Shortest-paths problem (최단 경로 문제)

> Introduction to Algorithms 3rd edition 챕터 24를 정리한 내용

Preface

최단 경로 문제에서, 가중치가 있고 방향이 있는 그래프 $G = (V, E)$ 와 `edge` 를 실수의 가중치(weight)로 매핑하는 $w : E \rightarrow \mathbb{R}$ 함수가 주어진다. 경로 $p =< v_0, v_1, \ldots, v_k >$ 의 가중치 $w(p)$ 는 구성하는 `edges`의 합이 된다.

$$\sum_{i=1}^{k} w(v_{i-1}, v_i)$$

이때 u 부터 v 까지의 최단 경로(shortest-path weight) $\delta(u, v)$ 는 다음과 같이 정의된다.

$$\delta(u,v) = \begin{cases} min\{w(p) : u \rightsquigarrow^p v\}, & \text{if there is a path from u to v.} \\ \infty, & \text{otherwise.} \end{cases}$$

이때 u 부터 v 까지의 경로 p 중 $w(p) = \delta(u, v)$ 를 만족하는 p가 된다.

edge의 가중치는 거리 뿐만 아니라 소요시간, 비용, 패널티, 손실 혹은 알고리즘에서 최소화 하기 위한 어떤 값이든 될 수 있다.

Variants

이 챕터에선 single-source shortest-paths problem, 즉 주어진 그래프 $G = (V, E)$ 에서 시작점 $s \in V$ 부터 각 점 $v \in V$ 까지 최단 경로를 찾는 문제에 집중해서 알아보자. 이 single-source problem을 위한 알고리즘은 다음과 같은 varients를 포함하는 여러 문제를 해결할 수 있다.

Single-destination shortest-paths problem:
주어진 목적지 정점 t 에 도착하는 각 정점 v 로부터 최단경로를 구하는 문제. 그래프의 각 `edge`의 방향을 reversing하여 이 문제를 single-source problem으로 해결할 수 있다.

Single-pair shortest-path problem:
주어진 u, v 가 있을 때, u 에서 v 로 가는 최단경로를 구하는 문제. source vertex u 에 대하여 single-source problem을 풀면, 이 문제 또한 해결할 수 있다. 게다가, 이 문제를 해결하기 위한 방법으로 알려진 모든 알고리즘들은 worst-case에서 점근적인(asymptotic) 런타임 소요 시간은 최상의 single-source algorithms과 동일하다.

All-pairs shortest-paths problem:
정점 u, v 의 모든 pair를 위해 u에서 v로 가는 최단경로를 찾는 문제. 비록 각 정점들에 대해 single-source algorithm을 실행하여 문제를 해결할 수 있지만, 보통 더 빠른 방법으로 해결할 수 있다. 게다가, 그 구조는 그 자체로 흥미로우며 그 내용은 따로 다루게 될 것이다.

optimal substructure of a shortest path

최단경로 알고리즘은 주로 두 정점의 최단거리는 다른 최단거리를 포함한다는 성질에 의존한다. 이는 DP 및 Greedy 에서 적용되는 성질과 유사하다. Dijkstra 알고리즘은 그리디 알고리즘이고, Floyd-Warshall 알고리즘은 DP 알고리즘이다. 다음과 같은 lemma는 최단 경로의 optimal-substructure property 를 더 정밀하게 해준다.

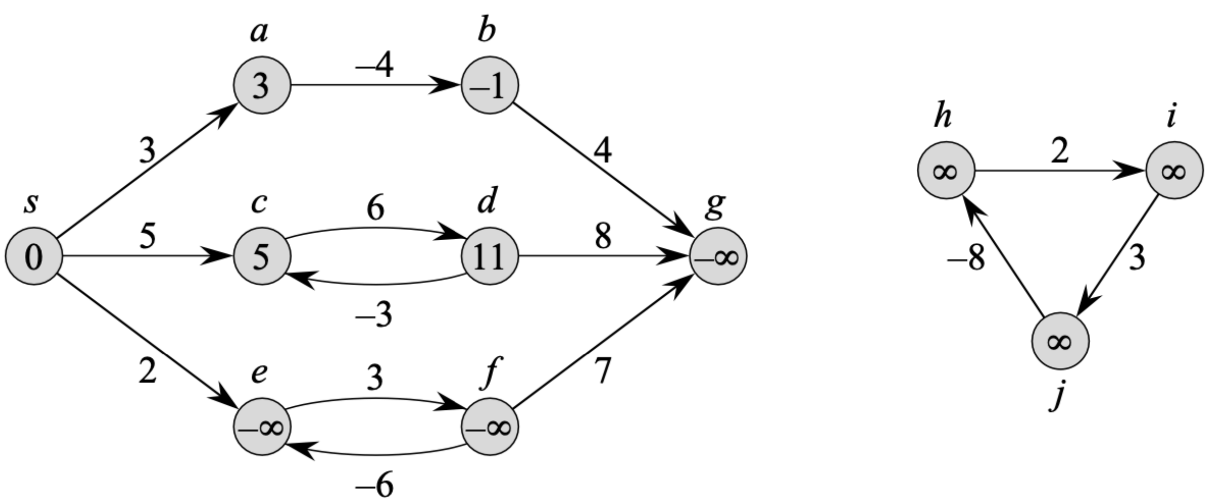
Lemma 24.1 (Subpaths of shortest paths are shortest paths)

가중치가 있고 방향이 있는 그래프 $G = (V, E)$ 와 weight function $w : E \to \mathbb{R}$ 가 주어졌을 때, $p =< v_0, v_1, \ldots, v_k >$ 를 정점 v_0 에서 v_k 로 가는 최단경로라 하자. 그리고 임의의 $0 \le i \le j \le k$ 를 만족하는 i, j 에 대하여 $p_{ij} =< v_i, v_{i+1}, \ldots, v_j >$ 를 정점 v_i 에서 v_j 로 가는 subpath라고 하자. 그러면 p_{ij}는 v_i 에서 v_j 로 가는 최단 경로이다.

