Single-source Shortest Paths

Tao Hou

Problem Definition

Given a weighted (directed or undirected) graph $G = (V, E)$, a ***source*** vertex s and a ***target*** vertex t in G, compute a path from s to t of ***minimum weight*** (i.e., the ***shortest*** path)

- The weight is a function $w : E \longrightarrow \mathbb{R}$ on the edges
- The weight of a path is the sum of weights of all edges on the path

Two variations:

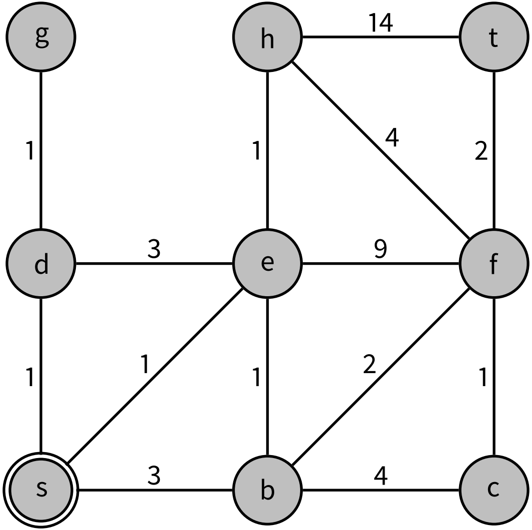
Single-source Shortest Paths

Given a ***source*** vertex s of G, compute the shortest paths from s to ***all other vertices***

All-pair Shortest Paths

Compute the shortest paths for all pairs of vertices

Example



A shortest path from s to t is: $s \longrightarrow e \longrightarrow b \longrightarrow f \longrightarrow t$ of weight 6

- In reality, the weight can be the length, cost, or time of roads, transportation lines etc.

More definitions

- The weight of the shortest path from s to t is called the ***distance***, or ***shortest-path distance***, from s to t and is denoted as $\delta(s, t)$.
- We have $\delta(s, t) = \infty$ if there is no path from s to t

More definitions

- The weight of the shortest path from s to t is called the ***distance***, or ***shortest-path distance***, from s to t and is denoted as $\delta(s, t)$.
- We have $\delta(s, t) = \infty$ if there is no path from s to t
- In the problem, edge weights can be ***negative***.
- However, if there is a ***negative-weight*** cycle on the path from s to t, $\delta(s, t)$ (as well as the problem) is not well-defined:
  - We can choose go through the cycles for arbitrary times and the weight of the path can arbitrarily lowered.

Algorithms

- BFS (Review)
- Dijkstra's algorithm
- Bellman-Ford
- An algorithm for DAG

Graph data structures

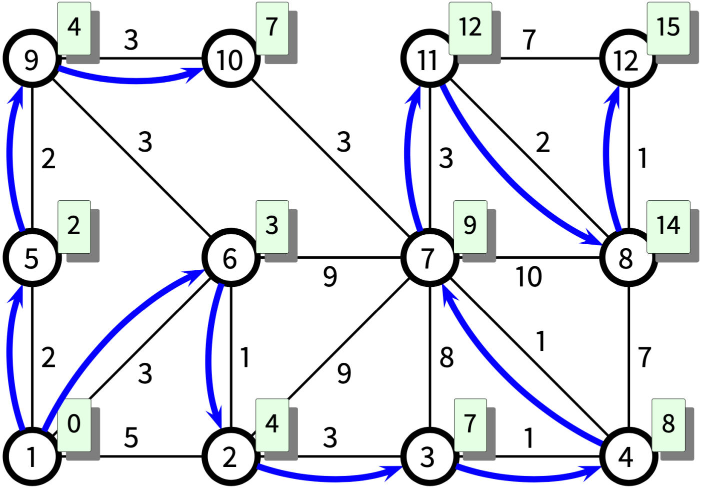
We assume ***adjacency list*** as data structures for graphs

Representing shortest paths from *s*

Through a ***shortest-path tree*** rooted at *s*, where the (unique) simple path from *s* to any vertex *v* in the tree is a shortest path from *s* to *v*

- There must be no cycles in a shortest path
  - The problem is not well-defined with negative-weight cycles
  - There can be no cycles with non-negative weight in a shortest path
- Shortest paths have the ***optimal substructure*** property
- We use $P[v]$ to record the parent of *v* in the tree (like in BFS/DFS)

Example of shortest path tree



Breadth-First Search

- One of the simplest but also a fundamental algorithm
  - Some more advance graph algorithm such as *Prim*'s and *Dijkstra*'s can be considered as built on BFS

Breadth-First Search

- One of the simplest but also a fundamental algorithm
  - Some more advance graph algorithm such as *Prim*'s and *Dijkstra*'s can be considered as built on BFS
- *Input:* $G = (V, E)$ and a ***source*** vertex $s \in V$
  - explores the graph starting from s, touching all vertices that are reachable from s
  - computes the distance of each vertex from s ('distance' means minimum number of edges)
  - iterates through the vertices at increasing distance
    - the algorithm discovers all vertices at distance k from s before discovering any vertices at distance $k + 1$ (hence the name)
  - produces a ***BFS tree*** rooted at s
    - An edge (u, v) in the tree means that v is '***discovered***' by visiting u
  - works on both *directed* and *undirected* graphs

Breadth-First Search

- One of the simplest but also a fundamental algorithm
  - Some more advance graph algorithm such as *Prim*'s and *Dijkstra*'s can be considered as built on BFS

- *Input:* $G = (V, E)$ and a ***source*** vertex $s \in V$
  - explores the graph starting from s, touching all vertices that are reachable from s
  - computes the distance of each vertex from s ('distance' means minimum number of edges)
  - iterates through the vertices at increasing distance
    - the algorithm discovers all vertices at distance k from s before discovering any vertices at distance $k + 1$ (hence the name)
  - produces a ***BFS tree*** rooted at s
    - An edge (u, v) in the tree means that v is '***discovered***' by visiting u
  - works on both *directed* and *undirected* graphs

- Breadth-first search computes the single-source shortest paths for s with ***weights of all edges being 1***.
  - The BFS tree is the shortest-path tree in this case

Breadth-First Search: High-level idea

A central data structure: A (FIFO) ***Queue***

Two phases of accessing a vertex u

- ***Discovering***: put u into the queue waiting to be *visited*
- ***Visiting***: access the adjacency list of u and try to *discover* each adjacent vertex

Breadth-First Search: High-level idea

A central data structure: A (FIFO) ***Queue***

Two phases of accessing a vertex u

- ***Discovering***: put u into the queue waiting to be *visited*
- ***Visiting***: access the adjacency list of u and try to *discover* each adjacent vertex

Process

- Initially, the seed s is the only vertex discovered (i.e., in the queue)
- Each iteration takes a vertex u from from the queue and visits u, until the queue is empty

Breadth-First Search: High-level idea

A central data structure: A (FIFO) ***Queue***

Two phases of accessing a vertex u

- ***Discovering***: put u into the queue waiting to be *visited*
- ***Visiting***: access the adjacency list of u and try to *discover* each adjacent vertex

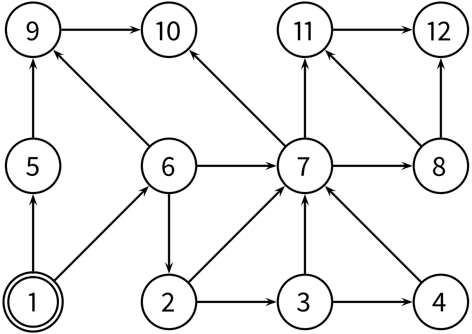
Process

- Initially, the seed s is the only vertex discovered (i.e., in the queue)
- Each iteration takes a vertex u from from the queue and visits u, until the queue is empty

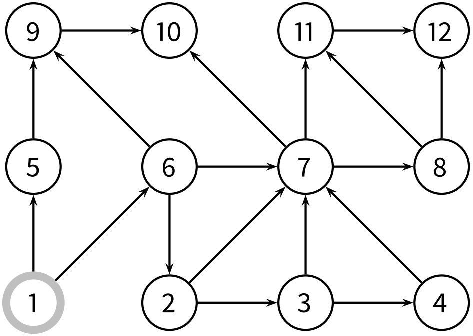
Coloring for vertices:

- ***white***: 'undiscovered', initial color
- ***gray***: 'discovered', but haven't been 'visited'
- ***black***: finished 'visiting'

