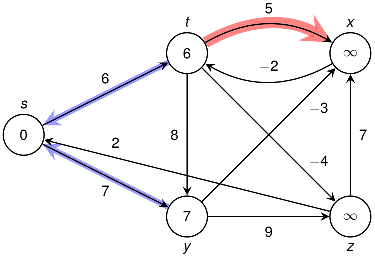


6.4: Single-Source Shortest Paths

Frank Stajano Thomas Sauerwald

Lent 2016

UNIVERSITY OF CAMBRIDGE

Outline

Dijkstra's Algorithm

Historical Remarks

Source: Wikipedia

- Dutch computer scientist
- developed Dijkstra's shortest path algorithm in 1956 (and published in 1959)
- many more fundamental contributions to computer science and engineering
- Turing Award (1972)

Edsger Wybe Dijkstra (1930-2002)

Some Quotes

"It is practically impossible to teach good programming to students that have had a prior exposure to BASIC: as potential programmers they are mentally mutilated beyond hope of regeneration."

"If you want more effective programmers, you will discover that they should not waste their time debugging, they should not introduce the bugs to start with."

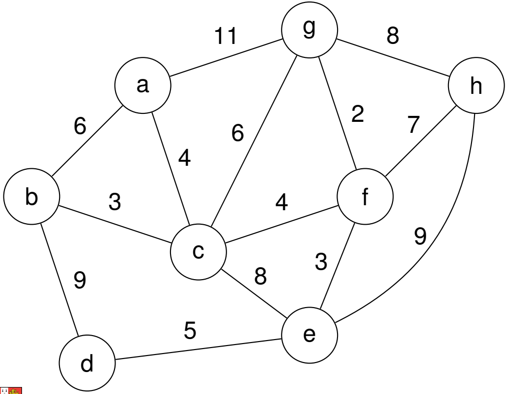
"FORTRAN's tragic fate has been its wide acceptance, mentally chaining thousands and thousands of programmers to our past mistakes."

"Programming is one of the most difficult branches of applied mathematics; the poorer mathematicians had better remain pure mathematicians."

Recap: Prim's Algorithm

Basic Strategy

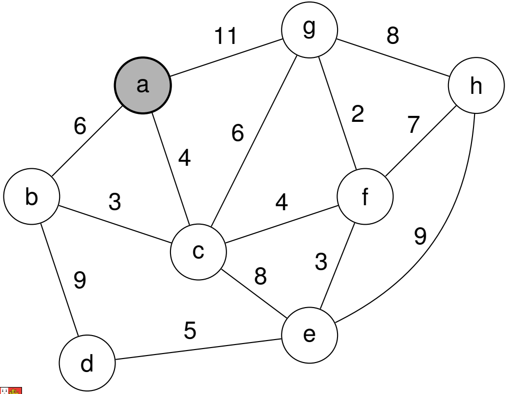
- Start growing a tree from a designated root vertex



Recap: Prim's Algorithm

Basic Strategy

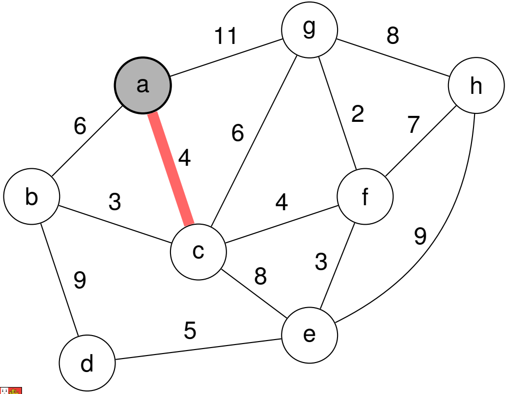
- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to *A* that does not yield cycle



Recap: Prim's Algorithm

Basic Strategy

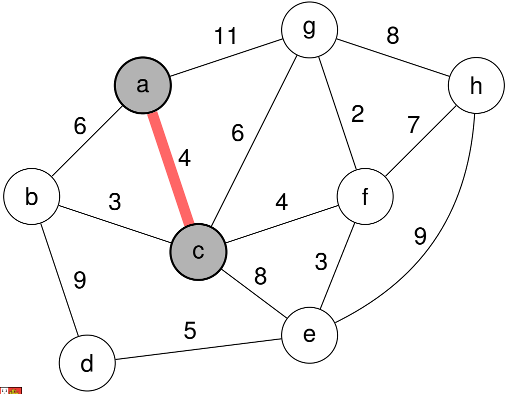
- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to A that does not yield cycle