proof

경로 p를 $v_0 \leadsto^{p_{0i}} v_i \leadsto^{p_{ij}} v_j \leadsto^{p_{jk}} v_k$ 로 분해하면, $w(p) = w(p_{0i}) + w(p_{ij}) + w(p_{jk})$ 가 된다. 이제, v_i 에서 v_j 로 가는 path $p_{ij}^{\grave{}}$ 가 존재하며 $w(p_{ij}^{\grave{}}) < w(p_{ij})$ 를 만족한다고 하자. 그러면 $v_0 \leadsto^{p_{0i}} v_i \leadsto^{p_{ij}^{\grave{}}} v_j \leadsto^{p_{jk}} v_k$ 는 v_1 에서 v_k 로 가면서 가중치 $w(p_{0i}) + w(p_{ij}^{\grave{}}) + w(p_{jk})$ 를 가지는데, 이는 $w(p)$보다 작아지게 되며 이는 p가 v_0 에서 v_k로 가는 가정에 모순된다.

Negative-weight edges



> *출처 : Introduction to Algorithms 3rd p646*
> *e, f, g는 negative-weight cycle을 가져서 weight가 $-\infty$ 이고, h, i, j 는 negative-weight cycle을 가지지만 s에서 도달가능하지 않기 때문에 weight가 ∞ 이다.

종종 single-source 최단경로 문제는 음의 가중치를 가진 `edge`를 포함할 수 있다. 만약 그래프 $G = (V, E)$ 가 source s로 부터 도달가능한 "음의 가중치 사이클(negative weight cycles)"이 존재하지 않는다면, 모든 $v \in V$ 에서 최단 경로 가중치 $\delta(s, v)$ 는 음의 가중치가 되더라도 잘 정의된다. 하지만 도달 가능한 "음의 가중치 사이클(negative weight cycles)"이 존재한다면, 최단경로의 가중치는 잘 정의되지 않는다. s에서 사이클 위의 다른 점으로 가는 path는 negative-weight cycle을 돌면서 더 짧은(shortest) path를 가지게 되기 때문이다. 따라서 s 에서 v로 가는 경로에 negative-weight cycle이 존재한다면 $\delta(s, v) = -\infty$ 로 정의한다.

`Dijkstra` 알고리즘의 경우 모든 `edge`의 가중치는 음이 아니라고 가정하고, `Bellman-Ford` 알고리즘은 음의 가중치를 가지는 `edge`를 허용하며, source에서 도달가능한 negative-weight cycle이 없을 때 정확한 답을 구해준다. 보통, negative-weight cycle가 존재하면 알고리즘은 이를 탐지해내게 된다.

Cycles

최단경로에선 cycle이 포함되지 않는다.

- 음의 가중치를 갖는 cycle은 위에서 살펴보았듯, 포함되지 않는다.
- 0이 아닌 양의 가중치를 갖는 cycle의 경우, `source` 에서 `destination` 까지 더 작은 weight를 가지는 경로로 움직이기 때문에, cycle가 존재할 수 없다. 즉, 만약 $p =< v_0, v_1, \ldots, v_k >$ 가 경로이고 $c =< v_i, v_{i+1}, \ldots, v_j >$ 가 이 경로의 양의 가중치를 갖는 cycle (positive-weight cycle)이면 (이때 $v_i = v_j$ and $w(c) > 0$) 경로 $p` =< v_0, v_1, \ldots, v_i, v_{j+1}, v_{j+2}, \ldots, v_k >$ 는 가중치 $w(p`) = w(p) - w(c) < w(p)$ 를 가지기 때문에 p는 v_0에서 v_k 로 가는 최단경로가 될 수 없다.
- 0의 가중치를 갖는 cycle은 어떤 경로에서든 0의 가중치를 갖는 cycle을 제거할 수 있다. 따라서, 만약 source s에서 destination v 로 가는 최단경로에 0의 가중치를 갖는 cycle가 있다면, 이 cycle이 없는 또다른 최단경로가 존재하게 된다. 최단경로에 0의 가중치를 갖는 cycle이 존재하는 한, 이 cycle이 더 이상 존재하지 않을 때 까지 제거할 수 있다.
  그러므로, 최단경로를 찾을 때, cycle이 존재하지 않는 simple paths들로 구성된다 라는 일반적인 특징을 잃지 않을 수 있다.
  따라서 어떤 그래프 $G = (V, E)$ 안에 acyclic path는 최대 $|V|$의 서로 다른 정점들로 구성되고, 또한 최대 $|V| - 1$ 개의 `edge`들로 구성된다.

Representing shortest paths

종종 최단경로의 가중치 뿐만 아니라 그 최단경로 위의 정점들을 구하고 싶을 수 있다. 그래프 $G = (V, E)$ 가 주어졌을 때, 각 정점 $v \in V$ 를 `predecessor v` 로 저장(maintain)한다. π 는 또다른 정점이거나 `NIL` 이다. 이 문서의 최단경로 알고리즘에선 정점 v에서 유래한 `predecessor`들의 chain이 s 부터 v 까지 최단경로를 따라 거꾸로 가도록 `π attribute`를 정의한다. 즉, $v.\pi \neq NIL$ 인 정점 v 가 주어졌을 때, `Print-Path(G, s, v)`는 s 부터 v 까지의 최단경로를 출력한다.

```
Print-Path(G, s, v)
        if v == s
                print s
        else if v.π == NIL
```

```
                print "no path from" s "to" v "exists"
        else Print-Path(G, s, v.π)
                print v
```

하지만 최단경로 알고리즘을 실행하는 도중에, π 값이 최단경로를 나타내지 않을 수 있다. BFS때와 마찬가지로, π 값에 의해 구해진 `predecessor subgraph` $G_\pi = (V_\pi, E_\pi)$ 를 생각해야 한다. 이때 정점의 집합 V_π 는 `non-NIL predecessors` 로 구성된 정점의 집합 G 와 `source` s 의 합으로 정의한다.

