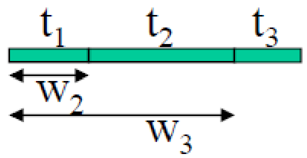
No removal of elements. Just adjust the cost.



w의 distance 값이 바뀌면 ⇒ heap이 아님. ⇒ heap으로 만들기

Scheduling Problems

- Minimizing Total Time in the System
  - Problem Definition
    - Input : A list of jobs J_1, J_2, …, J_n. Each job J_i has service time t_i and it waits w_i before start execution.
    - Question : Find a schedule that minimize $\Sigma(t_i+w_i)$.
  - An Example : Three jobs and service times $t_1=5$, $t_2=10$ and $t_3=4$.



Schedule	*Total Time in the System*
[1, 2, 3]	$5+(5+10)+(5+10+4)=39$
[1, 3, 2]	$5+(5+4)+(5+4+10)=33$
[2, 1, 3]	$10+(10+5)+(10+5+4)=44$
[2, 3, 1]	$10+(10+4)+(10+4+5)=43$
[3, 1, 2]	$4+(4+5)+(4+5+10)=32$
[3, 2, 1]	$4+(4+10)+(4+10+5)=37$

Optimal schedule

A Greedy Algorithm : Schedule jobs in nondecreasing order by service time.
Proof. See the text book.

✦ A Proof of Optimality

▶ Theorem 4.3

The only schedule that minimizes the total time in the system is one that schedules jobs in nondecreasing order by service time.

Proof: For $1 \le i \le \overset{n}{\cancel{n-1}}$, let t_i be the service time for the ith job scheduled in some particular optimal schedule (one that minimizes the total time in the system). We need to show that the schedule has the jobs scheduled in nondecreasing order by service time. We show this using proof by contradiction.

가정

$\left.\begin{matrix} J_1 & J_2 & ---- & J_n \\ t_1 & t_2 & & t_n \end{matrix}\right\}$ 주어진 optimal schedule 이라고 하자.

$\Rightarrow t_1 \le t_2 \le --- \le t_n$ 이다.

만약 아니라면,

A { If they are not scheduled in nondecreasing order, then for at least one i where $1 \le i \le n-1$, $\quad t_i > t_{i+1}$.

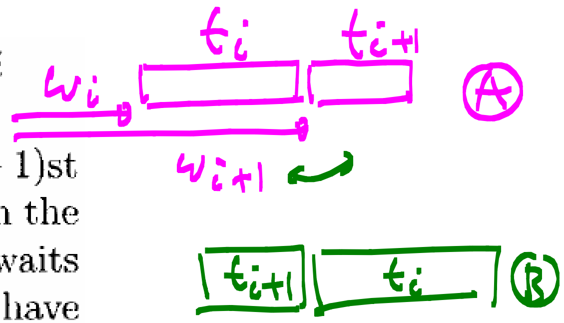
B { We can rearrange our original schedule by interchanging the ith and $(i+1)$st jobs. By doing this, we have taken t_i units off the time the $(i+1)$st job (in the original schedule) spends in the system. The reason is that it no longer waits while the ith job (in the original schedule) is being served.

C { Similarly, we have added t_{i+1} units to the time the ith job (in the original schedule) spends in the system. Clearly, we have not changed the time that any other job spends in the system. Therefore, if T is the total time in the system in our original schedule and T' is the total time in the rearranged schedule,

$$T' = T + t_{i+1} - t_i.$$

Because $t_i > t_{i+1}$, $T' < T$,

which contradicts the optimality of our original schedule.



J_{i+1} 의 수행 시간의 변화

A $\quad w_i + t_i + t_{i+1}$

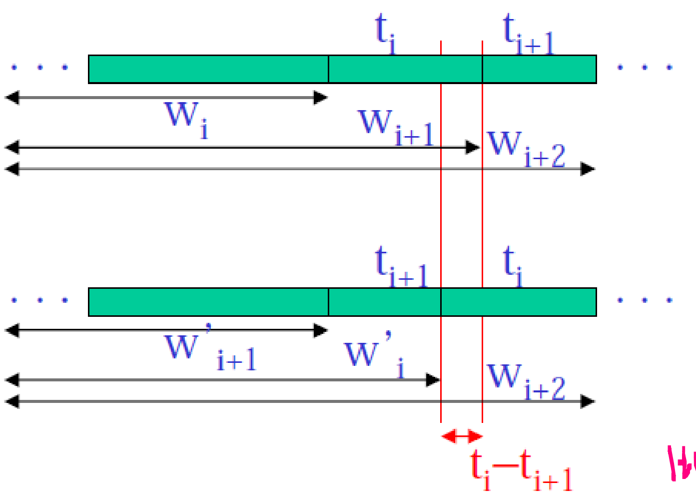
B $\quad w_i + t_{i+1}$

$(t_i \text{ off})$

J_i A $\quad w_i + t_i$

B $\quad w_i + t_{i+1} + t_i$

$(t_{i+1}$이 추가$)$



time for (i+1)st job:

$t_{i+1} + w_i$ (t_i off)

time for ith job:

$t_i + w_i + t_{i+1}$ (t_{i+1} more)

J_i, J_{i+1}

A $\quad 2w_i + 2t_i + t_{i+1}$

B $\quad 2w_i + 2t_{i+1} + t_i$

$w'_i + w'_{i+1} = w_i + w_{i+1} - t_i + t_{i+1}$

바뀐 cost가 더 작다.