Recap: Prim's Algorithm

Basic Strategy

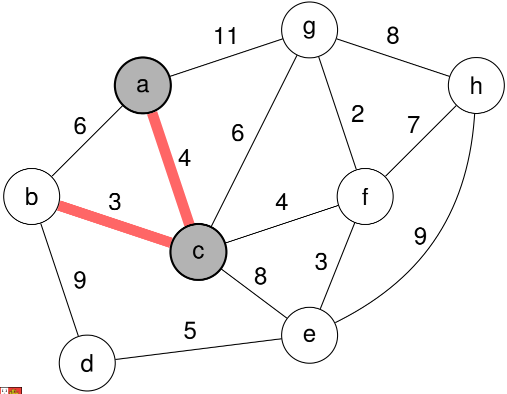
- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to A that does not yield cycle



Recap: Prim's Algorithm

Basic Strategy

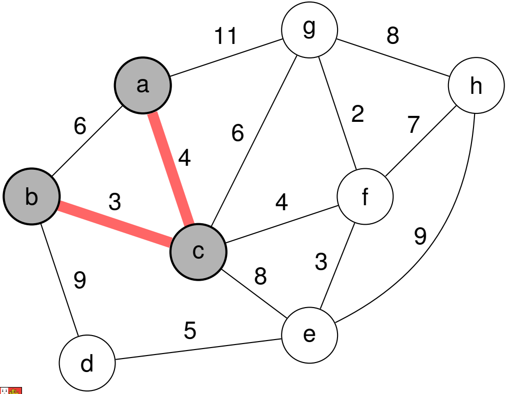
- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to A that does not yield cycle



Recap: Prim's Algorithm

Basic Strategy

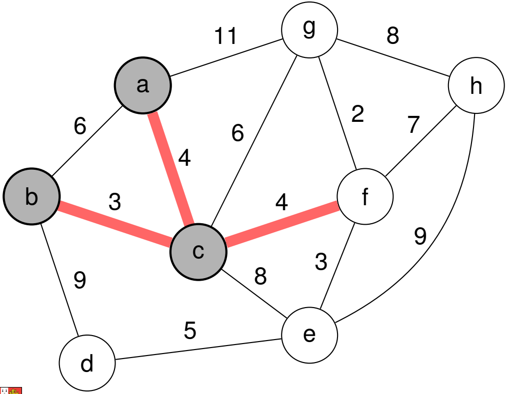
- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to A that does not yield cycle



Recap: Prim's Algorithm

Basic Strategy

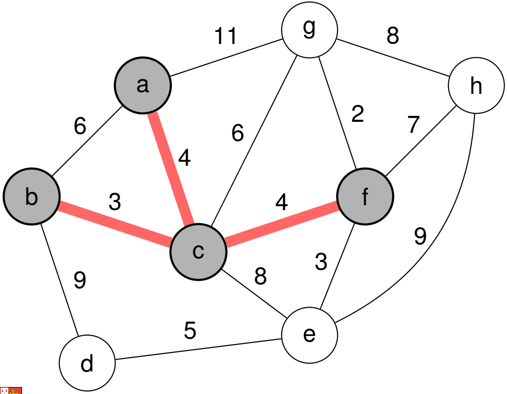
- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to A that does not yield cycle



Recap: Prim's Algorithm

Basic Strategy

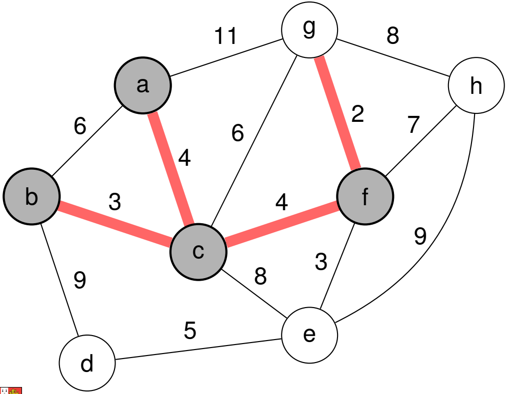
- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to *A* that does not yield cycle



Recap: Prim's Algorithm

Basic Strategy

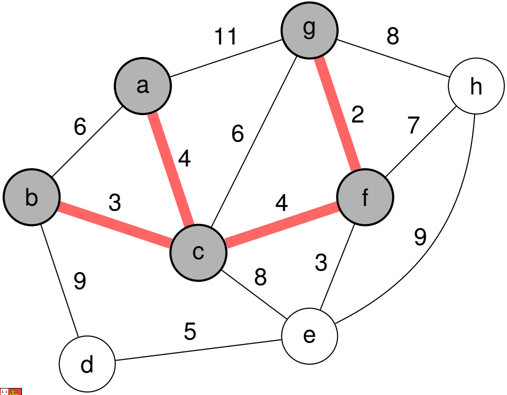
- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to A that does not yield cycle