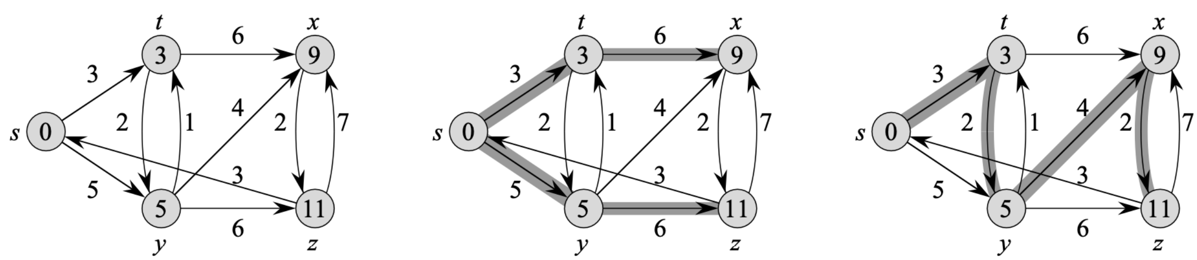
$$V_\pi = \{v \in V : v.\pi \neq NIL\} \cup \{s\}$$

방향이 있는 `edge` 집합 E_π 는 V_π 에 있는 정점들의 π 값에 의해 추론된 `edge`들의 집합이다.

$$E_\pi = \{(v.\pi, v) \in E : v \in V_\pi - \{s\}\}$$

이 문서에 나와있는 최단경로 알고리즘에 의해 만들어진 π 값들은 G_π 가 종료되는 때가 `shortest-paths tree` (즉, `source` s 에서 s로 부터 도달가능한 모든 정점까지의 최단 경로를 포함하는 `rooted tree`) 가 되는 성질이 있다는 것을 증명할 수 있다. `shortest-paths tree`는 `breadth-first tree`와 비슷하지만, `edge`의 개수 대신에 `edge`의 weight에 관한 `source` 로부터 최단경로를 포함하게 된다. 더 정확히 말하자면, 가중치가 있고, 방향이 있는 그래프 $G = (V, E)$를 정의하고 이 그래프의 weight function 이 $w : E \rightarrow \mathbb{R}$ 이며 shortest path들이 잘 정의 되도록 G에 `source` 정점 s 에서 도달가능한 `negative-weight cycle` 이 존재하지 않는다고 가정한다. s를 `root`로 하는 `shortest-paths tree` 는 `directed subgraph` $G\text{\`} = (V\text{\`}, E\text{\`})$ where $V\text{\`} \subseteq V$ and $E\text{\`} \subseteq E$ 이고 이때,

1. $V\text{\`}$ 는 G 안의 s에서 도달가능한 정점들의 집합
2. $G\text{\`}$ 는 `root`가 $s$인 `rooted tree`를 만들고,
3. 모든 $v \in V\text{\`}$ 에 대하여, $G\text{\`}$ 안의 s 에서 v 로 가는 유일한 `simple path`는 G 안의 s 에서 v로 가는 최단 경로이다.



(a) (b) (c)

> 출처: introduction to algorithms 3rd ed, p648
> (a) : `source` s 로 부터 최단 경로인 가중치와 방향이 있는 그래프
> (b) : 색칠된 `edge`는 `source` s 를 `root`로 하는 `shortest-paths tree`를 형성한다.
> (c) : 같은 `root`를 가지는 또다른 `shortest-paths tree`

최단경로는 꼭 하나만 존재할 필요 없으며, `shortest-paths tree`도 그렇다. 예를 들면, 위 그림은 같은 `root`에서 두개의 `shortest-paths trees` 가 나타난다.

Relaxation

이 챕터에서 알고리즘들은 `relaxation`이란 테크닉을 사용한다. 각 정점 $v \in V$ 에 대하여, $v.d$ 라는 `attribute`를 저장(maintain)하며, 이 `attribute`는 `source` s 에서 v 로 가는 최단경로 가중치의 `upper bound`를 의미한다. 이 $v.d$를 `shortest-path estimate`라고 명명하자. `shortest-path estimate`와 `predecessors(π)`를 다음 $\Theta(V)$ -time의 절차를 통해 초기화한다.

```
# pseudo code
Initialize-single-source(G, s)
        for each vertex v in G.V
                v.d = ∞
                v.π = NIL
        s.d = 0
```

```
// c++ code
void initializeSingleSource(Graph &G, Vertex &s){
        for (auto &v : G.V){
                v.d = INF;
                v.pi = nullptr;
        }
        s.d = 0;
}
```

edge (u, v)에 `relaxing` 하는 과정은 다음 $O(1)$-time의 절차를 통해 진행된다.

```
# pseudo code
Relax(u, v, w)
        if v.d > u.d + w(u,v)
                v.d = u.d + w(u,v)
                v.π = u
```

```
// c++ code
void Relax(Vertex &u, Vertex &v, function<int(Vertex&, Vertex&)> > w){
        if (v.d > u.d + w(u, v)){
                v.d = u.d + w(u, v);
                v.pi = u;
        }
}
```

이 장의 알고리즘들은 `Initialize-single-source`를 호출하고 여러번 `edge`들을 relaxing 한다. 게다가, `relaxation` 는 `shortest-path estimates`와 `predecessor`가 바뀌는 유일한 수단이 된다. 각 알고리즘들은 각 `edge`를 몇 번 relax 하는가, 그리고 어떤 순서로 `edge`를 relax 하는가에 따라 달라진다. `Dijkstra` 알고리즘과 `directed cyclic graph`를 위한 `shortest-paths`알고리즘은 각 `edge`를 정확히 한번만 `relaxing` 한다. `Bellman-Ford` 알고리즘은 각 `edge`를 $|V|-1$ 번 relaxing 한다.