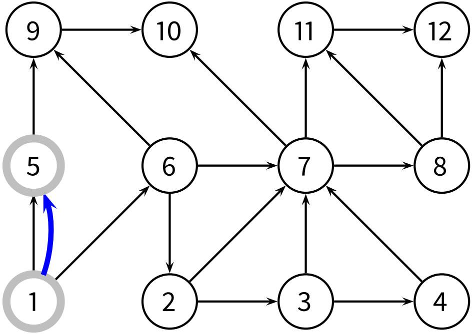
Example



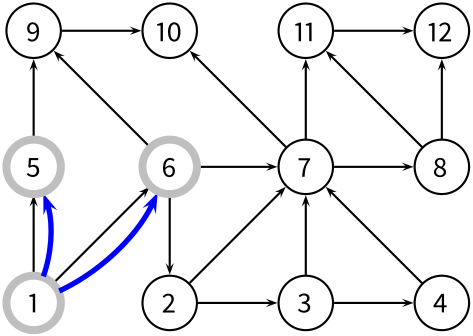
Example



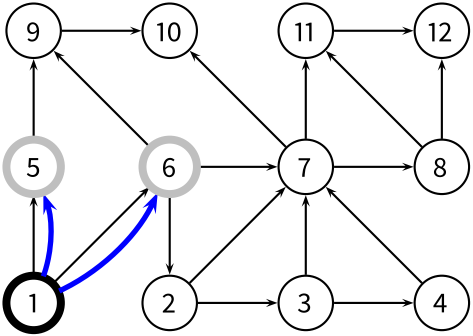
Example



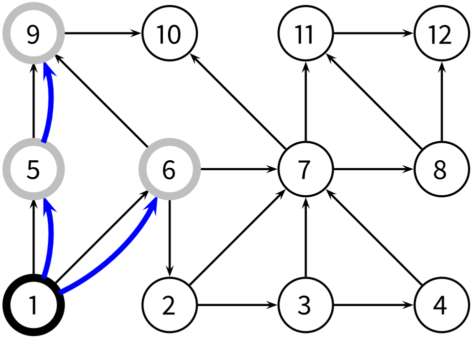
Example



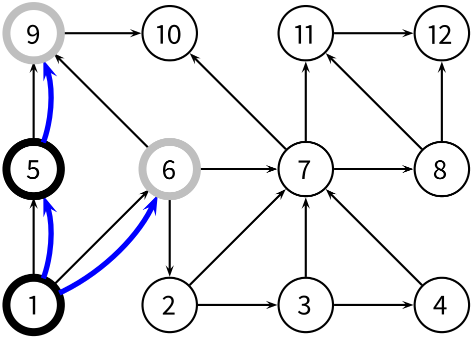
Example



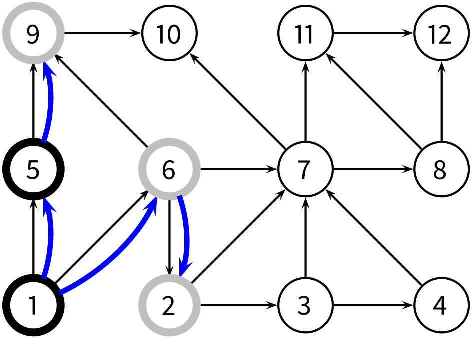
Example



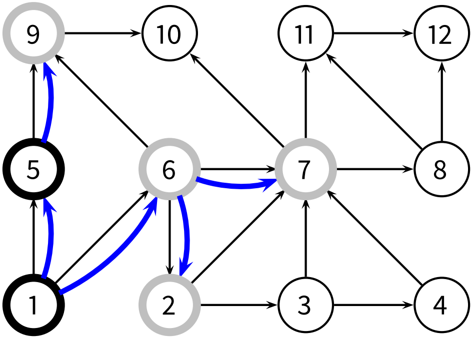
Example



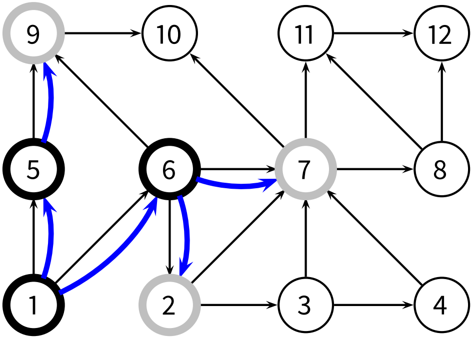
Example



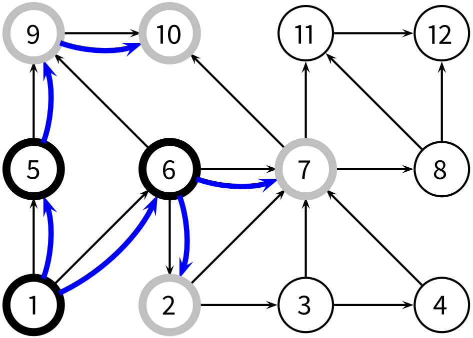
Example



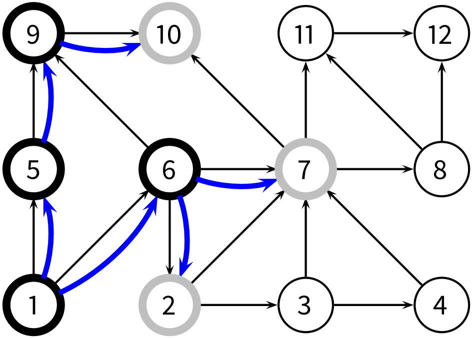
Example



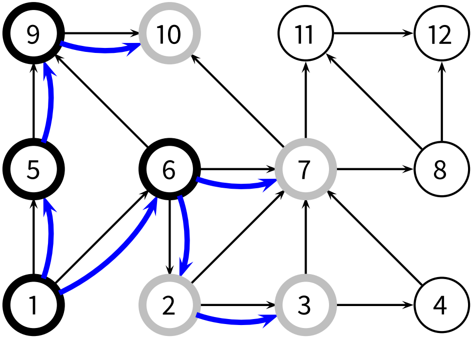
Example



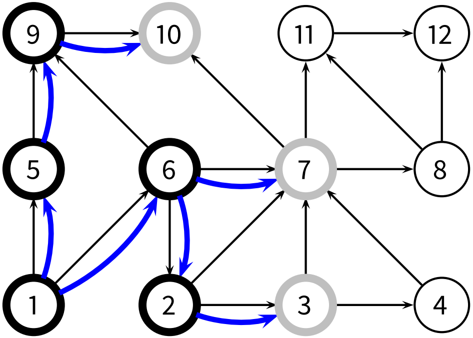
Example



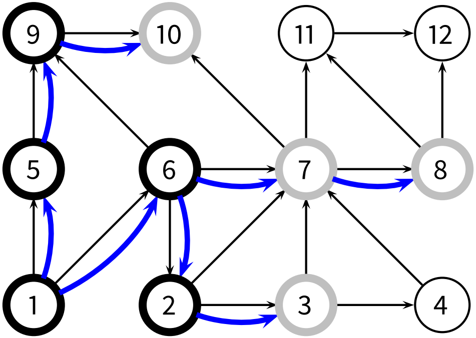
Example



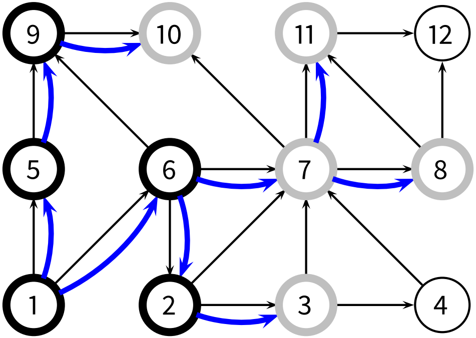
Example



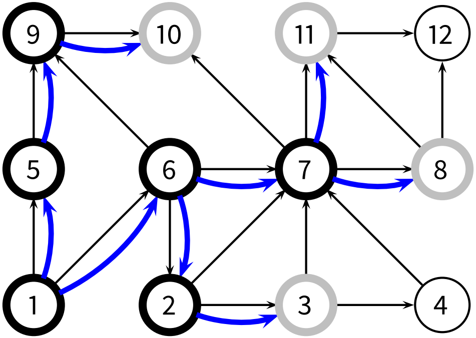
Example



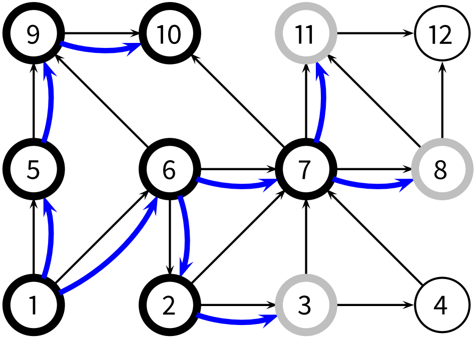
Example



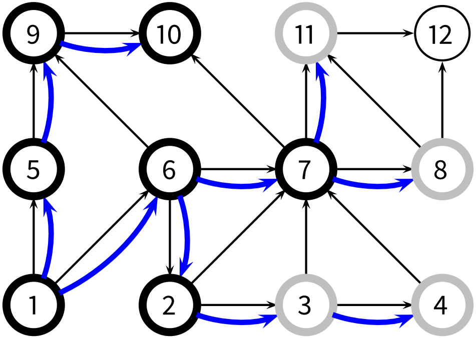
Example



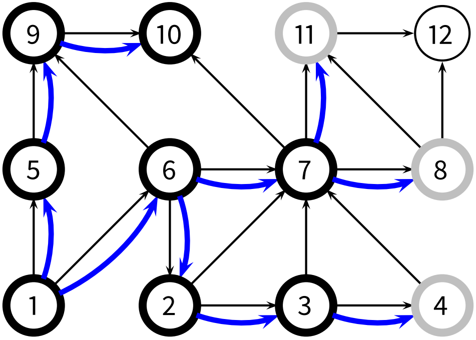
Example



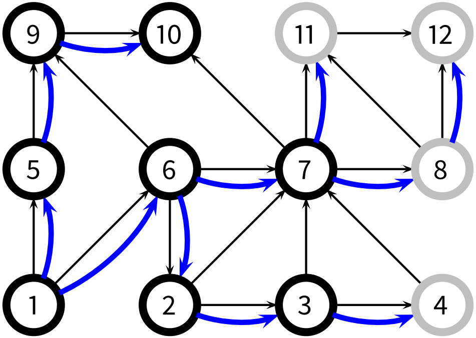
Example



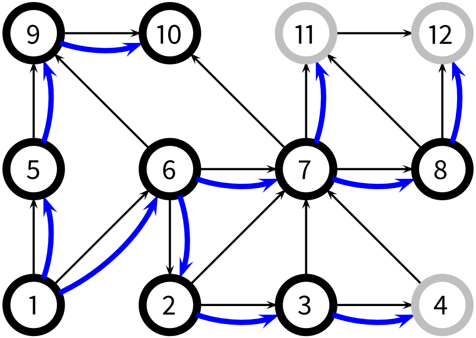
Example



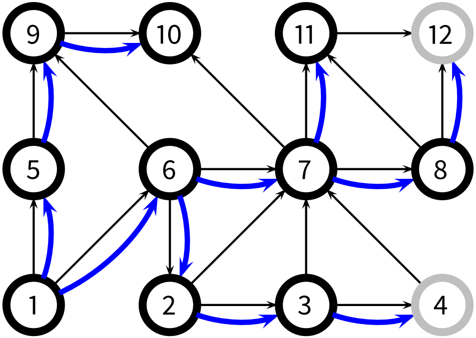
Example



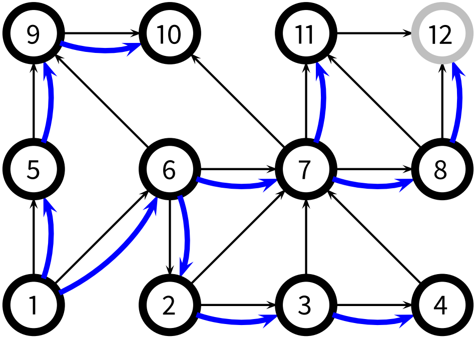
Example



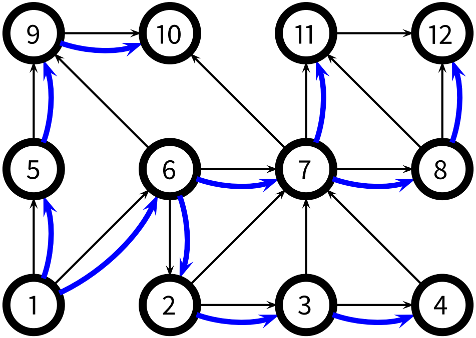
Example



Example



Example



```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
    color[u] = WHITE
    d[u] = ∞
    π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ Adj[u]
        if color[v] == WHITE
            color[v] = GRAY
            d[v] = d[u] + 1
            π[v] = u
            ENQUEUE(Q, v)
    color[u] = BLACK
```

- Coloring for vertices:
  - ***white***: ‘undiscovered’, initial color
  - ***gray***: ‘discovered’, but haven’t been ‘visited’
  - ***black***: finished ‘visiting’
-
  - ‘discovering’ means first encountered by the search
  - ‘visiting’ means to try to discover all adjacent vertices which are undiscovered
- Central data structure: a queue (first-in, first-out):
  - Contains ***gray*** vertices
- Some records we keep:
  - $color[u]$: color of a vertex u
  - $d[u]$: distance from s to u
  - $\pi[u]$: a vertex s.t. $(\pi[u], u)$ forms an edge in the BFS tree (there is another interpretation which we will see in Dijkstra’s)

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
        color[u] = WHITE
        d[u] = ∞
        π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
        u = DEQUEUE(Q)
        for each v ∈ Adj[u]
                if color[v] == WHITE
                        color[v] = GRAY
                        d[v] = d[u] + 1
                        π[v] = u
                        ENQUEUE(Q, v)
        color[u] = BLACK
```

- Initially, s is set as ‘discovered’: enqueued, ***gray***
  All other vertices are ‘undiscovered’ (***white***)

BFS Algorithm

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
        color[u] = WHITE
        d[u] = ∞
        π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
        u = DEQUEUE(Q)
        for each v ∈ Adj[u]
                if color[v] == WHITE
                        color[v] = GRAY
                        d[v] = d[u] + 1
                        π[v] = u
                        ENQUEUE(Q, v)
        color[u] = BLACK
```

- Initially, s is set as ‘discovered’: enqueued, ***gray***
  All other vertices are ‘undiscovered’ (***white***)
- Each iteration:
  - Dequeues a vertex u and tries to ‘discover’ (enqueue; mark as ***gray***) all its adjacent vertices which are ‘undiscovered’

BFS Algorithm

```
BFS(G, s)
  for each vertex u ∈ V(G) \ {s}
      color[u] = WHITE
      d[u] = ∞
      π[u] = NIL
  color[s] = GRAY
  d[s] = 0
  π[s] = NIL
  Q = ∅
  ENQUEUE(Q, s)
  while Q ≠ ∅
      u = DEQUEUE(Q)
      for each v ∈ Adj[u]
          if color[v] == WHITE
              color[v] = GRAY
              d[v] = d[u] + 1
              π[v] = u
              ENQUEUE(Q, v)
      color[u] = BLACK
```

- Initially, s is set as 'discovered': enqueued, ***gray***
  All other vertices are 'undiscovered' (***white***)
- Each iteration:
  - Dequeues a vertex u and tries to 'discover' (enqueue; mark as ***gray***) all its adjacent vertices which are 'undiscovered'
  - Whenever we discover a vertex v, we add the edge (u, v) to the BFS-tree (called a ***tree edge***)

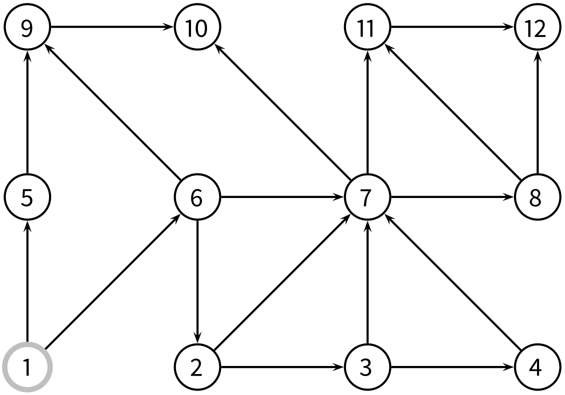
BFS Algorithm

```
BFS(G, s)
  for each vertex u ∈ V(G) \ {s}
        color[u] = WHITE
        d[u] = ∞
        π[u] = NIL
  color[s] = GRAY
  d[s] = 0
  π[s] = NIL
  Q = ∅
  ENQUEUE(Q, s)
  while Q ≠ ∅
        u = DEQUEUE(Q)
        for each v ∈ Adj[u]
              if color[v] == WHITE
                    color[v] = GRAY
                    d[v] = d[u] + 1
                    π[v] = u
                    ENQUEUE(Q, v)
        color[u] = BLACK
```

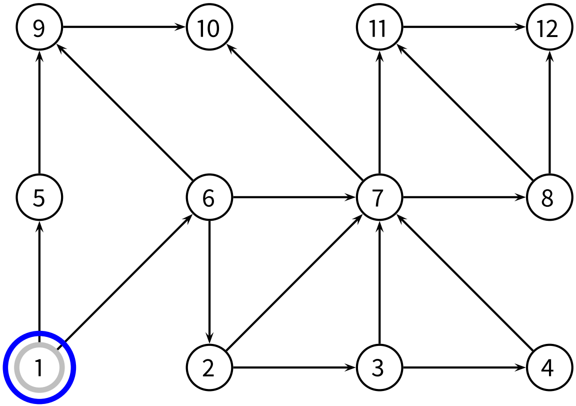
- Initially, s is set as 'discovered': enqueued, ***gray***
  All other vertices are 'undiscovered' (***white***)
- Each iteration:
  - Dequeues a vertex u and tries to 'discover' (enqueue; mark as ***gray***) all its adjacent vertices which are 'undiscovered'
  - Whenever we discover a vertex v, we add the edge (u, v) to the BFS-tree (called a ***tree edge***)
  - After this, we finished visiting u and mark u as ***black***.

BFS Algorithm

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
    color[u] = WHITE
    d[u] = ∞
    π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ Adj[u]
        if color[v] == WHITE
            color[v] = GRAY
            d[v] = d[u] + 1
            π[v] = u
            ENQUEUE(Q, v)
    color[u] = BLACK
```



```
BFS(G, s)
  for each vertex u ∈ V(G) \ {s}
      color[u] = WHITE
      d[u] = ∞
      π[u] = NIL
  color[s] = GRAY
  d[s] = 0
  π[s] = NIL
  Q = ∅
  ENQUEUE(Q, s)
  while Q ≠ ∅
      u = DEQUEUE(Q)
      for each v ∈ Adj[u]
          if color[v] == WHITE
              color[v] = GRAY
              d[v] = d[u] + 1
              π[v] = u
              ENQUEUE(Q, v)
      color[u] = BLACK
```

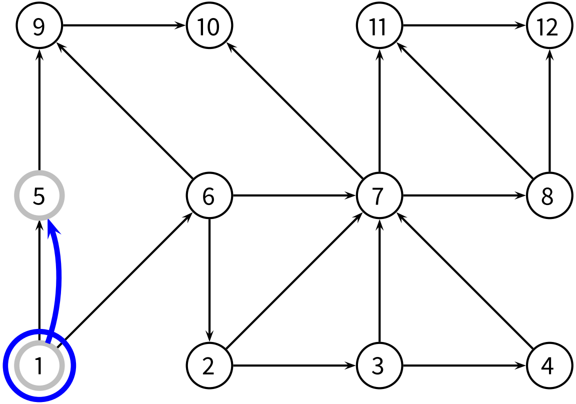


$u = 1$

$Q = \emptyset$

BFS Algorithm

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
        color[u] = WHITE
        d[u] = ∞
        π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
        u = DEQUEUE(Q)
        for each v ∈ Adj[u]
                if color[v] == WHITE
                        color[v] = GRAY
                        d[v] = d[u] + 1
                        π[v] = u
                        ENQUEUE(Q, v)
        color[u] = BLACK
```