A $-$ B $= t_i - t_{i+1} > 0$

◆ Scheduling with Deadlines

✦ Problem Definition

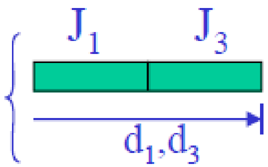
processing time은 모두 1이라고 가정

- ◆ Input : A list of jobs $J_1, J_2, \ldots, J_n$. Each J_i takes a unit time to finish and has a deadline d_i and a profit p_i.
- ◆ Question : Find a schedule that maximize the profit.
  - If a job J_i starts before or at d_i, profit p_i is obtained. Otherwise, no profit is obtained for J_i.

Job	*Deadline*	*Profit*
1	2	30
2	1	35
3	2	25
4	1	40

1	3

Schedule	*Total Profit*
[1, 3]	30 + 25 = 55
[2, 1]	35 + 30 = 65
[2, 3]	35 + 25 = 60
[3, 1]	25 + 30 = 55
[4, 1]	40 + 30 = 70
[4, 3]	40 + 25 = 65



Others can not be scheduled. So, no profits for other jobs.

Optimal sequence (optimal set={1,4})

Schedule jobs in decreasing order of profits. 이득이 큰것부터 schedule 하도

- J_4 is scheduled.
- J_2 is not possible to schedule.
- J_1 is scheduled.
- J_3 is not possible.

J_4	J_1

Feasible sequence (feasible set={3,4})

Schedule [1,2] is not possible. (not feasible sequence)

1	2

~ 2의 deadline =1 ×

✦ An Algorithm

```
sort the jobs in nonincreasing order by profit;
S = ∅;                                          이득의 내림차순으로 정렬
while (the instance is not solved){
  select next job; // selection procedure
  if (S is feasible with this job added)//feasibility
     add this job to S;                    //check
  if (there are no more jobs) // solution check
     the instance is solved;
}
```

feasibility check?

Job	*Deadline*	*Profit*
1	3	40
2	1	35
3	1	30
4	3	25
5	1	20
6	3	15
7	2	10

1. S is set to $\varnothing$.
2. S is set to $\{1\}$ because the sequence $[1]$ is feasible.
3. S is set to $\{1, 2\}$ because the sequence $[2, 1]$ is feasible. J3 포기
4. $\{1, 2, 3\}$ is rejected because there is no feasible sequence for this set.
5. S is set to $\{1, 2, 4\}$ because the sequence $[2, 1, 4]$ is feasible.
6. $\{1, 2, 4, 5\}$ is rejected because there is no feasible sequence for this set.
7. $\{1, 2, 4, 6\}$ is rejected because there is no feasible sequence for this set.
8. $\{1, 2, 4, 7\}$ is rejected because there is no feasible sequence for this set.

{1,2,4} is an optimal set.
[2,1,4] and [2,4,1] are optimal sequences. Just choose one.

✦ An Efficient Way to Determine Feasibility

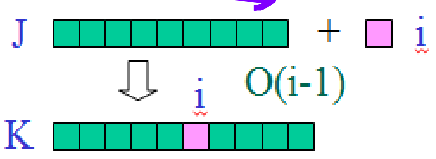
▲ Lemma 4.3 Let S be a set of jobs. Then S is feasible if and only if the sequence obtained by ordering the jobs in S according to nondecreasing deadlines is feasible.

S가 feasible ⟺ S의 원소를 deadline의 오름차순으로 정렬한 seq가 feasible하다.

✦ An Algorithm

```
sort the jobs by profit (nonincreasing); // O(nlogn)
void schedule (int n, const int deadline[],
                    sequence_of_integer& j) {
  index i;
  sequence_of_integer K;  // temp storage
  J = [1];
  for (i = 2; i <= n; i++){
①    K = J with i added according to
          nondecreasing values of deadline[i];
②    if (K is feasible) // O(i) (일일이 조사)
        J = K;
  }
}
```

deadline의 오름차순으로 정렬된 array



① deadline의 오름차순이 되도록 i를 insert

② feasible한지 check한다

to add ith job   to check feasibility

$$\sum_{i=2}^{n}[(i-1)+i] = n^2 - 1 \in \Theta(n^2)$$

Read Theorem4.4 (An optimality proof)

11/24