Recap: Prim's Algorithm

Basic Strategy

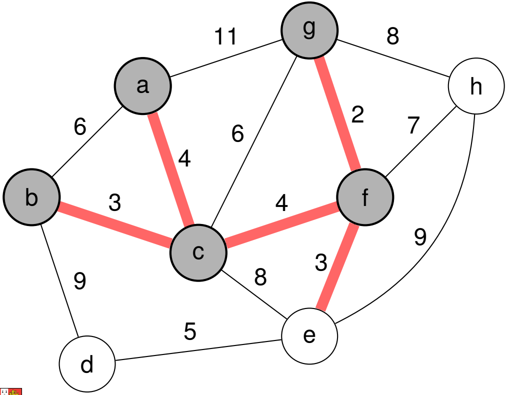
- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to A that does not yield cycle



Recap: Prim's Algorithm

Basic Strategy

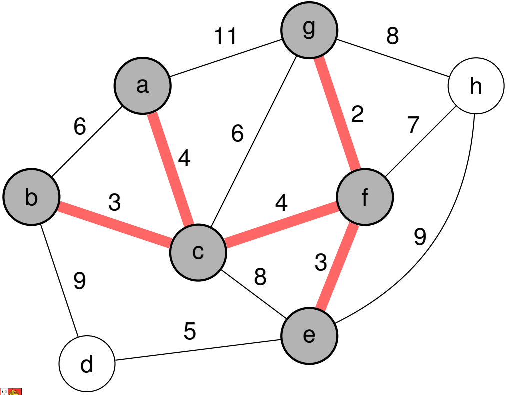
- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to A that does not yield cycle



Recap: Prim's Algorithm

Basic Strategy

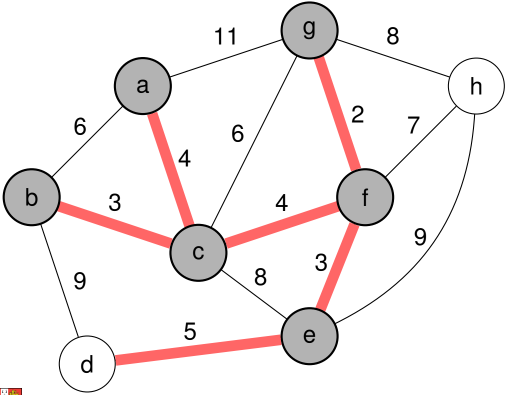
- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to A that does not yield cycle



Recap: Prim's Algorithm

Basic Strategy

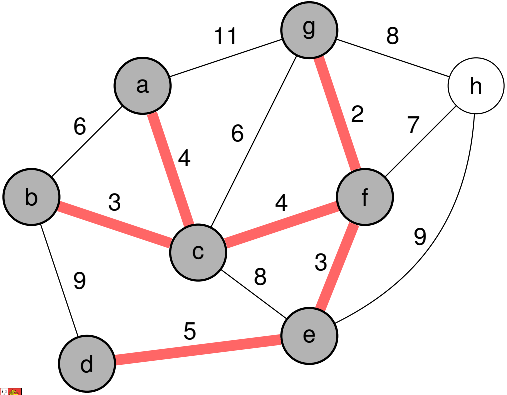
- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to A that does not yield cycle



Recap: Prim's Algorithm

Basic Strategy

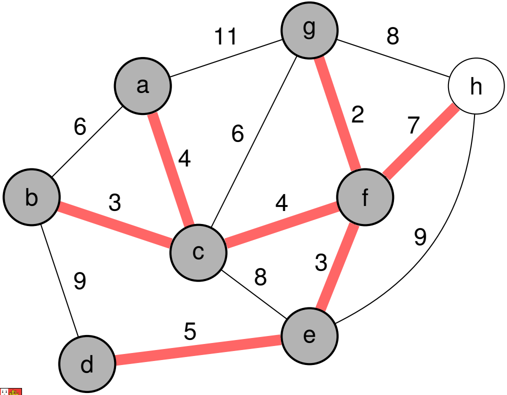
- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to *A* that does not yield cycle



Recap: Prim's Algorithm

Basic Strategy

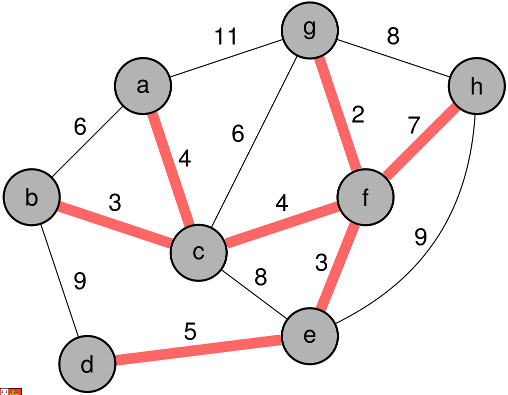
- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to A that does not yield cycle