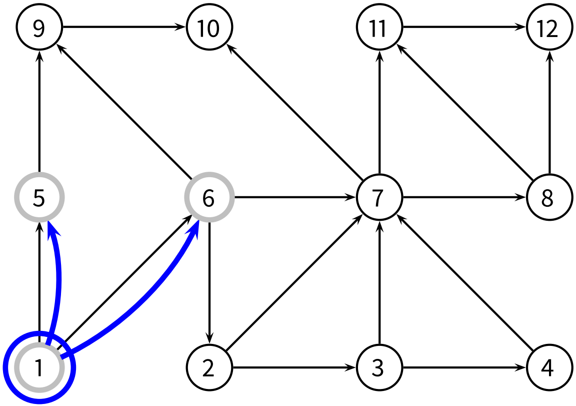


$u = 1$

$Q = \{5\}$

BFS Algorithm

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
        color[u] = WHITE
        d[u] = ∞
        π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
        u = DEQUEUE(Q)
        for each v ∈ Adj[u]
                if color[v] == WHITE
                        color[v] = GRAY
                        d[v] = d[u] + 1
                        π[v] = u
                        ENQUEUE(Q, v)
        color[u] = BLACK
```

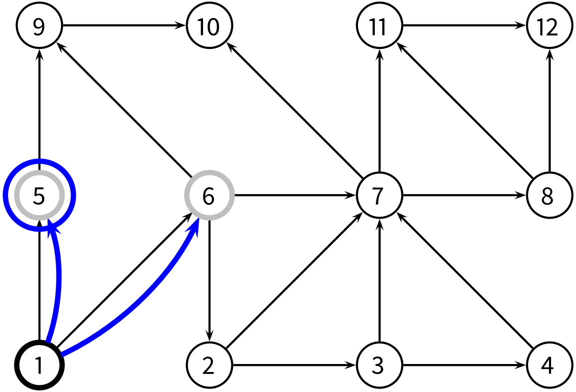


$u = 1$

$Q = \{5, 6\}$

BFS Algorithm

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
    color[u] = WHITE
    d[u] = ∞
    π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ Adj[u]
        if color[v] == WHITE
            color[v] = GRAY
            d[v] = d[u] + 1
            π[v] = u
            ENQUEUE(Q, v)
    color[u] = BLACK
```

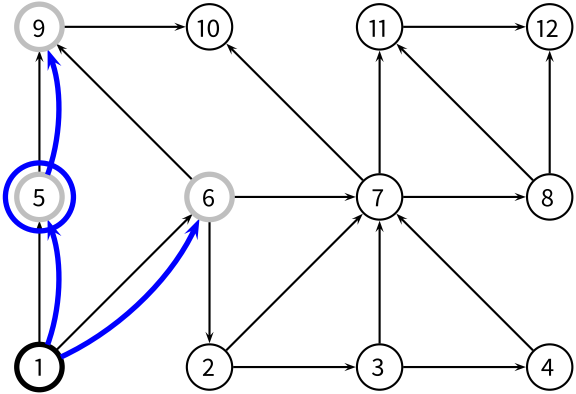


$u = 5$

$Q = \{6\}$

BFS Algorithm

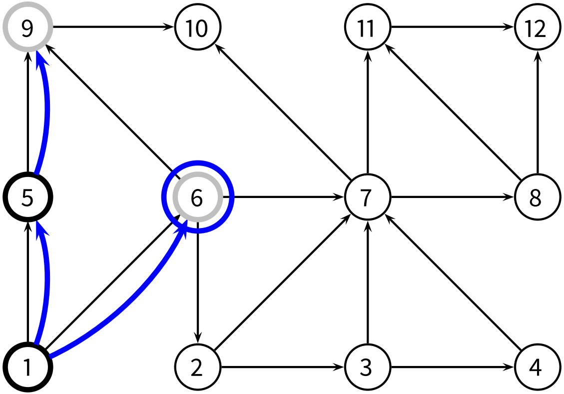
```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
        color[u] = WHITE
        d[u] = ∞
        π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
        u = DEQUEUE(Q)
        for each v ∈ Adj[u]
                if color[v] == WHITE
                        color[v] = GRAY
                        d[v] = d[u] + 1
                        π[v] = u
                        ENQUEUE(Q, v)
        color[u] = BLACK
```



$u = 5$

$Q = \{6, 9\}$

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
    color[u] = WHITE
    d[u] = ∞
    π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ Adj[u]
        if color[v] == WHITE
            color[v] = GRAY
            d[v] = d[u] + 1
            π[v] = u
            ENQUEUE(Q, v)
    color[u] = BLACK
```

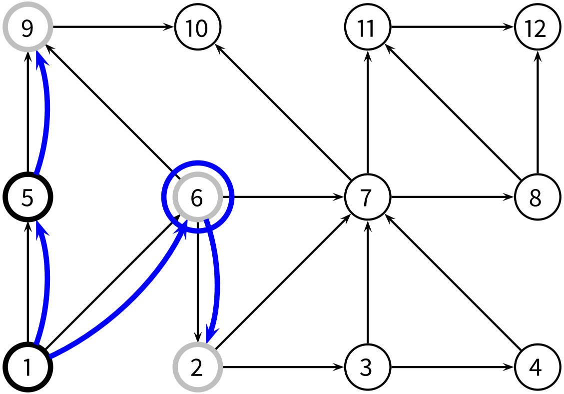


$u = 6$

$Q = \{9\}$

BFS Algorithm

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
        color[u] = WHITE
        d[u] = ∞
        π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
        u = DEQUEUE(Q)
        for each v ∈ Adj[u]
                if color[v] == WHITE
                        color[v] = GRAY
                        d[v] = d[u] + 1
                        π[v] = u
                        ENQUEUE(Q, v)
        color[u] = BLACK
```

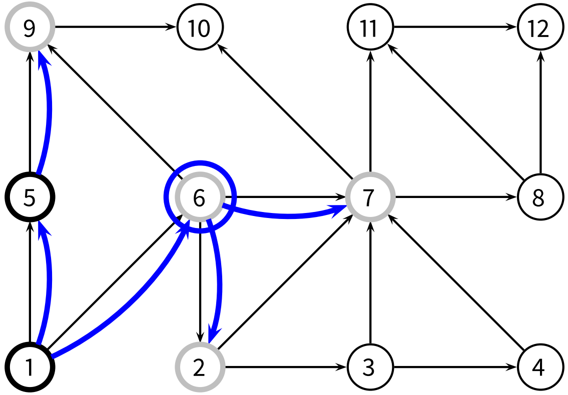


$u = 6$

$Q = \{9, 2, 7\}$

BFS Algorithm

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
        color[u] = WHITE
        d[u] = ∞
        π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
        u = DEQUEUE(Q)
        for each v ∈ Adj[u]
                if color[v] == WHITE
                        color[v] = GRAY
                        d[v] = d[u] + 1
                        π[v] = u
                        ENQUEUE(Q, v)
        color[u] = BLACK
```

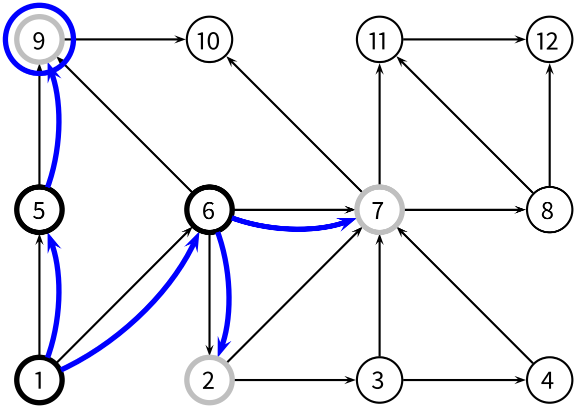


$u = 6$

$Q = \{9, 2, 7\}$

BFS Algorithm

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
    color[u] = WHITE
    d[u] = ∞
    π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ Adj[u]
        if color[v] == WHITE
            color[v] = GRAY
            d[v] = d[u] + 1
            π[v] = u
            ENQUEUE(Q, v)
    color[u] = BLACK
```

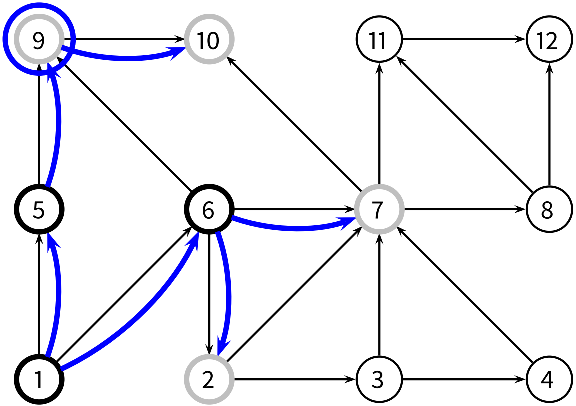


$u = 9$

$Q = \{2, 7\}$

BFS Algorithm

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
        color[u] = WHITE
        d[u] = ∞
        π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
        u = DEQUEUE(Q)
        for each v ∈ Adj[u]
                if color[v] == WHITE
                        color[v] = GRAY
                        d[v] = d[u] + 1
                        π[v] = u
                        ENQUEUE(Q, v)
        color[u] = BLACK
```

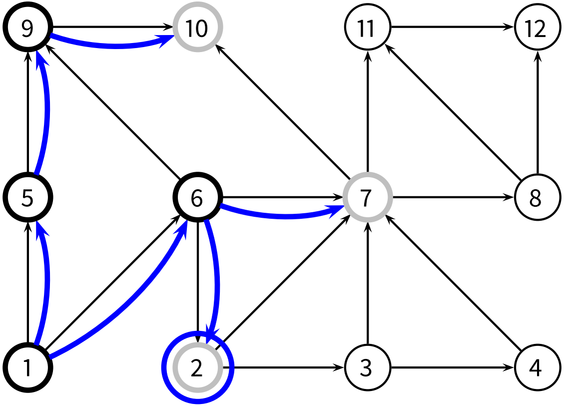


$u = 9$

$Q = \{2, 7, 10\}$

BFS Algorithm

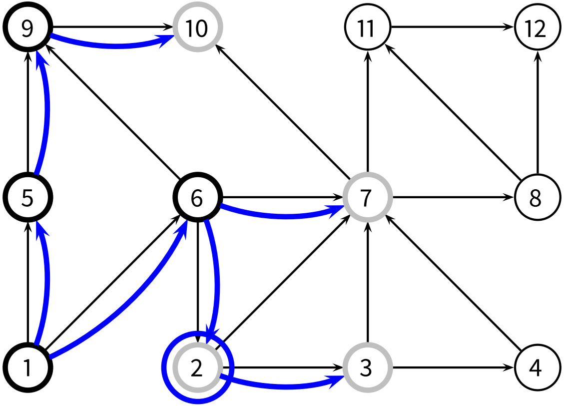
```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
    color[u] = WHITE
    d[u] = ∞
    π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ Adj[u]
        if color[v] == WHITE
            color[v] = GRAY
            d[v] = d[u] + 1
            π[v] = u
            ENQUEUE(Q, v)
    color[u] = BLACK
```



$u = 2$

$Q = \{7, 10\}$

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
    color[u] = WHITE
    d[u] = ∞
    π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ Adj[u]
        if color[v] == WHITE
            color[v] = GRAY
            d[v] = d[u] + 1
            π[v] = u
            ENQUEUE(Q, v)
    color[u] = BLACK
```

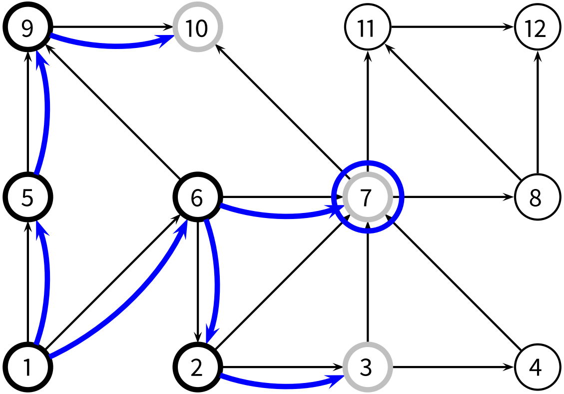


$u = 2$

$Q = \{7, 10, 3\}$

BFS Algorithm

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
        color[u] = WHITE
        d[u] = ∞
        π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
        u = DEQUEUE(Q)
        for each v ∈ Adj[u]
                if color[v] == WHITE
                        color[v] = GRAY
                        d[v] = d[u] + 1
                        π[v] = u
                        ENQUEUE(Q, v)
        color[u] = BLACK
```

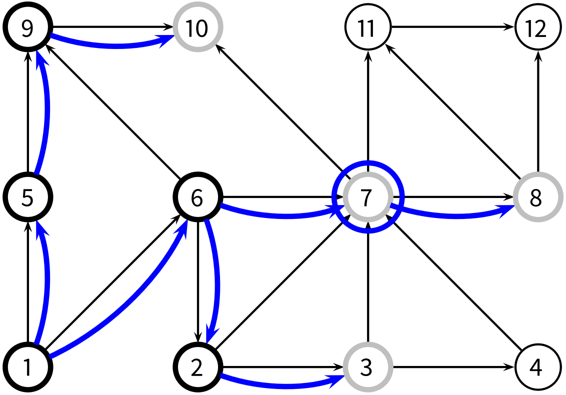


$u = 7$

$Q = \{10, 3\}$

BFS Algorithm

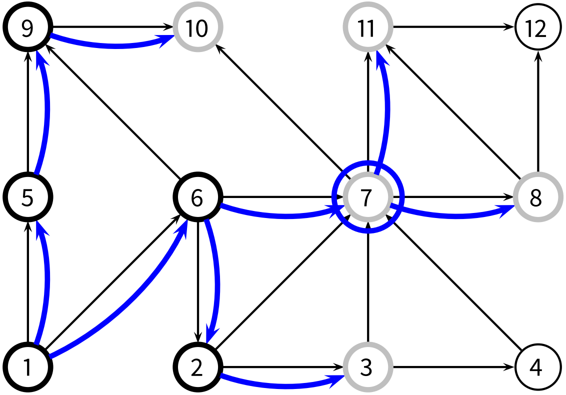
```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
    color[u] = WHITE
    d[u] = ∞
    π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ Adj[u]
        if color[v] == WHITE
            color[v] = GRAY
            d[v] = d[u] + 1
            π[v] = u
            ENQUEUE(Q, v)
    color[u] = BLACK
```



$u = 7$

$Q = \{10, 3, 8\}$

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
    color[u] = WHITE
    d[u] = ∞
    π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ Adj[u]
        if color[v] == WHITE
            color[v] = GRAY
            d[v] = d[u] + 1
            π[v] = u
            ENQUEUE(Q, v)
    color[u] = BLACK
```