Properties of shortest paths and relaxation

이 챕터의 알고리즘들을 증명하기 위해서, `shortest path`와 `relaxation`에 관한 몇가지 성질들을 살펴보자. 각 성질에서, 그래프는 `Initialize-single-source(G, s)` 를 통하여 초기화 되어 있고, `shortest-path estimates`와 `predecessor subgraph`는 일련의 `relaxation step`을 통해서 바뀐다고 가정하자.

- **Triangle inequality**
  - 어떤 $edge(u,v) \in E$ 에 대하여, $\delta(s,v) \le \delta(s,u) + w(u,v)$ 이다.
- **Upper-bound property**
  - 항상 모든 정점 $v \in V$ 에 대하여, $v.d \ge \delta(s,v)$ 이며 한번 $v.d$가 값 $\delta(s,v)$ 를 가지면 절대 변하지 않는다.
- **No-path property**
  - s 에서 v로 가는 경로가 없으면, 항상 $v.d = \delta(s,v) = \infty$ 이다.
- **Convergence property**
  - 만약 어떤 $u, v \in V$ 에 대하여 $s \rightsquigarrow u \to v$ 는 G 안의 최단경로 이고, $edge(u,v)$ 를 `relaxing` 하기 전 어느 때나 $u.d = \delta(s,u)$ 이면, 그 후 항상 $v.d = \delta(s,v)$ 이다.
- **Path-relaxation property**
  - 만약 $p =< v_0, v_1, \ldots, v_k >$ 가 $s = v_0$ 에서 v_k 로 가는 최단경로 이고, p의 `edge`들을 $(v_0,v_1),(v_1,v_2),\ldots,(v_{k-1},v_k)$ 의 순서로 `relax` 했다면, $v_k.d = \delta(s, v_k)$이다.
  - 이 성질은 p의 `edge`의 `relaxation`과 섞이는 상황까지 포함하여, 어떤 다른 `relaxation step`가 발생하더라도 무관하게 유지된다.
- **Predecessor-subgraph propery**
  - 모든 $v \in V$ 에 대하여 일단 $v.d = \delta(s,v)$를 만족하면, `predecessor subgraph`는 s를 `root`로 하는 `shortest-paths tree`이다.

후술할 알고리즘에서, directed graph G 는 adjacency-list으로 저장되어 있다고 가정하며, weight를 같이 저장하기 때문에 `edge`의 weight을 구하는데 $O(1)$의 시간이 걸린다고 가정한다.

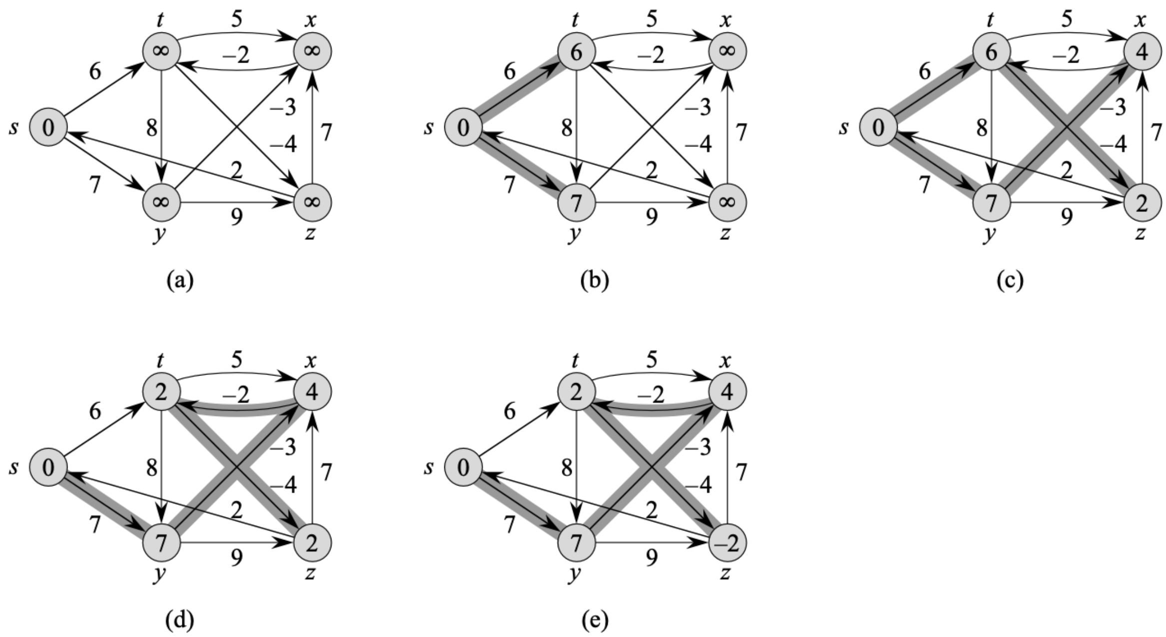
The Bellman-Ford algorithm

Bellman-Ford algorithm은 single-source shortest-path 문제의 일반적인 해결책으로, `edge`의 가중치가 음수인 상황에서도 작동한다. 가중치와 방향이 있고 `source` 가 s 이며 weight function이 $w: E \to \mathbb{R}$ 인 그래프 $G = (V, E)$가 주어졌을 때, `Bellman-Ford algorithm`은 source에서 도달할수 있는 negative-weight cycle이 있는지 여부를 `boolean`값으로 리턴한다. 만약 이런 cycle가 있으면, 해답이 존재하지 않다는 것을 의미하고, cycle이 없으면, `shortest path`와 그 가중치를 구한다.