Recap: Prim's Algorithm

Basic Strategy

- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to *A* that does not yield cycle

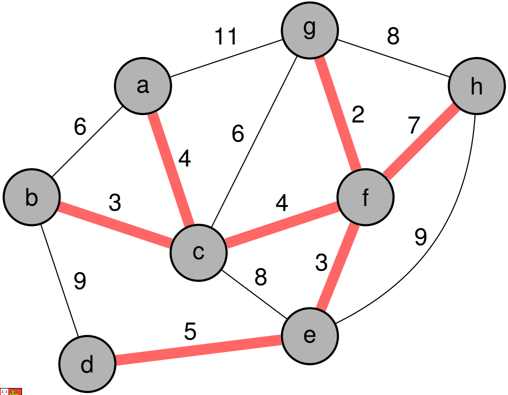


Recap: Prim's Algorithm

Basic Strategy

- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to A that does not yield cycle

Implementation will be based on vertices!

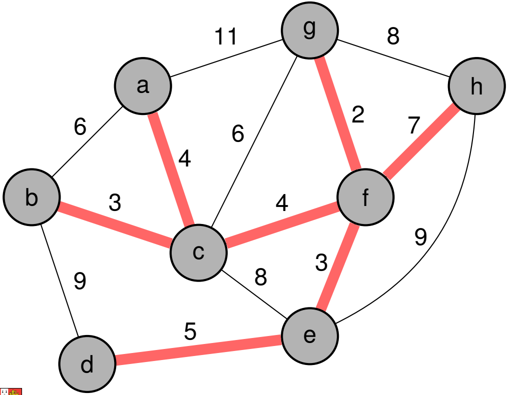


Recap: Prim's Algorithm

Basic Strategy

- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to A that does not yield cycle

Assign every vertex not in A a key which is at all stages equal to the smallest weight of an edge connecting to A



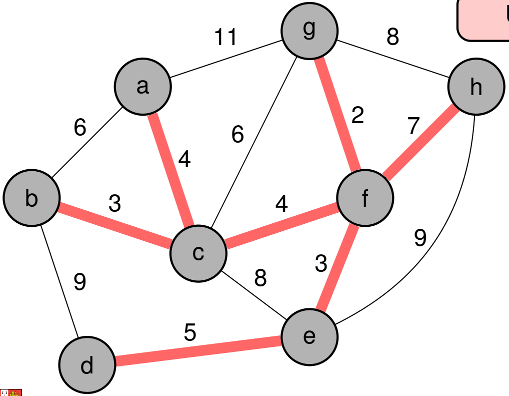
Recap: Prim's Algorithm

Basic Strategy

- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to A that does not yield cycle

Assign every vertex not in A a key which is at all stages equal to the smallest weight of an edge connecting to A

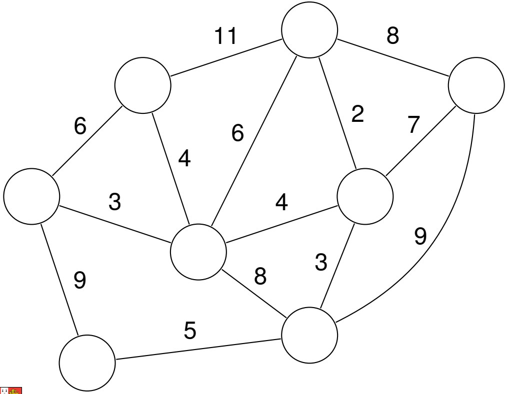
Use a Priority Queue!



Recap: Prim's Algorithm

Implementation

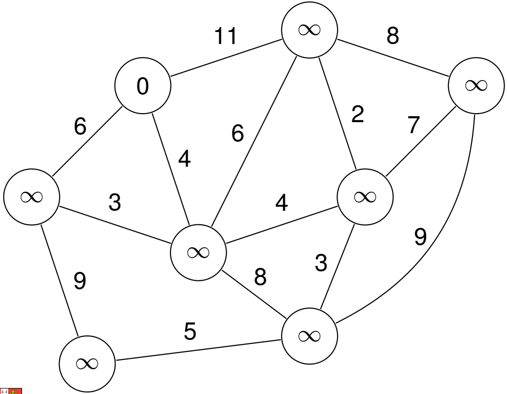
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Recap: Prim's Algorithm