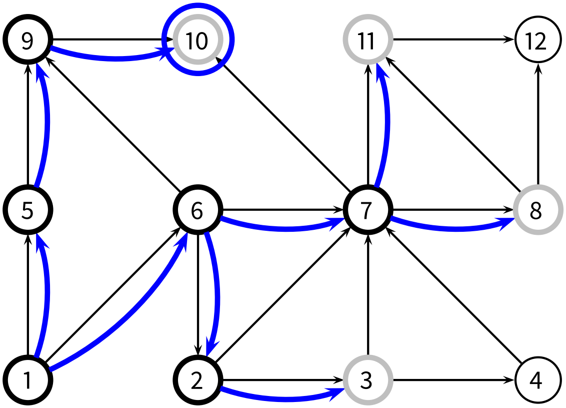


$u = 7$

$Q = \{10, 3, 8, 11\}$

BFS Algorithm

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
    color[u] = WHITE
    d[u] = ∞
    π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ Adj[u]
        if color[v] == WHITE
            color[v] = GRAY
            d[v] = d[u] + 1
            π[v] = u
            ENQUEUE(Q, v)
    color[u] = BLACK
```

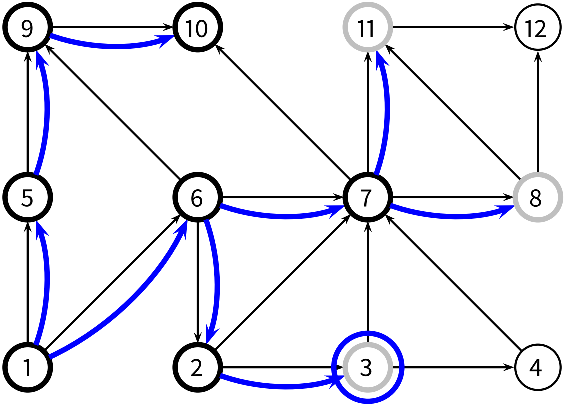


$u = 10$

$Q = \{3, 8, 11\}$

BFS Algorithm

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
    color[u] = WHITE
    d[u] = ∞
    π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ Adj[u]
        if color[v] == WHITE
            color[v] = GRAY
            d[v] = d[u] + 1
            π[v] = u
            ENQUEUE(Q, v)
    color[u] = BLACK
```

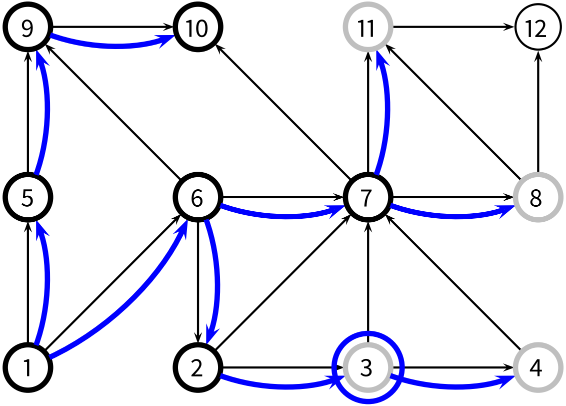


$u = 3$

$Q = \{8, 11\}$

BFS Algorithm

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
        color[u] = WHITE
        d[u] = ∞
        π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
        u = DEQUEUE(Q)
        for each v ∈ Adj[u]
                if color[v] == WHITE
                        color[v] = GRAY
                        d[v] = d[u] + 1
                        π[v] = u
                        ENQUEUE(Q, v)
        color[u] = BLACK
```

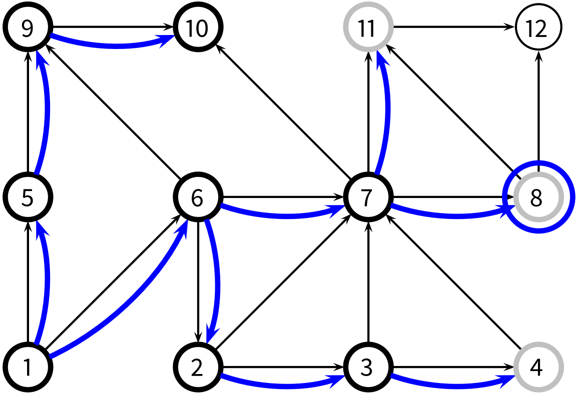


$u = 3$

$Q = \{8, 11, 4\}$

BFS Algorithm

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
        color[u] = WHITE
        d[u] = ∞
        π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
        u = DEQUEUE(Q)
        for each v ∈ Adj[u]
                if color[v] == WHITE
                        color[v] = GRAY
                        d[v] = d[u] + 1
                        π[v] = u
                        ENQUEUE(Q, v)
        color[u] = BLACK
```

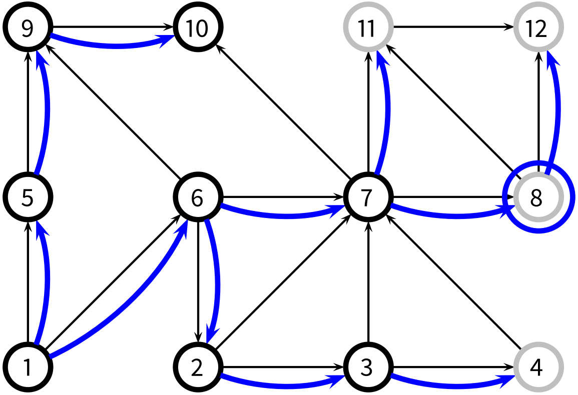


$u = 8$

$Q = \{11, 4\}$

BFS Algorithm

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
    color[u] = WHITE
    d[u] = ∞
    π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ Adj[u]
        if color[v] == WHITE
            color[v] = GRAY
            d[v] = d[u] + 1
            π[v] = u
            ENQUEUE(Q, v)
    color[u] = BLACK
```

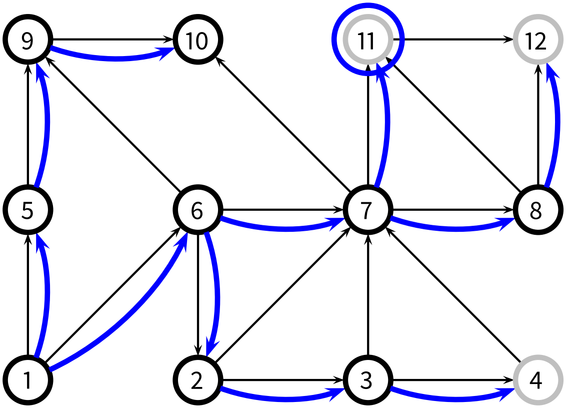


$u = 8$

$Q = \{11, 4, 12\}$

BFS Algorithm

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
    color[u] = WHITE
    d[u] = ∞
    π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ Adj[u]
        if color[v] == WHITE
            color[v] = GRAY
            d[v] = d[u] + 1
            π[v] = u
            ENQUEUE(Q, v)
    color[u] = BLACK
```

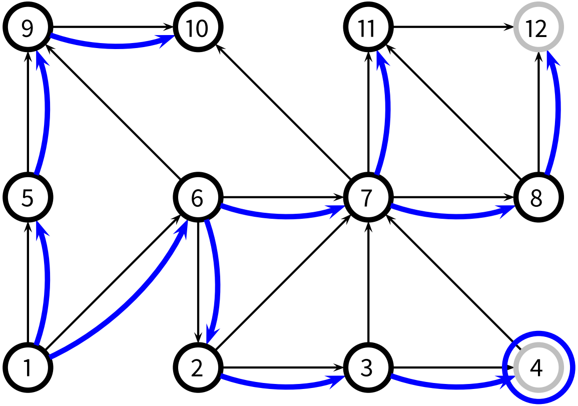


$u = 11$

$Q = \{4, 12\}$

BFS Algorithm

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
    color[u] = WHITE
    d[u] = ∞
    π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ Adj[u]
        if color[v] == WHITE
            color[v] = GRAY
            d[v] = d[u] + 1
            π[v] = u
            ENQUEUE(Q, v)
    color[u] = BLACK
```

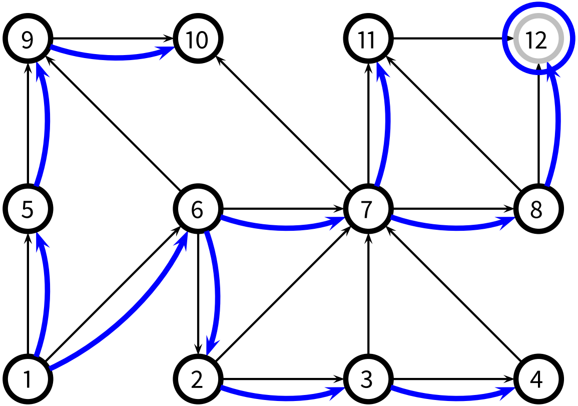


$u = 4$

$Q = \{12\}$

BFS Algorithm

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
    color[u] = WHITE
    d[u] = ∞
    π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ Adj[u]
        if color[v] == WHITE
            color[v] = GRAY
            d[v] = d[u] + 1
            π[v] = u
            ENQUEUE(Q, v)
    color[u] = BLACK
```

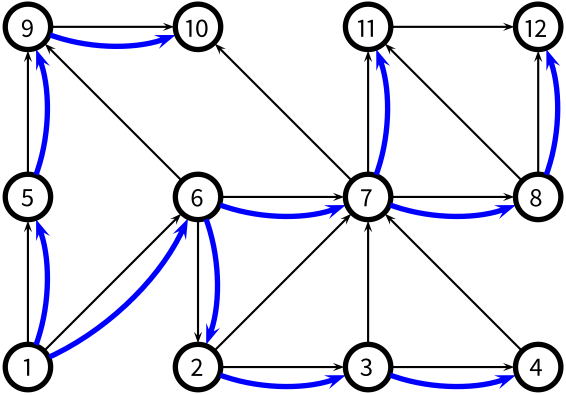


$u = 12$

$Q = \emptyset$

BFS Algorithm

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
    color[u] = WHITE
    d[u] = ∞
    π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ Adj[u]
        if color[v] == WHITE
            color[v] = GRAY
            d[v] = d[u] + 1
            π[v] = u
            ENQUEUE(Q, v)
    color[u] = BLACK
```



Complexity of BFS

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
    color[u] = WHITE
    d[u] = ∞
    π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ Adj[u]
        if color[v] == WHITE
            color[v] = GRAY
            d[v] = d[u] + 1
            π[v] = u
            ENQUEUE(Q, v)
    color[u] = BLACK
```

Complexity of BFS

```
BFS(G, s)
    for each vertex u ∈ V(G) \ {s}
        color[u] = WHITE
        d[u] = ∞
        π[u] = NIL
    color[s] = GRAY
    d[s] = 0
    π[s] = NIL
    Q = ∅
    ENQUEUE(Q, s)
    while Q ≠ ∅
        u = DEQUEUE(Q)
        for each v ∈ Adj[u]
            if color[v] == WHITE
                color[v] = GRAY
                d[v] = d[u] + 1
                π[v] = u
                ENQUEUE(Q, v)
        color[u] = BLACK
```

- We enqueue a vertex only if it is white, and we immediately color it gray; thus, we enqueue every vertex *at most once*

Complexity of BFS

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
    color[u] = WHITE
    d[u] = ∞
    π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ Adj[u]
        if color[v] == WHITE
            color[v] = GRAY
            d[v] = d[u] + 1
            π[v] = u
            ENQUEUE(Q, v)
    color[u] = BLACK
```

- We enqueue a vertex only if it is white, and we immediately color it gray; thus, we enqueue every vertex *at most once*
- This means that we also dequeue every vertex *at most once*
- So, the (dequeue) while loop executes $O(|V|)$ times

Complexity of BFS

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
    color[u] = WHITE
    d[u] = ∞
    π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ Adj[u]
        if color[v] == WHITE
            color[v] = GRAY
            d[v] = d[u] + 1
            π[v] = u
            ENQUEUE(Q, v)
    color[u] = BLACK
```

- We enqueue a vertex only if it is white, and we immediately color it gray; thus, we enqueue every vertex *at most once*
- This means that we also dequeue every vertex *at most once*
- So, the (dequeue) while loop executes $O(|V|)$ times
- The inner loop: For each vertex u, the inner loop executes for no more than out-deg(u) times, for a total of $\sum_{u \in V}$ out-deg$(u) =$

Complexity of BFS

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
        color[u] = WHITE
        d[u] = ∞
        π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
        u = DEQUEUE(Q)
        for each v ∈ Adj[u]
                if color[v] == WHITE
                        color[v] = GRAY
                        d[v] = d[u] + 1
                        π[v] = u
                        ENQUEUE(Q, v)
        color[u] = BLACK
```

- We enqueue a vertex only if it is white, and we immediately color it gray; thus, we enqueue every vertex *at most once*
- This means that we also dequeue every vertex *at most once*
- So, the (dequeue) while loop executes $O(|V|)$ times
- The inner loop: For each vertex u, the inner loop executes for no more than out-deg(u) times, for a total of $\sum_{u \in V}$ out-deg$(u) = |E|$ times

Complexity of BFS