이 알고리즘은 `edge`들을 relaxing 하고, 실제 최단경로의 가중치인 $\delta(s,v)$ 가 될 때 까지, `source` s 에서 각 정점 $v \in V$ 까지의 최단 경로 가중치 `estimate v.d` 를 점진적으로 감소시킨다.

```
# pseudo code
Bellman-Ford(G, w, s)
        Initialize-single-source(G, s)
        for i = 1 to |G.V| - 1
                for each edge (u,v) in G.E
                        Relax(u, v, w)
        for each edge(u, v) in G.E
                if v.d > u.d + w(u, v)
                        return false
        return true
```

```cpp
// c++ code
bool bellmanFord(Graph &G, function<int<Vertex &, Vertex&> > w, Vectex &s){
        initializeSingleSource(G, s);
        for (int i = 1; i < G.size() - 1; ++i){
                for (auto &edge : G.E)
                        Relax(edge.u, edge.v, w);
        }
        for (auto &edge : G.E){
                if (edge.v.d > edge.u.d + w(edge.u, edge.v))
                        return false;
        }
        return true;
}
```



> 출처: introduction to algorithms 3rd ed, p652
> Bellman-Ford 알고리즘의 작동 과정

Bellman-Ford 알고리즘은 $O(VE)$-time 의 시간복잡도를 가지며, 초기화에 $\Theta(V)$time 이 소요되고(line 1), $|V|-1$ 번만큼 for 루프를 돌면서 각 `edge`들을 체크하고($\Theta(E)$) (line 2-4) 마지막 for 루프에서 $O(E)$의 시간이 소요된다.
`Bellman-Ford` 알고리즘의 정확성을 증명하기 위해서, negative-weight cycle가 없다면 `source` 에서 도달 가능한 모든 정점들에 대하여 알고리즘이 정확한 shortest-path weight을 계산한다는것을 보이자.

Lemma 24.2

$G=(V,E)$ 를 가중치와 방향이 있고 source s 와 weight function $w: E->\mathbb{R}$ 을 가지는 그래프 라고 하자. 이때 G 에는 s에서 도달가능한 negative-weight cycles가 없다고 가정하자. 그러면 $|V|-1$ 번의 반복을 통하여 s 에서 도달 가능한 보든 정점 v에 대하여 $v.d=\delta(s,v)$ 를 구할 수 있다.

Proof

*lemma*를 증명하기 위해 path-relaxation 성질을 생각해보자. s에서 도달가능한 어떤 정점 v 를 고려해보자. 그리고 $v_0=s$, $v_k=v$ 는 s에서 v로 가는 어떤 최단 경로를 $p=<v_0,v_1,\ldots,v_k>$ 라 하자. `Shortest paths`들은 simple 이기 때문에, p는 최대 $|V|-1$ 개의 `edge`를 가지고, 따라서 $k\leq|V|-1$ 이다. 각 $|V|-1|$ 의 for loop 반복은 모든 $|E|$ `edge`들을 relaxing 한다. i번째 반복에서 `relax`된 `edge`들은, $i=1,2,\ldots,k$ 인 (v_{i-1},v_i) 이다. `path-relaxation property` 에 의하여,
$v.d=v_k.d=\delta(s,v_k)=\delta(s,v)$ 이다.

Corollary 24.3

$G = (V, E)$ 를 가중치와 방향이 있고 source s 와 weight function $w : E- > \mathbb{R}$ 을 가지는 그래프 라고 하자. 그리고 G 는 s에서 도달가능한 `negative-weight cycle`이 존재하지 않는다고 가정하자. 그러면, 각 정점 $v \in V$ 에 대하여 s에서 v로 가는 필요충분 조건은 `Bellman-Ford`를 G에 대하여 실행했을때, $v.d < \infty$ 로 종료되었을 때 이다.

Theorem 24.4 (Correctness of the Bellman-Ford algorithm)

`Bellman-Ford`를 G 에 적용하는데, 이때 $G = (V, E)$ 를 가중치와 방향이 있고 source s 와 weight function $w : E- > \mathbb{R}$ 을 가지는 그래프 라고 하자. 만약 G가 souece s에서 도달 가능한 `negative-weight cycle` 를 포함하지 않으면, `true`를 반환하고, 모든 정점 $v \in V$ 에 대하여 $v.d = \delta(s, v)$ 를 구할 수 있으며 `predecessor subgraph`인 G_π 는 s를 `root`로 하는 `shortest-paths tree` 이다. 만약 G 가 s에서 도달가능한 `negative-weight cycle`을 가진다면, `false`를 반환한다.

proof

그래프 G 가 source s 에서 도달가능한 `negative-weight cycle` 을 포함하지 않는다고 가정해보자. 일단, 모든 정점 $v \in V$ 에 대하여 종료시 $v.d = \delta(s, v)$ 를 만족한다는 `claim`을 증명해보자. 만약 v가 s에서 도달가능하다면, *lemma 24.2*가 이 `claim`을 증명한다. 만약 v가 s에서 도달가능하지 않다면, 이 `claim`은 `no-path property`를 따르게 된다. 따라서, 모든 상황에 대하여 증명이 된다. 이 `claim`과 함께, `predecessor-subgraph property`는 G_π 가 `shortest-paths tree`임을 암시한다. 이제 이 `claim`을 `Bellman-Ford`에서 `true`를 반환한다는것을 보이기 위해 사용해보자. 종료시, 모든 $edges(u, v) \in E$ 에 대하여,

$$\begin{aligned} v.d &= \delta(s, v) \\ &\le \delta(s, u) + w(u, v) \qquad \text{(by the triangle inequality)} \\ &= u.d + w(u, v), \end{aligned}$$