Implementation

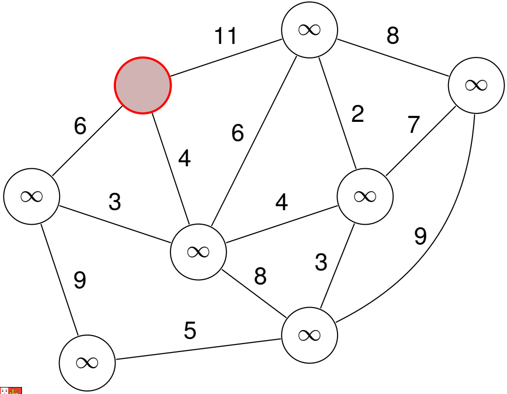
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Recap: Prim's Algorithm

Implementation

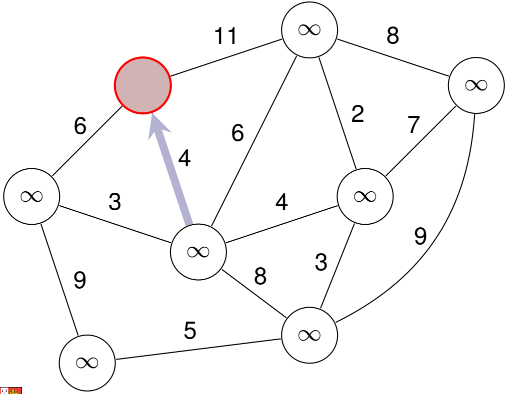
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Recap: Prim's Algorithm

Implementation

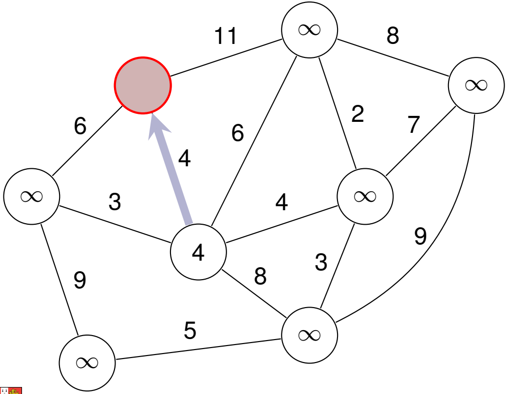
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Recap: Prim's Algorithm

Implementation

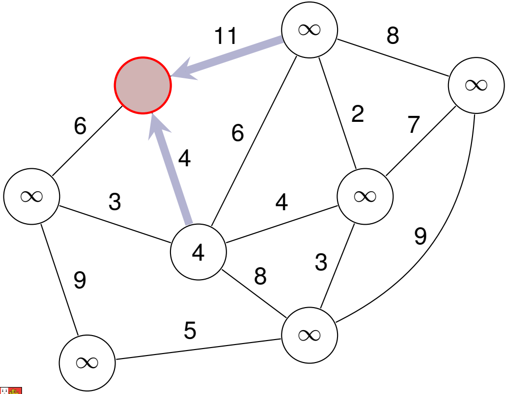
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Recap: Prim's Algorithm

Implementation

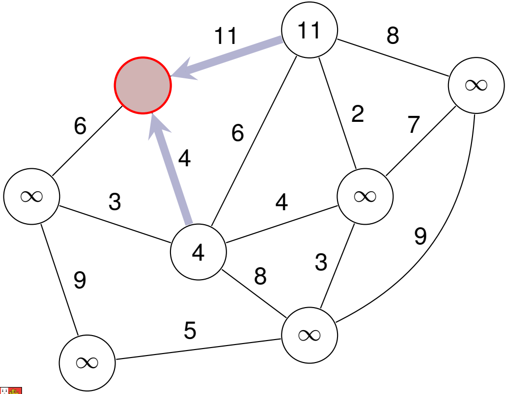
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Recap: Prim's Algorithm

Implementation

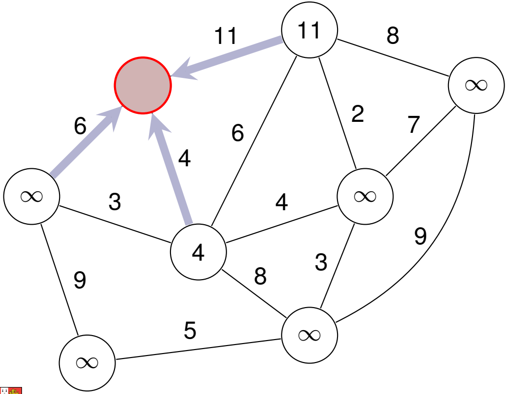
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Recap: Prim's Algorithm

Implementation

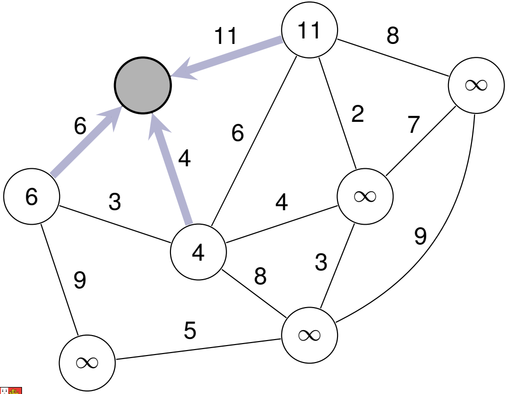
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Recap: Prim's Algorithm