```
BFS(G, s)
for each vertex u ∈ V(G) \ {s}
    color[u] = WHITE
    d[u] = ∞
    π[u] = NIL
color[s] = GRAY
d[s] = 0
π[s] = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ Adj[u]
        if color[v] == WHITE
            color[v] = GRAY
            d[v] = d[u] + 1
            π[v] = u
            ENQUEUE(Q, v)
    color[u] = BLACK
```

- We enqueue a vertex only if it is white, and we immediately color it gray; thus, we enqueue every vertex *at most once*
- This means that we also dequeue every vertex *at most once*
- So, the (dequeue) while loop executes $O(|V|)$ times
- The inner loop: For each vertex u, the inner loop executes for no more than out-deg(u) times, for a total of $\sum_{u \in V}$ out-deg(u) $= |E|$ times
- So, $O(|V| + |E|)$ (because $|E|$ may be less than $|V|$)

BFS gives the shortest-path distance

Claim

Let $\delta(s, v)$ be the minimum numbers of edges of any path from s to v. Then, we claim that $d[v] = \delta(s, v)$.

BFS gives the shortest-path distance

Claim

Let $\delta(s, v)$ be the minimum numbers of edges of any path from s to v. Then, we claim that $d[v] = \delta(s, v)$.

Reason:

- First, we observe that, before a vertex with distance $k + 1$ is visited, all vertices with distance $\leq k$ have been visited
  - At the beginning, there is only one vertex s with distance ≤ 0, and s is the first vertex visited. When visiting s, we discover all vertices with distance ≤ 1
  - When visiting vertices with distance 1, we discover all vertices with distance ≤ 2
  - ... (The predecessor of every vertex with distance k must be a vertex with distance $k - 1$)

BFS gives the shortest-path distance

Claim

Let $\delta(s, v)$ be the minimum numbers of edges of any path from s to v. Then, we claim that $d[v] = \delta(s, v)$.

Reason:

- First, we observe that, before a vertex with distance $k + 1$ is visited, all vertices with distance $\leq k$ have been visited
  - At the beginning, there is only one vertex s with distance ≤ 0, and s is the first vertex visited. When visiting s, we discover all vertices with distance ≤ 1
  - When visiting vertices with distance 1, we discover all vertices with distance ≤ 2
  - ... (The predecessor of every vertex with distance k must be a vertex with distance $k - 1$)
- For contradiction, suppose the claim is not true, and consider the first vertex v visited by BFS s.t. $\delta(s, v) \neq d[v]$
- Assume v is discovered when visiting u. We have $d[v] = d[u] + 1$.
- Since $\delta(s, v) \leq d[v]$, we have that $\delta(s, v) < d[v]$.

BFS gives the shortest-path distance

Claim

Let $\delta(s, v)$ be the minimum numbers of edges of any path from s to v. Then, we claim that $d[v] = \delta(s, v)$.

Reason:

- First, we observe that, before a vertex with distance $k + 1$ is visited, all vertices with distance $\leq k$ have been visited
  - At the beginning, there is only one vertex s with distance ≤ 0, and s is the first vertex visited. When visiting s, we discover all vertices with distance ≤ 1
  - When visiting vertices with distance 1, we discover all vertices with distance ≤ 2
  - ... (The predecessor of every vertex with distance k must be a vertex with distance $k - 1$)
- For contradiction, suppose the claim is not true, and consider the first vertex v visited by BFS s.t. $\delta(s, v) \neq d[v]$
- Assume v is discovered when visiting u. We have $d[v] = d[u] + 1$.
- Since $\delta(s, v) \leq d[v]$, we have that $\delta(s, v) < d[v]$.
- Let w be the predecessor of v on a shortest path from s to v. We have $\delta(s, w) = \delta(s, v) - 1$. So $\delta(s, w) < d[v] - 1 = d[u] = \delta(s, u)$.
- This means that w must be visited before u, and when we visit w, we must have marked v as gray. This contradicts the fact that when we visit u, the color of v is still white.

Why the edges $(\pi[v], v)$ form a tree?

Here we prove that the underlying undirected graph is an (undirected) tree

Why the edges $(\pi[v], v)$ form a tree?

Here we prove that the underlying undirected graph is an (undirected) tree

- Consider only vertices connected to s (i.e., in the connected component containing s), and let the number of vertices connected to s be n_0

Why the edges $(\pi[v], v)$ form a tree?

Here we prove that the underlying undirected graph is an (undirected) tree

- Consider only vertices connected to s (i.e., in the connected component containing s), and let the number of vertices connected to s be n_0
- The number of tree edges:

Why the edges $(\pi[v], v)$ form a tree?

Here we prove that the underlying undirected graph is an (undirected) tree

- Consider only vertices connected to s (i.e., in the connected component containing s), and let the number of vertices connected to s be n_0
- The number of tree edges: $n_0 - 1$

Why the edges $(\pi[v], v)$ form a tree?

Here we prove that the underlying undirected graph is an (undirected) tree

- Consider only vertices connected to s (i.e., in the connected component containing s), and let the number of vertices connected to s be n_0
- The number of tree edges: $n_0 - 1$
- The underlying undirected graph formed by these vertices and edges is definitely connected (because we are only attaching an edge to the partial graph we are building)

Why the edges $(\pi[v], v)$ form a tree?

Here we prove that the underlying undirected graph is an (undirected) tree

- Consider only vertices connected to s (i.e., in the connected component containing s), and let the number of vertices connected to s be n_0
- The number of tree edges: $n_0 - 1$
- The underlying undirected graph formed by these vertices and edges is definitely connected (because we are only attaching an edge to the partial graph we are building)
- Previous fact: A connected, undirected graph with n vertices and $n-1$ edges is a tree

Dijkstra's Algorithm

- Assumes all edges have ***non-negative*** weights
- A ***greedy*** algorithm

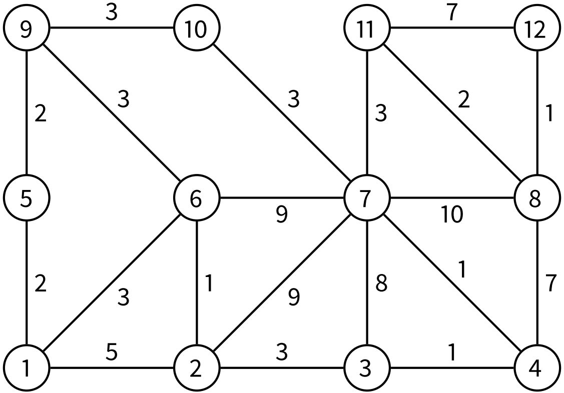
Dijkstra's Algorithm