이고, 따라서 `Bellman-Ford` pseudo code에서 `false`를 반환하는 상황은 없다.
이제, 그래프 G가 s에서 도달 가능한 negative-weight cycle이 존재한다고 가정해보자. 이 cycle을 $v_0 = V_k$ 일때, $c =< v_0, v_1, \ldots, v_k >$ 라고 하자. 그러면,

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0 \qquad (24.\ 1).$$

반례를 도출하기 위해 `Bellman-Ford` 알고리즘에서 `true`를 반환한다고 가정해보자. 그러면, $v_i.d \le v_{i-1}.d + w(v_{i-1}, v_i)$ for $i = 1, 2, \ldots, k$ 이다. cycle c에서 부등식들을 더하면,

$$\begin{aligned} \sum_{i=1}^{k} v_i.d &\le \sum_{i=1}^{k} (v_{i-1}.d + w(v_{i-1}, v_i)) \\ &= \sum_{i=1}^{k} v_{i-1}.d + \sum_{i=1}^{k} w(v_{i-1}, v_i). \end{aligned}$$

$v_0 = v_k$ 이기 때문에, c의 각 정점들은 각 $\sum_{i=1}^{k} v_i.d$ 와 $\sum_{i=1}^{k} v_{i-1}.d$ 에서 한번만 나타나고, 그래서

$$\sum_{i=1}^{k} v_i.d = \sum_{i=1}^{k} v_{i-1}.d.$$

게다가, *Corollary 24.3* 에 의하여, $v_i.d$ for $i = 1, 2, \ldots, k$ 는 유한하다. 그러므로

$$0 \leq \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

이며 이는 부등식 (24.1)에 모순이다. 따라서 `Bellman-Ford` 알고리즘은 그래프 G에 `source`에서 도달가능한 `negative-weight cycle`이 존재하지 않는다면 `true`를 반환하고, 그렇지 않으면 `false`를 반환한다.

Dijkstra's algorithm

`Dijkstra's 알고리즘`은 가중치와 방향이 있는 그래프 $G = (V, E)$ 에서 모든 `edge`의 가중치가 음수가 아닌 상황의 `single-source shortest-paths` 문제를 해결한다. 그러므로 각 $edge(u, v)$ 에서 $w(u, v) \geq 0$ 이라 가정한다. 적절하게 구현한다면 `Dijkstra's algorithm`의 러닝타임은 `Bellman-Ford algorithm` 보다 빠르다.

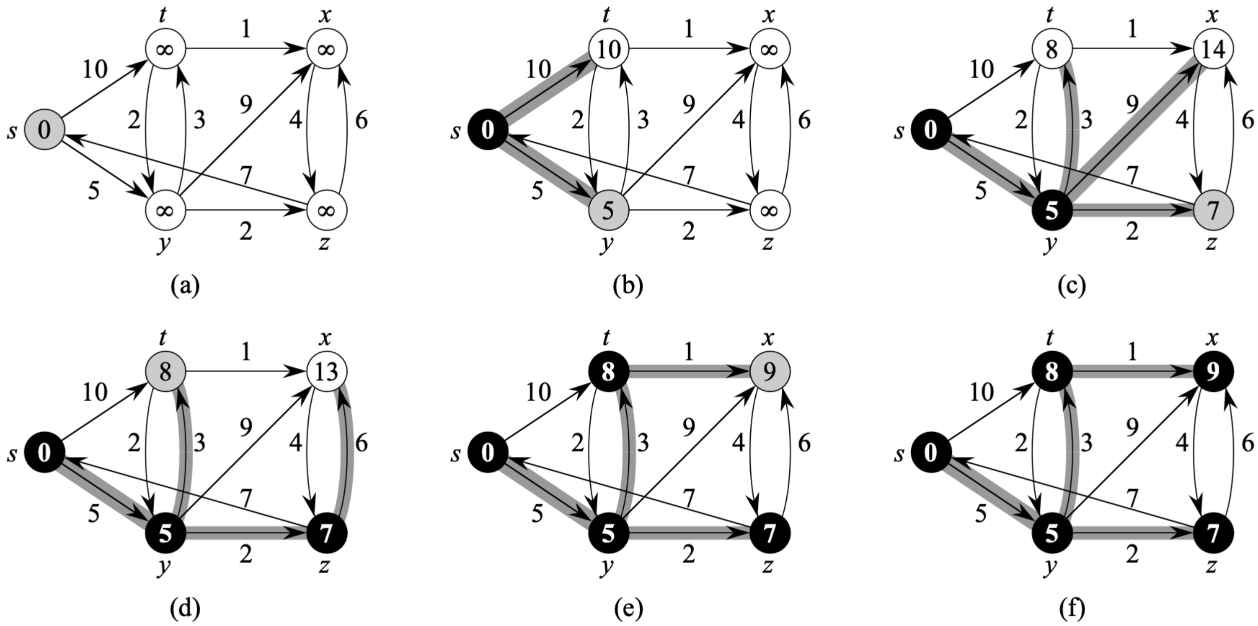
`Dijkstra's algorithm`은 정점들의 집합 S를 계속 저장(maintain)하는데, 이 S는 `source` s 로 부터 최종적인 최단경로의 가중치가 결정된 정점들로 구성된다. 반복적으로 `minimum shortest-path estimate` 인 정점 $u \in V - S$ 를 선택하고, u를 S에 더하며, u를 제외한 모든 `edge`를 `relax` 한다. 아래와 같은 구현에서, d를 `key`로 하는 정점들의 최소 우선순위 큐 Q 를 사용한다.

```
#pseudo code
Dijkstra(G, w, s)
	Initialize-single-source(G, s) # 초기화
	S = ø # set S 초기화
	Q = G.V # 모든 정점을 우선순위 큐 Q에 저장
	while Q != ø # Q = V - S 유지
		u = Extract-Min(Q)
		S = S U {u} # 정점 u는 항상 V - S 내의 임의의 정점의 smallest shortest-path estimate
		for each vertex v in G.Adj[u]
			Relax(u, v, w) # estimate u.d 와 predecessor u.π 업데이트
```

```
// c++ code
void Dijkstra(Graph &G, std::function<int (Vertex &u, Vertex &v)> w, Vertex &s){

}
```