Implementation

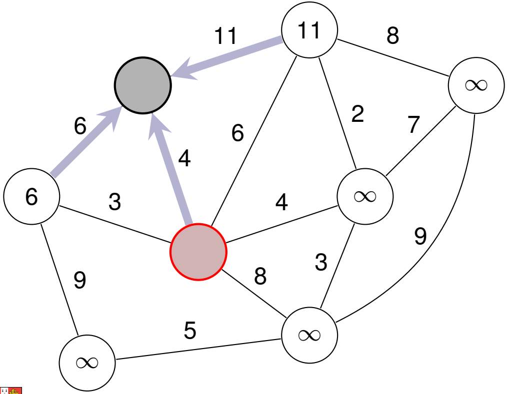
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Recap: Prim's Algorithm

Implementation

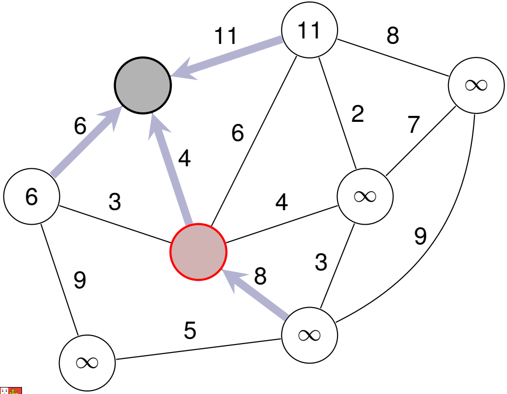
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Recap: Prim's Algorithm

Implementation

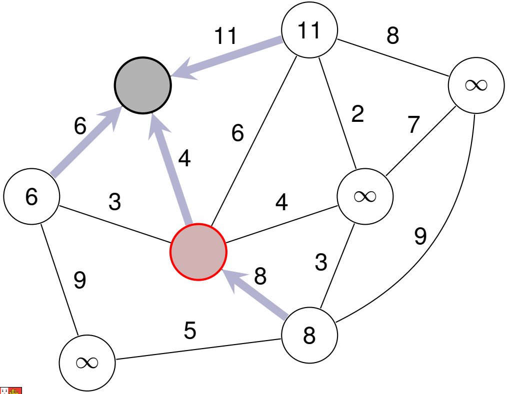
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Recap: Prim's Algorithm

Implementation

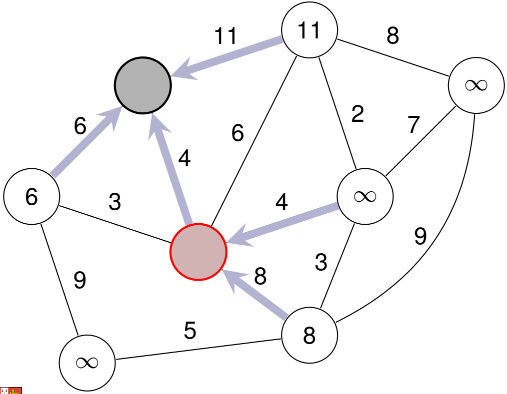
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Recap: Prim's Algorithm

Implementation

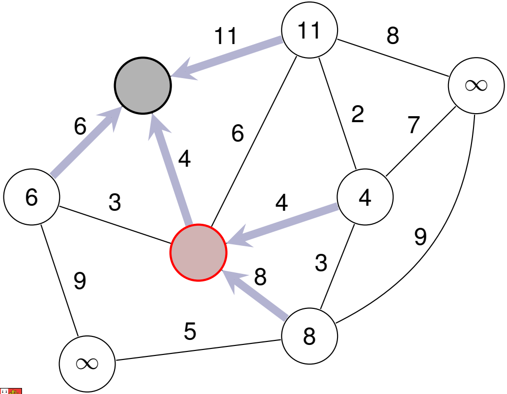
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Recap: Prim's Algorithm

Implementation

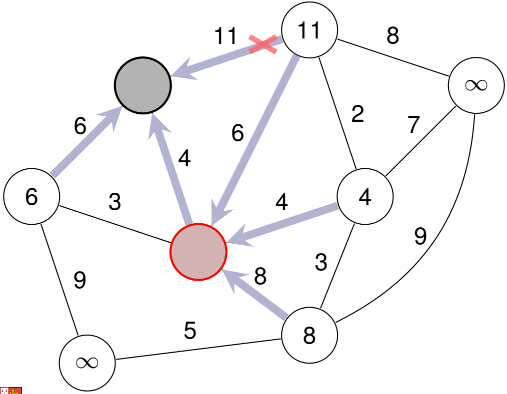
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Recap: Prim's Algorithm