```
DIJKSTRA(G, s, w)
N = ∅
for each vertex v ∈ V(G)
        D[v] = ∞
        P[v] = NIL
D[s] = 0
while N ≠ V(G)
        find u ∉ N such that D[u] is minimal
        N = N ∪ {u}
        for all v ∈ Adj(u) \ N
                if D[u] + w(u, v) < D[v]
                        D[v] = D[u] + w(u, v)
                        P[v] = u
```

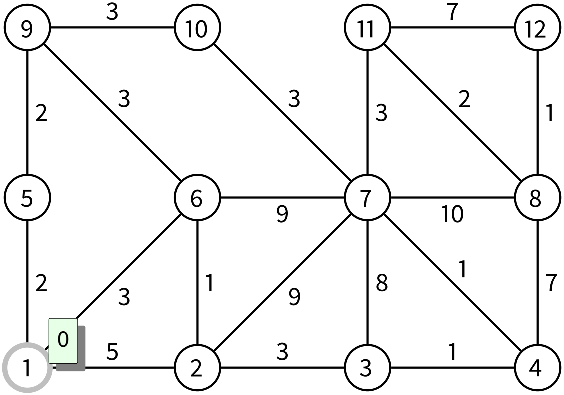
(all algorithms for S-S have the same ***initialization*** and ***relaxation*** process)

- Grows a shortest-path tree from s
  - N: vertices in the (partial) shortest path tree
  - $P[v]$: parent of v in the (partial) shortest path tree (also the vertex preceding v on the shortest path from s)
- Maintains an 'estimate' of the distance to v
  - $D[v]$: weight of the shortest path from s to v where all edges other than the last is from the partial tree
- In each step, makes a greedy choice by adding to the tree a vertex u with minimum value of D
- After adding u to the tree, updates $D[v]$ for the neighbors of u outside the tree if needed (*relaxation*)
- Stops when the tree spans the graph

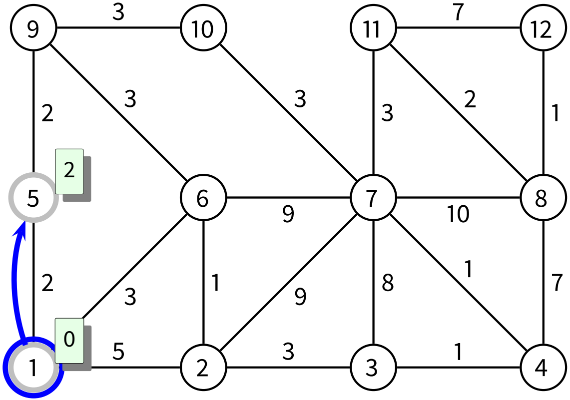
Example



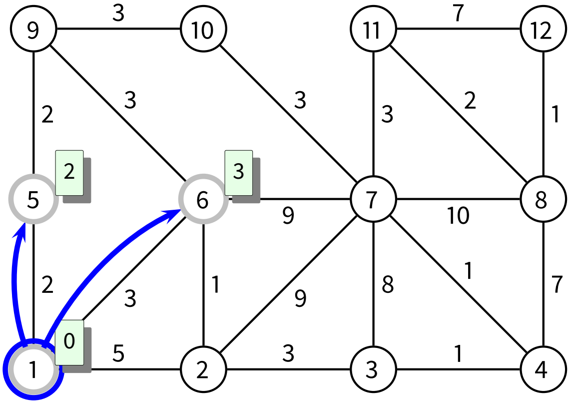
Example



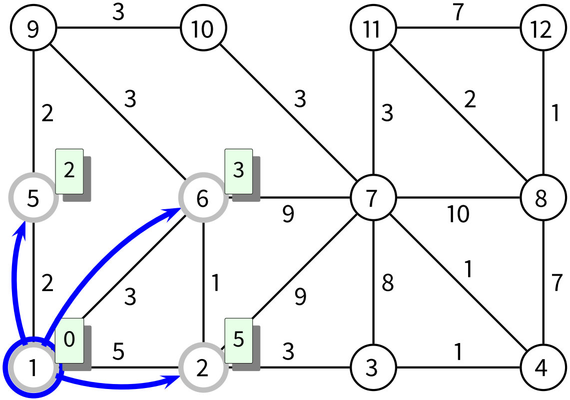
Example



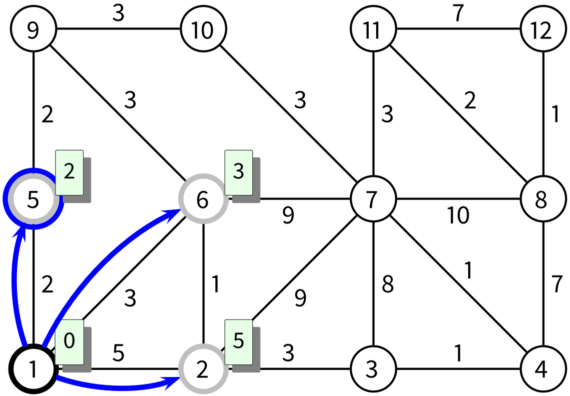
Example



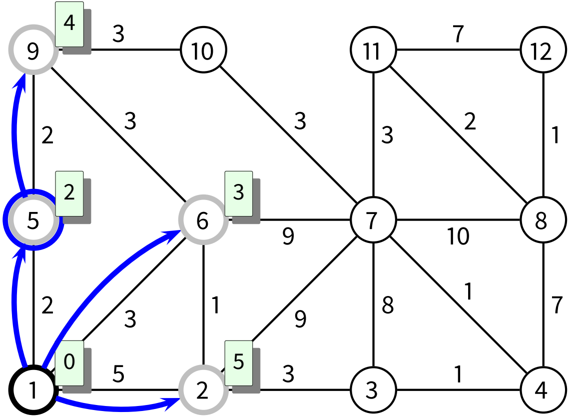
Example



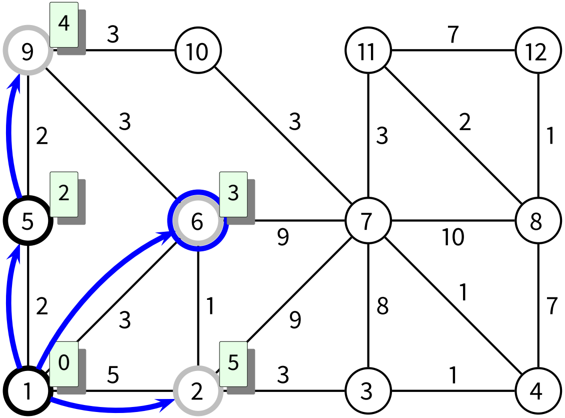
Example



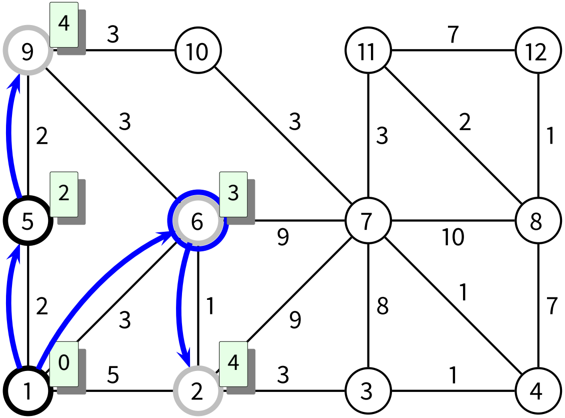
Example



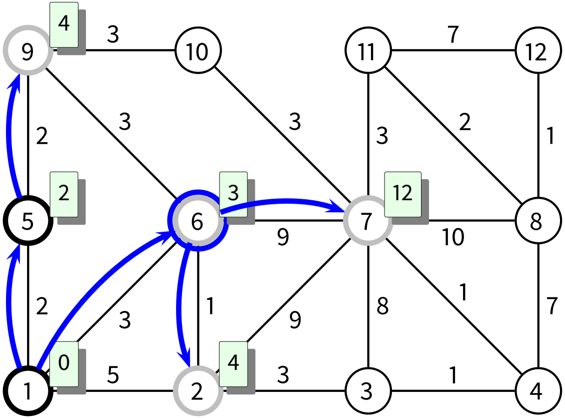
Example



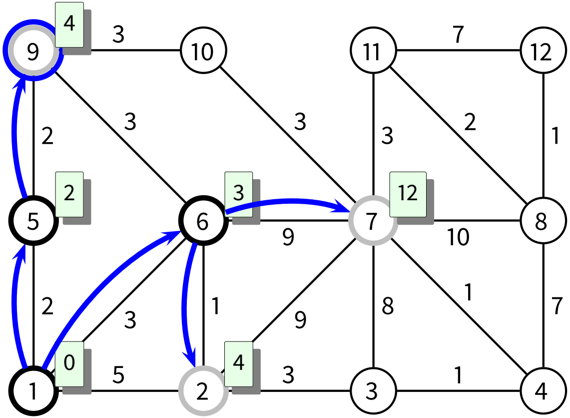
Example



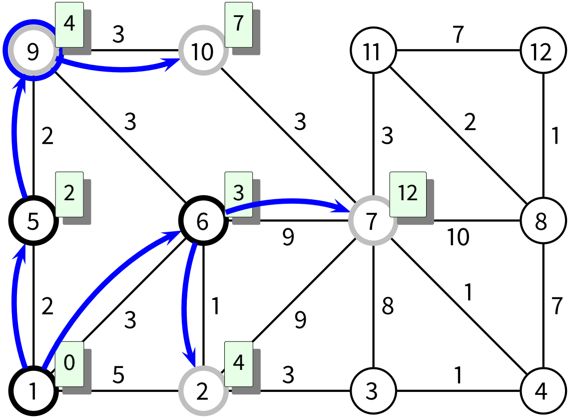
Example



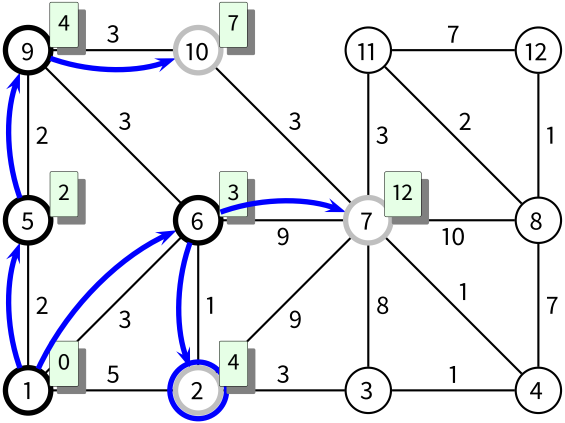
Example



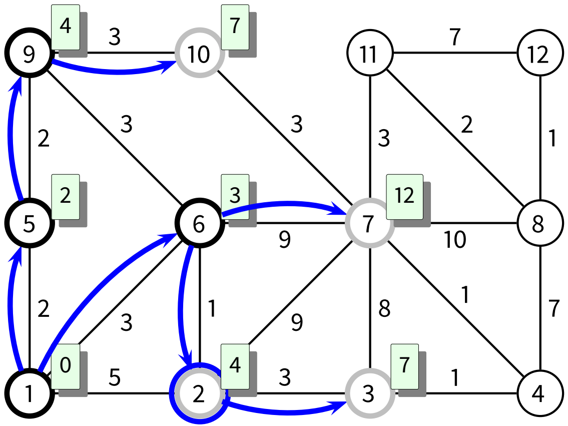
Example



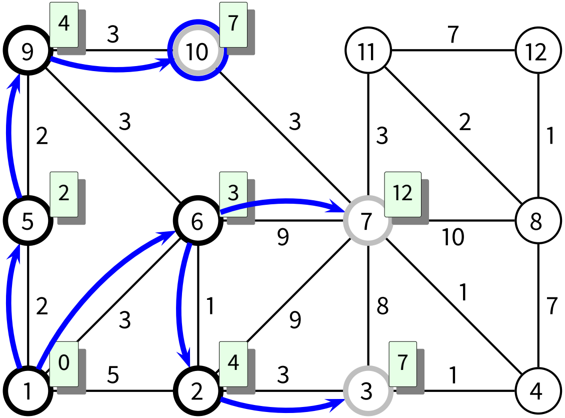
Example



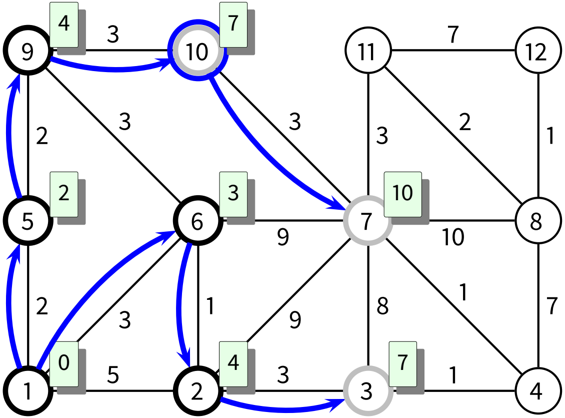
Example



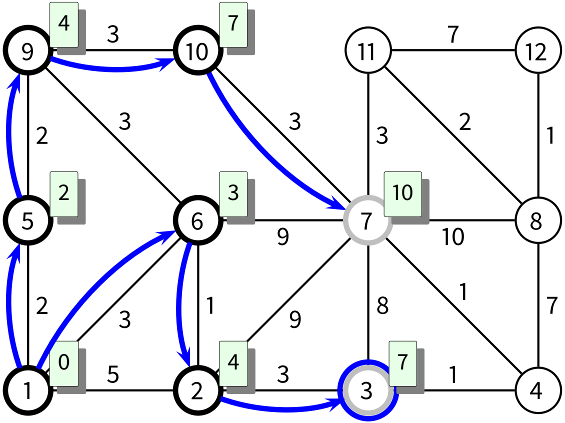
Example



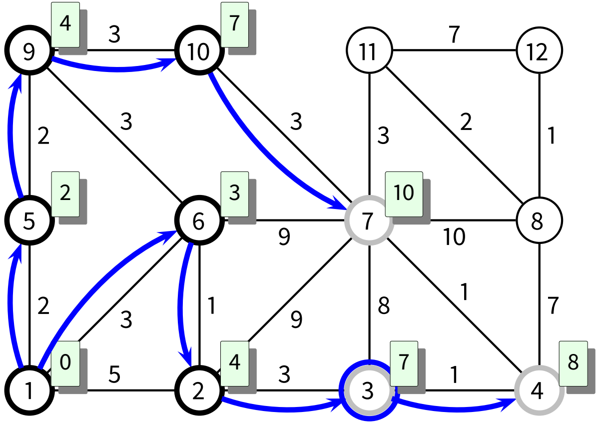
Example



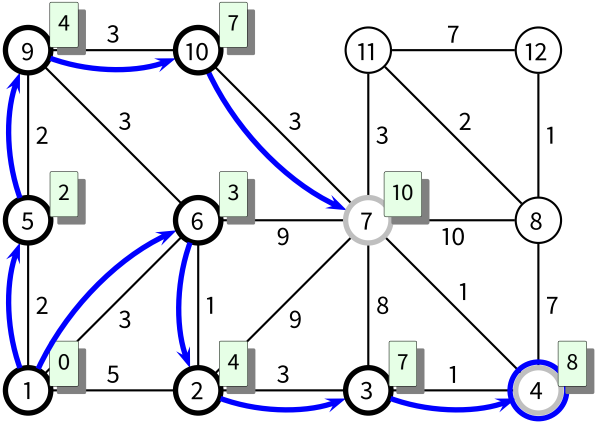
Example



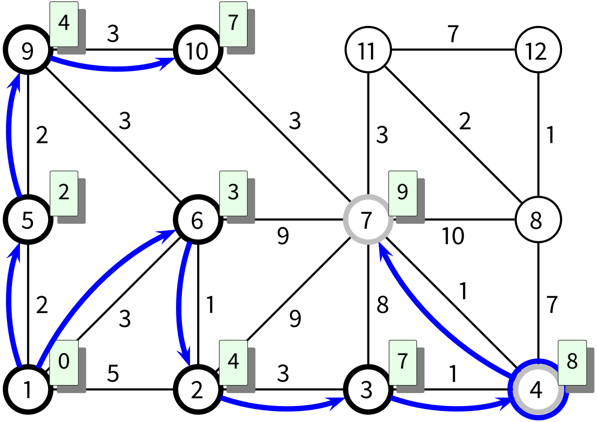
Example



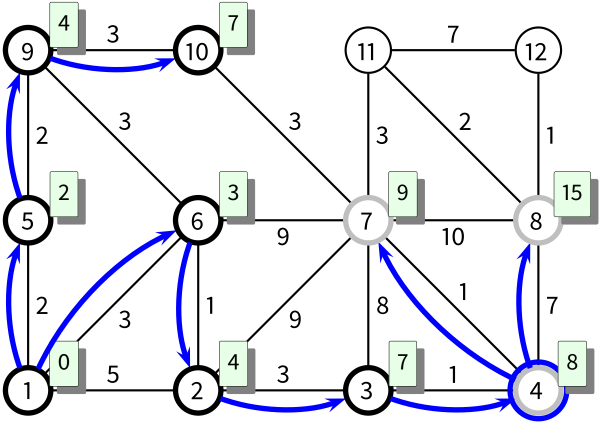
Example



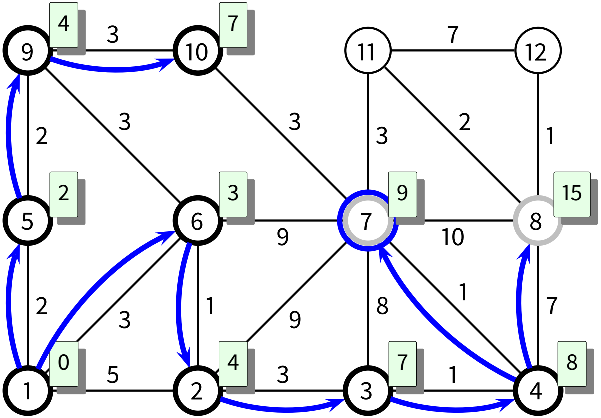
Example



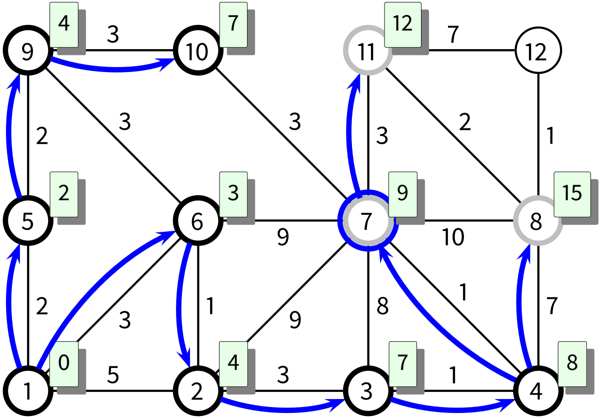
Example



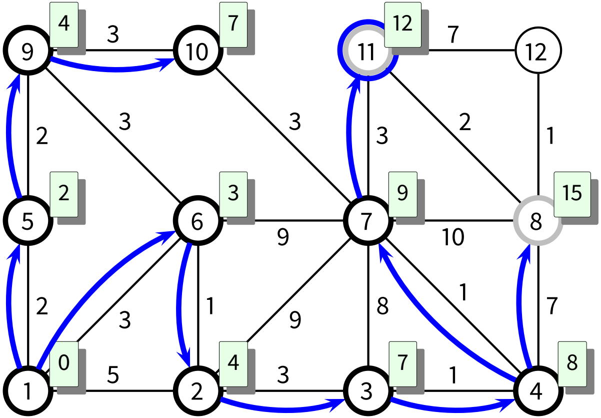
Example



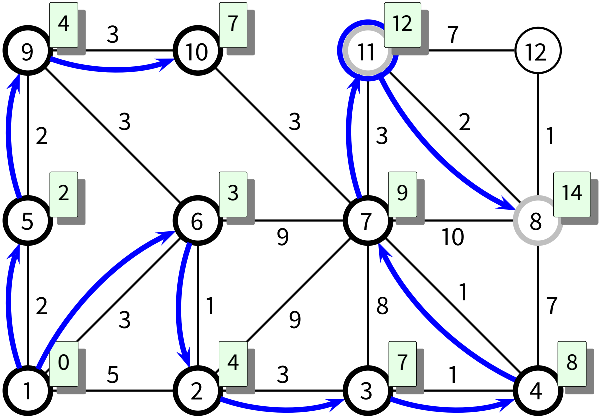
Example



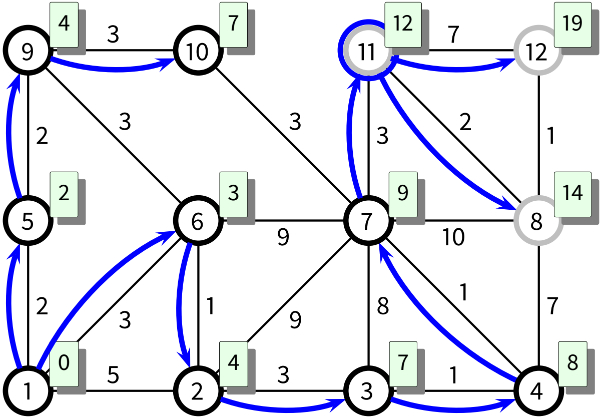
Example



Example

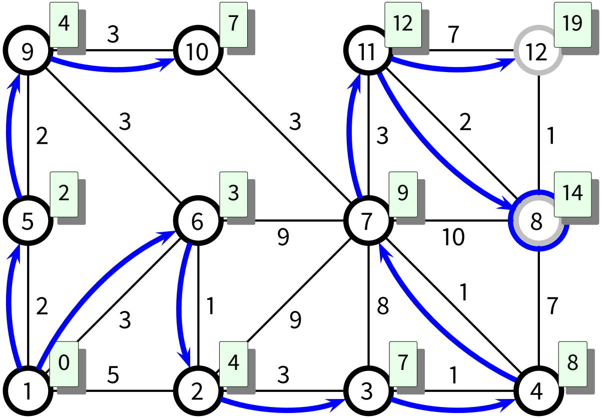


Example



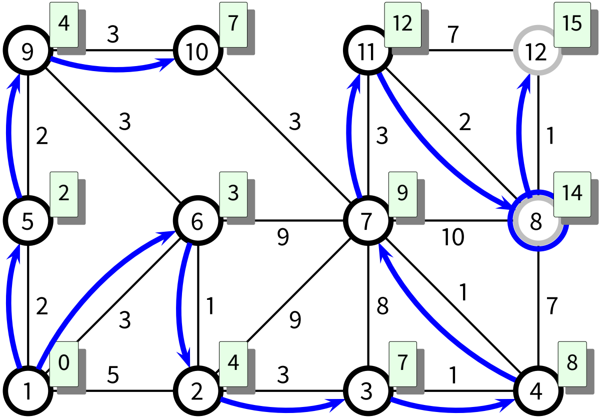
9
4
3
10
7
11
12
7
12
19
2
3
3
2
1
3
5
2
6
3
9
7
9
10
8
14
2
3
1
9
8
1
7
1
0
5
2
4
3
3
7
1
4
8

Example



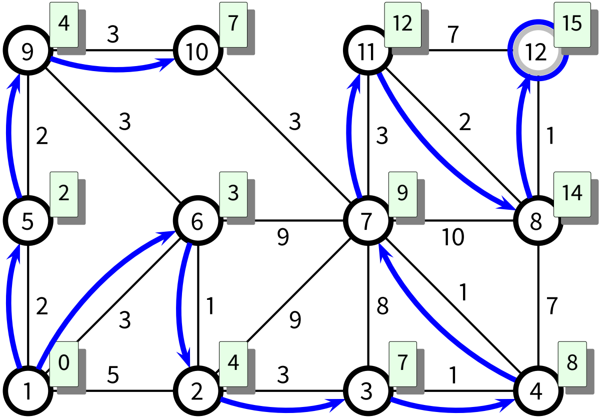
9
4
3
10
7
11
12
7
12
19
2
3
3
3
2
1
5
2
6
3
9
7
9
10
8
14
2
1
3
1
9
8
7
1
0
5
2
4
3
3
7
1
4
8

Example

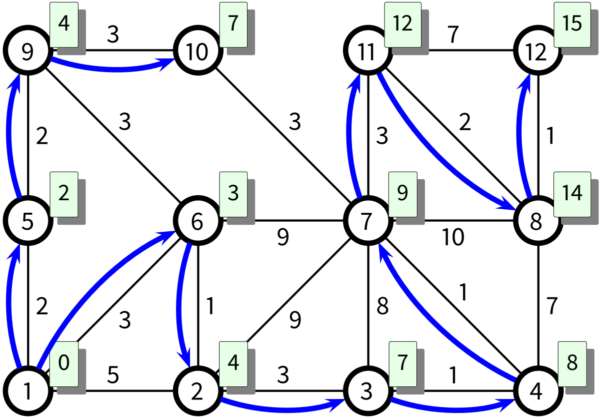


9
4
3
10
7
11
12
7
12
15
2
3
3
3
2
1
5
2
6
3
9
7
9
10
8
14
2
3
1
9
8
1
7
1
0
5
2
4
3
3
7
1
4
8

Example


9
4
3
10
7
11
12
7
12
15
2
3
3
3
2
1
5
2
6
3
9
7
9
10
8
14
2
3
1
9
8
1
7
1
0
5
2
4
3
3
7
1
4
8

Example



Dijkstra's Algorithm: Complexity