우선순위 큐 Q에 삽입이 한 차례만 되고, S에 추가하는 과정도 한번만 되니 `line 4-8`의 `while loop`는 정확히 $|V|$ 번 돌게 된다.



> *출처: introduction to algorithms 3rd ed, p659*
> `Dijkstra's algorithms`의 진행 과정

`Dijkstra's algorithm`은 항상 $|V - S|$ 의 "lightest"이거나 "closest"인 정점을 집합 S에 추가하기 위해 선택하므로, `greedy strategy`를 사용한다고 볼 수 있다. 일반적으로 `greedy strategies`는 최적의 답을 내놓지 않지만, 아래의 `theorem`과 그 `corollary`에서 `Dijkstra's algorithm`은 실제 최단경로를 구해준다는 것을 보일 것이다. 핵심은 매번 정점 u를 집합 S에 넣어서, $u.d = \delta(s, u)$를 유지한다는것을 보이는 것이다.

Theorem 24.6 (Correctness of Dikstra's algorithm)

`Dijkstra's algorithm`은 `non-negative weight function` w 와 `source` s 가 있는 가중치와 방향이 있는 그래프 $G = (V, E)$ 에서 실행되며, 모든 정점 $u \in V$ 에 대하여 $u.d = \delta(s, u)$ 를 만족하며 종료된다.

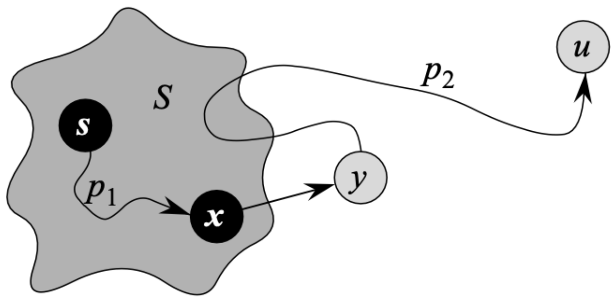
proof : 다음과 같은 loop invariant를 사용하자.

`line 4-8`의 `while loop` 반복을 시작할 때, $v \in S$ 인 각 정점에서 $v.d = \delta(s, v)$ 이다.

각 정점 $u \in V$ 에 대하여 u 가 S에 추가될때마다, $u.d = \delta(s, u)$를 만족한다는 것만 보이는 것으로 충분하다. 일단 $u.d = \delta(s, u)$ 라는 것을 보이면, `upper-bound peoperty`에 의존하여 그 이후 항상 등식이 성립함을 보이면 된다.

Initialization: 시작할 때, $S = \phi$ 이고, 따라서 이 `invariant`는 자명하게 참이다.

Maintenance: 각 반복(iteration)에서, 집합 S에 추가되는 정점이 $u.d = \delta(s, u)$를 보이고자 한다. 모순을 이끌어내기 위해서, 첫 정점 u가 집합 S에 추가될 때, $u.d \neq \delta(s, u)$ 라고 하자. u가 S에 추가되는 시점인 `while loop`의 시작점에 초점을 맞춰서, s 에서 u로 가는 최단경로를 살펴보며 $u.d = \delta(s, u)$ 라는 모순을 이끌어내보자. $u = s$ 이면 s가 집합 S에 처음으로 추가되는 정점이고, 그러면 $s.d = \delta(s, s) = 0$ 이기 때문에 $u \neq s$ 여야 한다. 또한 s에서 u로 가는 경로가 있어야 하는데, 그렇지 않으면 `no-path property` 에 의하여 $u.d = \delta(s, u) = \infty$ 인데, 이는 가설 $u.d \neq \delta(s.u)$ 를 위반하기 때문이다. 적어도 s에서 u로 가는 하나의 경로는 있어야 하기 때문에, s에서 u를 연결하는 최단경로 p가 존재한다. u를 S에 추가하기 전에, 경로 p는 S에 있는 하나의 정점, 즉 s와 다른 $V - S$에 해당하는 정점, 즉 u에 연결된다. p에 연결된 첫 정점 $y \in V - S$ 와 y의 `predecessor`이며 p와 연결된 정점 $x \in S$ 를 고려해보자. 아래 그림과 같이, 경로 p를 $s \overset{p}{\leadsto}_1 x \to y \overset{p}{\leadsto}_2 u.$ 로 분해해보자. (각 경로 p_1, p_2는 `edge`가 없을 수 있다.)