Implementation

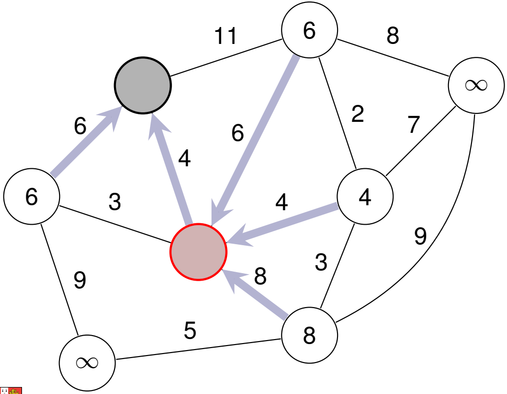
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Recap: Prim's Algorithm

Implementation

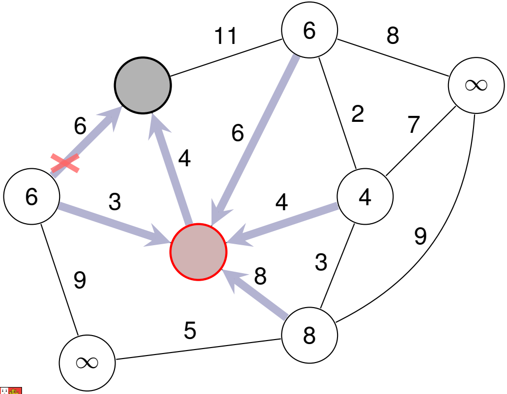
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Recap: Prim's Algorithm

Implementation

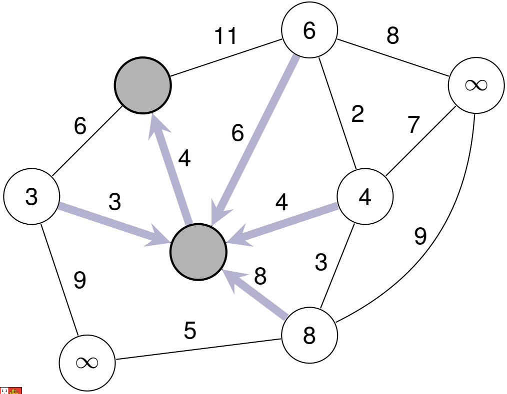
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Recap: Prim's Algorithm

Implementation

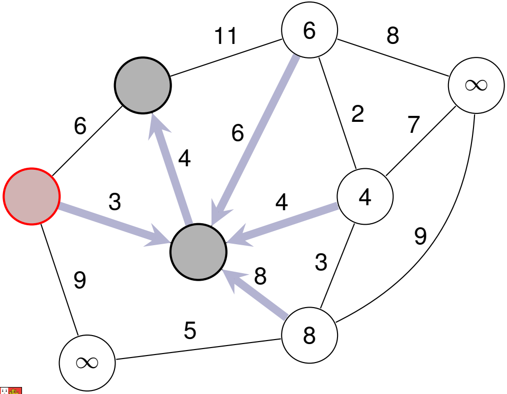
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Recap: Prim's Algorithm

Implementation

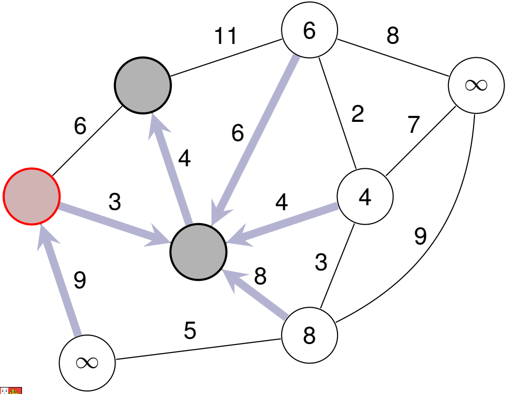
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Recap: Prim's Algorithm

Implementation

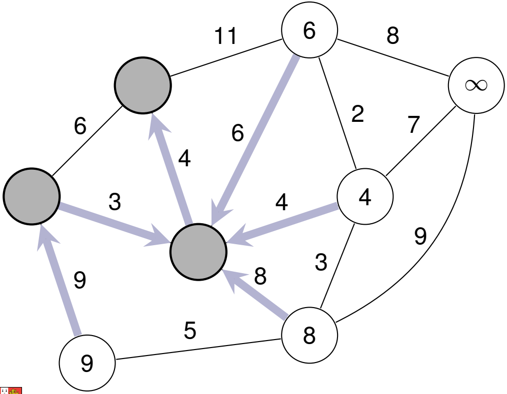
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Recap: Prim's Algorithm