```
DIJKSTRA(G, s, w)
1   N = ∅
2   for each vertex v ∈ V(G)
3       D[v] = ∞
4       P[v] = NIL
5   D[s] = 0
6   while N ≠ V(G)
7       find u ∉ N such that D[u] is minimal
8       N = N ∪ {u}
9       for all v ∈ Adj(u) \ N
10          if D[u] + w(u, v) < D[v]
11              D[v] = D[u] + w(u, v)
12              P[v] = u
```

- Use a heap H for storing D
  - Line 7: EXTRACT-MIN from H
  - Line 11: UPDATE-KEY in H
- Time complexity: $O(|E| \log |V|)$

Dijkstra's Algorithm: Correctness

Some facts:

- If there is no path from s to a v, then $D[v] = \infty$ at all time in the algorithm:
  - Observe the following: whenever $D[v] < \infty$, it always corresponds to a path from s to v
- $\delta(s, v) \leq D[v]$ for all v
  - Again, whenever $D[v] < \infty$, it always corresponds to a path from s to v

Loop variant:

At the start of each iteration of the **while** loop, $D[v] = \delta(s, v)$ for each $v \in N$

Dijkstra's Algorithm: Correctness

Loop variant:

At the start of each iteration of the **while** loop, $D[v] = \delta(s, v)$ for each $v \in N$

Proof:

- Initially, $N = \emptyset$ and this is trivially true
- We only need to show that the invariant is true for all the following iterations

Loop variant:

At the start of each iteration of the **while** loop, $D[v] = \delta(s, v)$ for each $v \in N$

Proof:

- Initially, $N = \varnothing$ and this is trivially true
- We only need to show that the invariant is true for all the following iterations
- For contradiction, let u be the first vertex for which $D[u] \neq \delta(s, u)$ when it is added to N
  - We must have $u \neq s$ because s is the first vertex added to N and $\delta(s, s) = D[s] = 0$; we also have that $N \neq \varnothing$ before u is added to N

Loop variant:

At the start of each iteration of the **while** loop, $D[v] = \delta(s, v)$ for each $v \in N$

Proof:

- Initially, $N = \emptyset$ and this is trivially true
- We only need to show that the invariant is true for all the following iterations
- For contradiction, let u be the first vertex for which $D[u] \neq \delta(s, u)$ when it is added to N
  - We must have $u \neq s$ because s is the first vertex added to N and $\delta(s, s) = D[s] = 0$; we also have that $N \neq \emptyset$ before u is added to N
- Since $D[u] \neq \delta(s, u)$, there must be a path from s to u, because otherwise $D[u] = \infty$ for always (previous facts) and $D[u] = \delta(s, u) = \infty$

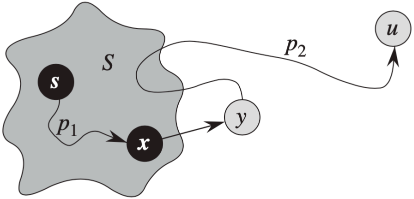
Loop variant:

At the start of each iteration of the **while** loop, $D[v] = \delta(s, v)$ for each $v \in N$

Proof:

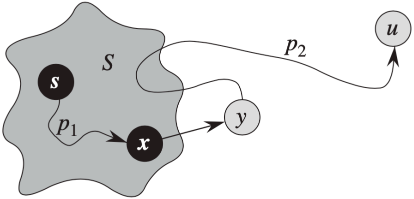
- Initially, $N = \emptyset$ and this is trivially true
- We only need to show that the invariant is true for all the following iterations
- For contradiction, let u be the first vertex for which $D[u] \neq \delta(s, u)$ when it is added to N
  - We must have $u \neq s$ because s is the first vertex added to N and $\delta(s, s) = D[s] = 0$; we also have that $N \neq \emptyset$ before u is added to N
- Since $D[u] \neq \delta(s, u)$, there must be a path from s to u, because otherwise $D[u] = \infty$ for always (previous facts) and $D[u] = \delta(s, u) = \infty$
- Let p be the shortest path from s to u
- Consider the N before adding u: Since the start of p is $s \in N$ and the end of p is $u \notin N$, we can let y be the first vertex along p such that $y \notin N$, and let x be predecessor of y along p ($x \in N$).

Dijkstra's Algorithm: Correctness



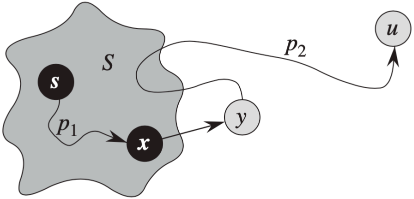
■ (Figure from CLRS)

Dijkstra's Algorithm: Correctness



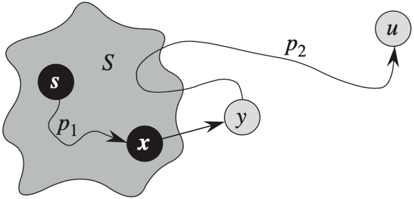
- (Figure from CLRS)
- Because u is the first vertex added to N for which $D[u] \neq \delta(s, u)$, we have $D[x] = \delta(s, x)$ when x was added

Dijkstra's Algorithm: Correctness

- 

(Figure from CLRS)
- Because u is the first vertex added to N for which $D[u] \neq \delta(s, u)$, we have $D[x] = \delta(s, x)$ when x was added
- From the path p, we know that $\delta(s, y) = \delta(s, x) + w(x, y)$. So when x was added to N, $D[y] = \delta(s, y)$ after the update in Line 9-11

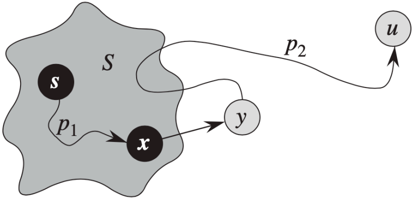
Dijkstra's Algorithm: Correctness

- 

(Figure from CLRS)
- Because u is the first vertex added to N for which $D[u] \neq \delta(s, u)$, we have $D[x] = \delta(s, x)$ when x was added
- From the path p, we know that $\delta(s, y) = \delta(s, x) + w(x, y)$. So when x was added to N, $D[y] = \delta(s, y)$ after the update in Line 9-11
- From the path p, we know that $\delta(s, y) \leq \delta(s, u)$. So

$$D[y] = \delta(s, y) \leq \delta(s, u) \leq D[u]$$

Dijkstra's Algorithm: Correctness

- 

(Figure from CLRS)
- Because u is the first vertex added to N for which $D[u] \neq \delta(s,u)$, we have $D[x] = \delta(s,x)$ when x was added
- From the path p, we know that $\delta(s,y) = \delta(s,x) + w(x,y)$. So when x was added to N, $D[y] = \delta(s,y)$ after the update in Line 9-11
- From the path p, we know that $\delta(s,y) \leq \delta(s,u)$. So

$$D[y] = \delta(s,y) \leq \delta(s,u) \leq D[u]$$

- Because both u and y were not in N when u was chosen in Line 7, we have $D[u] \leq D[y]$. So the above become equalities

$$D[y] = \delta(s,y) = \delta(s,u) = D[u]$$

A contradiction!

Bellman-Ford algorithm

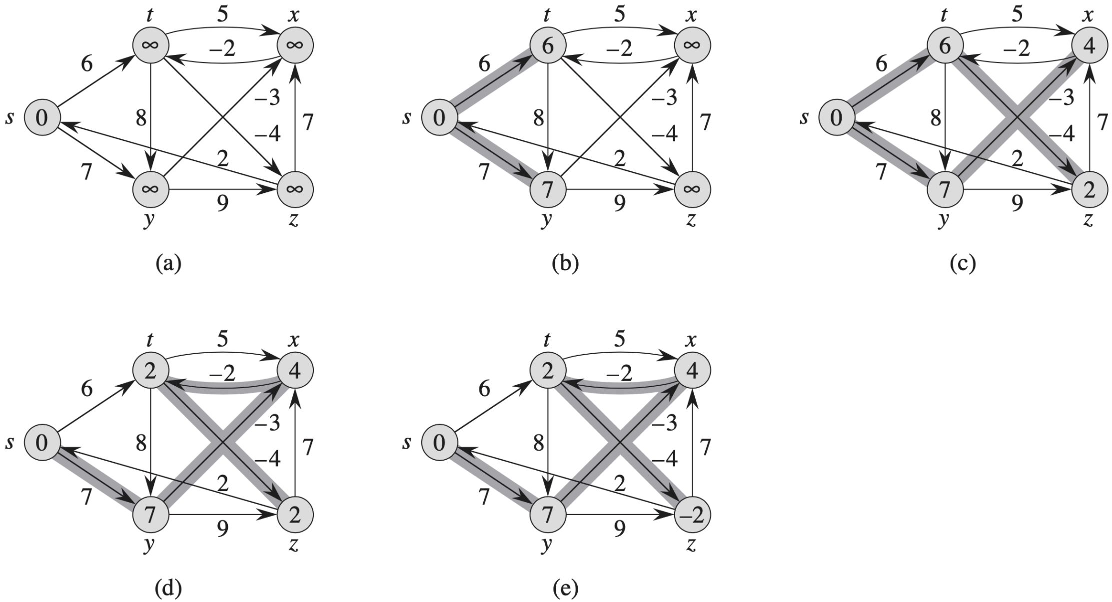
- The ***most general-purpose*** algorithm for computing single-source shortest paths: allows ***negative weights*** on edges, and works for ***any*** graphs
- Returns a ***boolean*** value indicating whether or not there is a negative-weight cycle that is reachable from the source
  - If there is such a cycle, the algorithm indicates that ***no solution*** exists.
  - If there is no such cycle, the algorithm produces the shortest paths and their weights.
- The idea of the algorithm is simple, after the initialization (common to all S-S shortest path algorithms), it has $|V| - 1$ ***rounds***, where each round ***relaxes all the edges***

Bellman-Ford algorithm

```
BELLMAN-FORD(G = (V, E), s, w)
for each vertex v ∈ V
    D[v] = ∞
    P[v] = NIL
D[s] = 0
for i = 1, . . . , |V| − 1
    for each edge (u, v) ∈ E
        RELAX(u, v)
for each edge (u, v) ∈ E
    if D[u] + w(u, v) < D[v]
        return FALSE
return TRUE
```