> *출처 : Introduction to algorithms 3rd ed, p660*
> Theorem 24.6 증명. 집합 S 는 u가 추가되기 전까지 공집합이다. `source` s로부터 최단경로인 p를 $s \overset{p}{\leadsto}_1 x \to y \overset{p}{\leadsto}_2 u$ 로 분해하는데, 이때 y는 S에 있지 않은 경로위의 첫 번째 정점이고, $x \in S$ 는 y의 바로 앞에 있는(immediately predecedes y) 정점이다. 정점 x, y는 distinct 하지만, $s = x$ 이거나 $y = u$ 일 수 있다. 경로 p_2는 집합 S에 다시 포함될 수도 아닐 수도 있다.

u가 S에 추가될 때, $y.d = \delta(s, y)$라고 `claim`을 세우자. 이 `claim`을 증명하기 위해 $x \in S$를 관찰해보자. 그러면 u가 S에 추가될 때 $u.d \neq \delta(s, u)$ 인 첫 정점으로 u를 선택했기 때문에, x가 S에 추가될 때, $x.d = \delta(s, x)$ 임을 알 수 있다. `edge` (x, y) 는 이때 `relax` 되기 때문에, 이 `claim`은 `convergence property`를 따른다.
이제 $u.d = \delta(s, u)$ 를 증명하기 위한 모순을 얻을 수 있다. y가 s에서 u로 가는 최단 경로인 u 이전에 나타나기 때문에, 그리고 모든 `edge`가 `non-negative`이기 때문에, $\delta(s, y) \leq \delta(s, u)$ 이고, 따라서

$$\begin{aligned} y.d &= \delta(s, y) \\ &\leq \delta(s, u) \qquad\qquad (24.2) \\ &\leq u.d \quad \text{(by the upper-bound property)}. \end{aligned}$$

하지만 `line 5`에서 u가 선택됐을 때, 정점 u와 y 둘 다 $V - S$ 에 포함되어 있으므로 $u.d \leq y.d$ 를 얻을 수 있다. 따라서 (24.2) 의 두 부등식이 사실상 등식으로 $y.d = \delta(s, y) = \delta(s, u) = u.d$ 와 같이 된다.
결과적으로, 우리가 선택한 u 가 $u.d = \delta(s, u)$ 가 되어 모순이 된다. 결국 u 가 S에 추가될 때, $u.d = \delta(s, u)$ 이며 이 등식은 항상 유지된다고 결론을 지을 수 있다.

Termination: 종료시, $Q = \phi$ 이며 이는 이전의 `invariant`인 $Q = V - S$ 를 따르기 때문에 $S = V$ 임을 암시한다. 결국, 모든 정점 $u \in V$ 에 대하여 $u.d = \delta(s, u)$ 이다.

Coroally 24.7

`Dijkstra's algorithm`을 가중치와 방향이 있고 `nonnegative weight function` w와 `source` s 를 가지는 그래프 $G = (V, E)$ 에 대하여 적용할 때, 종료시 `predecessor subgraph` G_π 는 s를 `root`로 하는 `shortest-paths tree` 이다.

proof

`Theorem 24.6`과 `predecessor-subtraph property`에 의하여 그러하다.

Analysis

`Dijkstra's algorithm`의 시간복잡도를 계산해보자. `min-priority queue`인 Q의 3가지 operation인 `Insert(line 3),Extract-min(line 5), Decrease-Key(in Relax)` 를 사용한다. `Insert`와 `Extract-Min`은 각 정점당 한번씩 호출하게 된다. $u \in V$ 인 정점들은 S에 정확히 한번씩만 추가되기 때문에, 일고리즘이 진행되면서 인접 리스트 `Adj[u]`의 각 `edge` 들은 `line 7-8`의 `for loop`에서 정확히 한번씩만 체크된다. 모든 인접 리스트의 `edge` 수는 $|E|$ 이고, 이 `for loop`는 총 $|E|$ 번 반복을 하고, 따라서 이 알고리즘은 최대 $|E|$ 번의 `Decrease-Key`를 호출한다.

`Dijkstra's algorithm`의 실행 시간은 어떻게 `min-priority queue`를 구현했는지에 따라 정해진다. 정점들이 1 부터 $|V|$까지 숫자로 이루어져 있어서 이점을 얻을 수 있는 우선순위 큐를 생각해보자. 그러면 단순히 $v.d$를 v번째 인덱스에 저장하는 배열로 구현할 수 있다. 각 `Insert`와 `Decrease-Key` operation은 $O(1)$-time 이 소요되고, 각 `Extract-Min` operation은 $O(V)$-time이 소요되며 전체 시간은 $O(V^2 + E) = O(V^2)$ 가 된다.

만약 그래프가 충분히 퍼져(sparse)있다면, 특히 $E = o(V^2 / lgV)$ 이면, `min-priority queue`를 `binary min-heap`을 이용하여 성능을 향샹시킬 수 있다. 그러면 각 `Extract-Min` operation은 $O(lgV)$이다. 이 operation을 $|V|$ 번 수행하게 된다. `binary min-heap`을 만드는데 소요되는 시간은 $O(V)$ 이다. 각 `Decrease-Key` operation은 $O(lgV)$ 의 시간이 소요되고, 여전히 이 operation을 최대 $|E|$번 수행하게 된다. 따라서 전체 소요 시간은 $O((V + E)lgV)$ 이며, 만약 모든 정점이 `source`에서 도달가능 하다면 $O(ElgV)$ 가 된다. 이 러닝타임은 $E = o(V^2 / lgV)$ 일 때, 확실히 향상된다.

`min-priority queue`를 `Fibonacci heap`으로 구현한다면, $O(VlgV + E)$ 까지 러닝 타임을 줄일 수 있다. 각 $|V|$ 번의 `Extract-Min` operation의 amortized cost는 $O(lgV)$ 이고, 각 `Decrease-Key` 호출은, 최대 $|E|$번 하게되며, 매번 $O(1)$의 amortized time이 소요된다. 역사적으로, `Fibonacci heap`은 `Dijkstra's algorithm`이 전형적으로 `Decrease-Key` 호출이 `Extract-Min`의 호출보다 훨 많다는 것에 모티브를 받아 개발되었으며, 그래서 `Extract-Min`의 amoritized time이 증가하지 않고도 각 `Decrease-Key` operation의 amortized time이 $o(lgV)$로 축소시킬 수 있으며, 이는 `binary heap`보다 점근적으로(asymptotically) 빠르게 된다.