```
RELAX(u, v)
if D[u] + w(u, v) < D[v]
    D[v] = D[u] + w(u, v)
    P[v] = u
```

Time complexity: $O(|V| \times |E|)$



(Example from CLRS)

Proposition 1

We always have $D[v] \geq \delta(s, v)$ in the algorithm. Furthermore, after $D[v] = \delta(s, v)$, $D[v]$ does not change no matter what relaxations we perform in the algorithm.

Proof:

- We have seen the argument for $D[v] \geq \delta(s, v)$ before, i.e., $D[v]$ always corresponds to the weight of an actual path from s to v (or else $D[v] = \infty$), and so it cannot be less than the optimal one, $\delta(s, v)$.
- For the second part, notice that a relaxation always decreases $D[x]$ for a vertex x. If we already have $D[v] = \delta(s, v)$, then $D[v]$ cannot be further decreased because $D[v] \geq \delta(s, v)$.

Path-relaxation property

Let $p = \langle v_0 = s, v_1, \ldots, v_k \rangle$ be a shortest path from s to v_k. After relaxing the edges in the order $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, we have $D[v_k] = \delta(s, v_k)$. This property holds even if there are relaxations of other edges intermixed with relaxations of the edges on p.

Path-relaxation property

Let $p = \langle v_0 = s, v_1, \ldots, v_k \rangle$ be a shortest path from s to v_k. After relaxing the edges in the order $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, we have $D[v_k] = \delta(s, v_k)$. This property holds even if there are relaxations of other edges intermixed with relaxations of the edges on p.

Proof:

- We prove by induction that, for each i, before relaxing (v_i, v_{i+1}), we have $D[v_i] = \delta(s, v_i)$, and after the relaxation, we have $D[v_{i+1}] = \delta(s, v_{i+1})$.

Path-relaxation property

Let $p = \langle v_0 = s, v_1, \ldots, v_k \rangle$ be a shortest path from s to v_k. After relaxing the edges in the order $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, we have $D[v_k] = \delta(s, v_k)$. This property holds even if there are relaxations of other edges intermixed with relaxations of the edges on p.

Proof:

- We prove by induction that, for each i, before relaxing (v_i, v_{i+1}), we have $D[v_i] = \delta(s, v_i)$, and after the relaxation, we have $D[v_{i+1}] = \delta(s, v_{i+1})$.
- Unlike other proofs by inductions, we prove the **induction step** first.
- Assume for $i - 1$, the claim is true. We have that after relaxing (v_{i-1}, v_i), $D[v_i] = \delta(s, v_i)$ (inductive assumption).
- After that, no matter what relaxations we perform, we always have $D[v_i] = \delta(s, v_i)$ (Proposition 1).

Path-relaxation property

Let $p = \langle v_0 = s, v_1, \ldots, v_k \rangle$ be a shortest path from s to v_k. After relaxing the edges in the order $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, we have $D[v_k] = \delta(s, v_k)$. This property holds even if there are relaxations of other edges intermixed with relaxations of the edges on p.

Proof:

- We prove by induction that, for each i, before relaxing (v_i, v_{i+1}), we have $D[v_i] = \delta(s, v_i)$, and after the relaxation, we have $D[v_{i+1}] = \delta(s, v_{i+1})$.
- Unlike other proofs by inductions, we prove the **induction step** first.
- Assume for $i - 1$, the claim is true. We have that after relaxing (v_{i-1}, v_i), $D[v_i] = \delta(s, v_i)$ (inductive assumption).
- After that, no matter what relaxations we perform, we always have $D[v_i] = \delta(s, v_i)$ (Proposition 1).
- Then, before relaxing (v_i, v_{i+1}), we have $D[v_i] = \delta(s, v_i)$ (first part is true)

Path-relaxation property

Let $p = \langle v_0 = s, v_1, \ldots, v_k \rangle$ be a shortest path from s to v_k. After relaxing the edges in the order $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, we have $D[v_k] = \delta(s, v_k)$. This property holds even if there are relaxations of other edges intermixed with relaxations of the edges on p.

Proof:

- We prove by induction that, for each i, before relaxing (v_i, v_{i+1}), we have $D[v_i] = \delta(s, v_i)$, and after the relaxation, we have $D[v_{i+1}] = \delta(s, v_{i+1})$.
- Unlike other proofs by inductions, we prove the **induction step** first.
- Assume for $i - 1$, the claim is true. We have that after relaxing (v_{i-1}, v_i), $D[v_i] = \delta(s, v_i)$ (inductive assumption).
- After that, no matter what relaxations we perform, we always have $D[v_i] = \delta(s, v_i)$ (Proposition 1).
- Then, before relaxing (v_i, v_{i+1}), we have $D[v_i] = \delta(s, v_i)$ (first part is true)
- When relaxing (v_i, v_{i+1}), if $D[v_{i+1}] = \delta(s, v_{i+1})$ already, then we have nothing to prove.

Path-relaxation property

Let $p = \langle v_0 = s, v_1, \ldots, v_k \rangle$ be a shortest path from s to v_k. After relaxing the edges in the order $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, we have $D[v_k] = \delta(s, v_k)$. This property holds even if there are relaxations of other edges intermixed with relaxations of the edges on p.

Proof:

- We prove by induction that, for each i, before relaxing (v_i, v_{i+1}), we have $D[v_i] = \delta(s, v_i)$, and after the relaxation, we have $D[v_{i+1}] = \delta(s, v_{i+1})$.
- Unlike other proofs by inductions, we prove the **induction step** first.
- Assume for $i - 1$, the claim is true. We have that after relaxing (v_{i-1}, v_i), $D[v_i] = \delta(s, v_i)$ (inductive assumption).
- After that, no matter what relaxations we perform, we always have $D[v_i] = \delta(s, v_i)$ (Proposition 1).
- Then, before relaxing (v_i, v_{i+1}), we have $D[v_i] = \delta(s, v_i)$ (first part is true)
- When relaxing (v_i, v_{i+1}), if $D[v_{i+1}] = \delta(s, v_{i+1})$ already, then we have nothing to prove.
- If $D[v_{i+1}] > \delta(s, v_{i+1})$, then $D[v_{i+1}] > \delta(s, v_{i+1}) = \delta(s, v_i) + w(v_i, v_{i+1}) = D[v_i] + w(v_i, v_{i+1})$, so $D[v_{i+1}]$ must be updated to $D[v_i] + w(v_i, v_{i+1}) = \delta(s, v_{i+1})$ by the relaxation.

Path-relaxation property

Let $p = \langle v_0 = s, v_1, \ldots, v_k \rangle$ be a shortest path from s to v_k. After relaxing the edges in the order $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, we have $d[v_k] = \delta(s, v_k)$. This property holds even if there are relaxations of other edges intermixed with relaxations of the edges on p.

Proof:

- For the **base case**, we first have that $D[v_0] = 0$ always for $v_0 = s$.
- The verification for the rest of the base case is the same as that for the induction step.

Correctness of Bellman-Ford

The two lemmas combined indicate that Bellman-Ford is correct:

Lemma 1

If G contains no negative-weight cycles reachable from s, then the algorithm returns TRUE, and we have $D[v] = \delta(s, v)$ for every vertex $v \in V$.

Lemma 2

If G contains a negative-weight cycle reachable from s, then the algorithm returns FALSE.

Lemma 1

If G contains no negative-weight cycles reachable from s, then the algorithm returns TRUE, and we have $D[v] = \delta(s, v)$ for every vertex $v \in V$.

Proof:

Lemma 1

If G contains no negative-weight cycles reachable from s, then the algorithm returns TRUE, and we have $D[v] = \delta(s, v)$ for every vertex $v \in V$.

Proof:

- If a vertex v is reachable from s, then let $p = \langle u_0 = s, u_1, \ldots, u_k = v \rangle$ be a shortest path from s to v.
- Since p contains no cycle, then the number of vertices on the path is $k + 1 \leq |V|$ (i.e, $k \leq |V| - 1$).
- So we have
  - The first round ($i = 1$) relaxes (u_0, u_1)
  - The second round ($i = 2$) relaxes (u_1, u_2)
  - ...
  - The k-th round ($i = k \leq |V| - 1$) relaxes (u_{k-1}, u_k)

  After the relaxations, $D[v] = \delta(s, v)$ (by Path-relaxation property)

Lemma 1

If G contains no negative-weight cycles reachable from s, then the algorithm returns TRUE, and we have $D[v] = \delta(s, v)$ for every vertex $v \in V$.

Proof:

- If a vertex v is reachable from s, then let $p = \langle u_0 = s, u_1, \ldots, u_k = v \rangle$ be a shortest path from s to v.
- Since p contains no cycle, then the number of vertices on the path is $k + 1 \leq |V|$ (i.e, $k \leq |V| - 1$).
- So we have
    - The first round ($i = 1$) relaxes (u_0, u_1)
    - The second round ($i = 2$) relaxes (u_1, u_2)
    - ...
    - The k-th round ($i = k \leq |V| - 1$) relaxes (u_{k-1}, u_k)

  After the relaxations, $D[v] = \delta(s, v)$ (by Path-relaxation property)
- If a vertex v is not reachable from s, then we have that $D[v] = \infty = \delta(s, v)$ always.

Lemma 1

If G contains no negative-weight cycles reachable from s, then the algorithm returns TRUE, and we have $D[v] = \delta(s, v)$ for every vertex $v \in V$.

Proof:

- We still need to prove that the algorithm returns TRUE.

Lemma 1

If G contains no negative-weight cycles reachable from s, then the algorithm returns TRUE, and we have $D[v] = \delta(s, v)$ for every vertex $v \in V$.

Proof:

- We still need to prove that the algorithm returns TRUE.
- For each edge (u, v), we have

$$\begin{aligned} D[v] &= \delta(s, v) && (1) \\ &\leq \delta(s, u) + w(u, v) && (2) \\ &= D[u] + w(u, v) && (3) \end{aligned}$$

(1)-(2) follows from 'triangle inequality'

Lemma 2

If G contains a negative-weight cycle reachable from s, then the algorithm returns FALSE.

Proof:

Lemma 2

If G contains a negative-weight cycle reachable from s, then the algorithm returns FALSE.

Proof:

- Let $c = \langle v_0, v_1, \ldots, v_k \rangle$ be a negative-weight cycle reachable from s where $v_0 = v_k$.
- We have $\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0$.

Lemma 2

If G contains a negative-weight cycle reachable from s, then the algorithm returns FALSE.

Proof:

- Let $c = \langle v_0, v_1, \ldots, v_k \rangle$ be a negative-weight cycle reachable from s where $v_0 = v_k$.
- We have $\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0$.
- For contradiction, assume the algorithm returns TRUE.
- Then, $D[v_i] \leq D[v_{i-1}] + w(v_{i-1}, v_i)$, for $i = 1, \ldots, k$

Lemma 2

If G contains a negative-weight cycle reachable from s, then the algorithm returns FALSE.

Proof:

- Let $c = \langle v_0, v_1, \ldots, v_k \rangle$ be a negative-weight cycle reachable from s where $v_0 = v_k$.
- We have $\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0$.
- For contradiction, assume the algorithm returns TRUE.
- Then, $D[v_i] \leq D[v_{i-1}] + w(v_{i-1}, v_i)$, for $i = 1, \ldots, k$
- Summing all the above inequalities:

$$\begin{aligned}
\sum_{i=1}^{k} D[v_i] &\leq \sum_{i=1}^{k} (D[v_{i-1}] + w(v_{i-1}, v_i)) \\
&= \sum_{i=1}^{k} D[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i)
\end{aligned}$$

Lemma 2

If G contains a negative-weight cycle reachable from s, then the algorithm returns FALSE.

Proof:

- Let $c = \langle v_0, v_1, \ldots, v_k \rangle$ be a negative-weight cycle reachable from s where $v_0 = v_k$.
- We have $\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0$.
- For contradiction, assume the algorithm returns TRUE.
- Then, $D[v_i] \leq D[v_{i-1}] + w(v_{i-1}, v_i)$, for $i = 1, \ldots, k$
- Summing all the above inequalities:

$$\begin{aligned}\sum_{i=1}^{k} D[v_i] &\leq \sum_{i=1}^{k} (D[v_{i-1}] + w(v_{i-1}, v_i)) \\ &= \sum_{i=1}^{k} D[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i)\end{aligned}$$

- Since $v_0 = v_k$, we have $\sum_{i=1}^{k} D[v_i] = \sum_{i=1}^{k} D[v_{i-1}]$

Lemma 2

If G contains a negative-weight cycle reachable from s, then the algorithm returns FALSE.

Proof:

- Let $c = \langle v_0, v_1, \ldots, v_k \rangle$ be a negative-weight cycle reachable from s where $v_0 = v_k$.
- We have $\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0$.
- For contradiction, assume the algorithm returns TRUE.
- Then, $D[v_i] \leq D[v_{i-1}] + w(v_{i-1}, v_i)$, for $i = 1, \ldots, k$
- Summing all the above inequalities:

$$\begin{aligned}\sum_{i=1}^{k} D[v_i] &\leq \sum_{i=1}^{k} (D[v_{i-1}] + w(v_{i-1}, v_i)) \\ &= \sum_{i=1}^{k} D[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i)\end{aligned}$$

- Since $v_0 = v_k$, we have $\sum_{i=1}^{k} D[v_i] = \sum_{i=1}^{k} D[v_{i-1}]$
- So the inequality becomes $0 \leq \sum_{i=1}^{k} w(v_{i-1}, v_i)$ (***contradiction***)

Single-source shortest paths in DAG

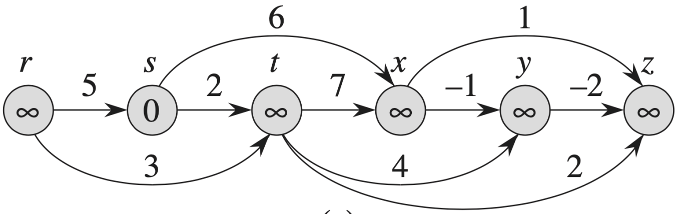
- Assumes the graph is a DAG (directed acyclic graph)
- Edges can have *negative* weights
  - Since we are dealing with DAG, no (negative-weight) cycles can exist
- Finding the shortest-path distance for vertices based on the ***order of topological sort***

Single-source shortest paths in DAG

```
DAG-SHORTEST-PATHS(G, s, w)
topologically sort vertices in G
for each vertex v ∈ V(G)
        D[v] = ∞
        P[v] = NIL
D[s] = 0
for each vertex u ∈ G, in topologically sorted order
        for all v ∈ Adj(u)
                RELAX(u, v)
```

Time complexity: $O(|V| + |E|)$

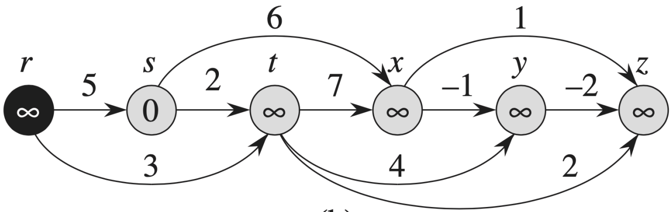
Single-source shortest paths in DAG



(a)

(Example from CLRS)

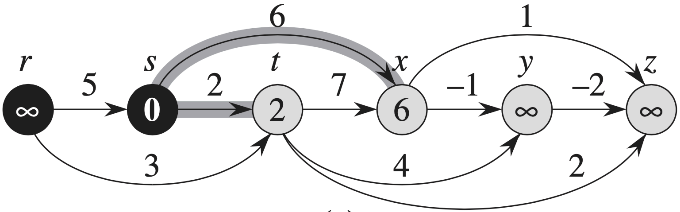
Single-source shortest paths in DAG



(b)

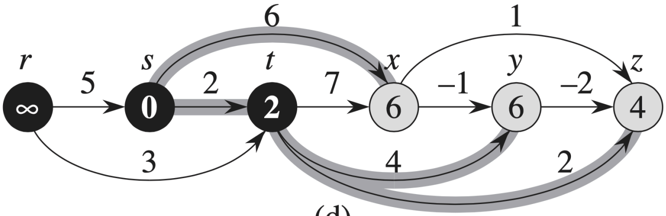
(Example from CLRS)

Single-source shortest paths in DAG



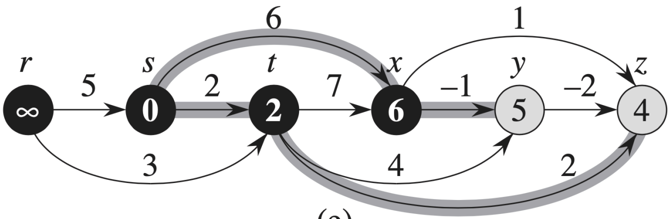
(c)

(Example from CLRS)



(d)

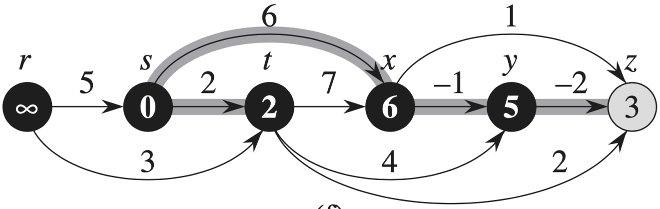
(Example from CLRS)



(e)

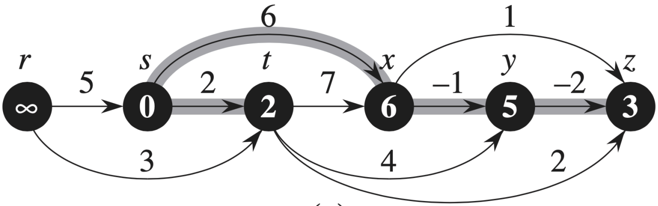
(Example from CLRS)

Single-source shortest paths in DAG



(f)

(Example from CLRS)



(g)

(Example from CLRS)

Proof of correctness

- Hint: The single-source shortest paths algorithm for DAG can be viewed as a ‘smarter’ way of doing Bellman-Ford, and therefore you can adjust the justification for Bellman-Ford to show the correctness of the DAG-algorithm.