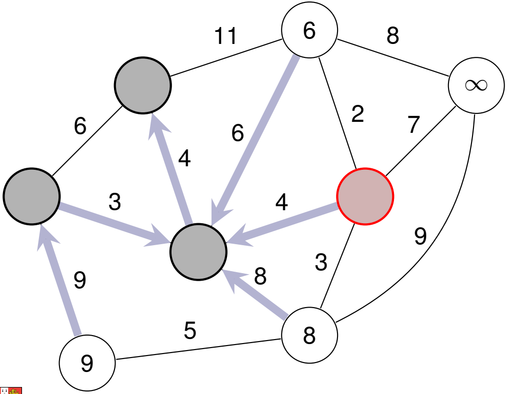
Implementation

- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Implementation

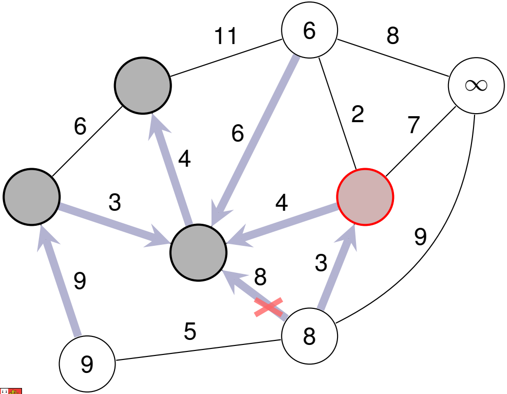
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Recap: Prim's Algorithm

Implementation

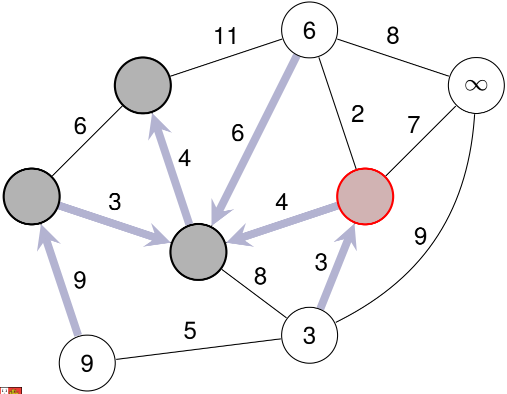
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Recap: Prim's Algorithm

Implementation

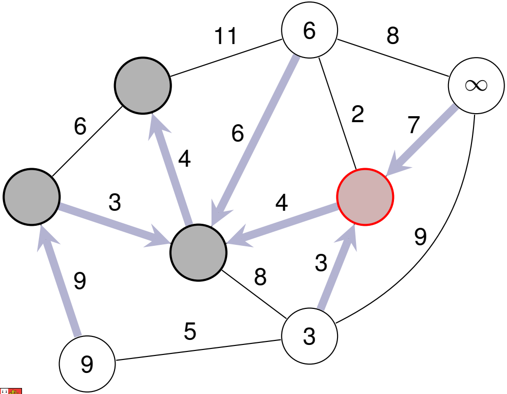
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Recap: Prim's Algorithm

Implementation

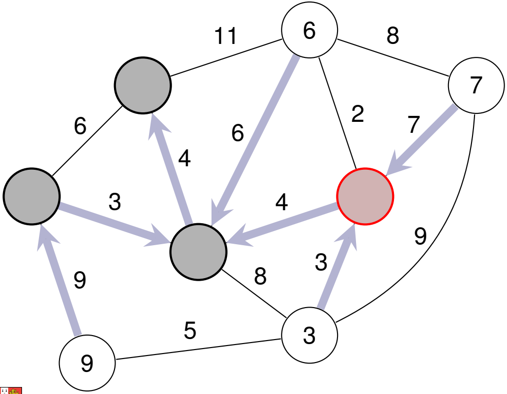
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Recap: Prim's Algorithm

Implementation

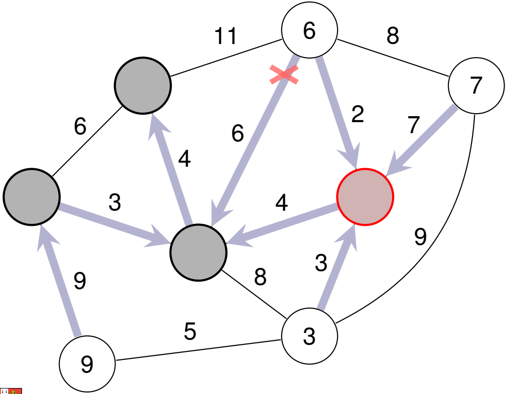
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Recap: Prim's Algorithm

Implementation

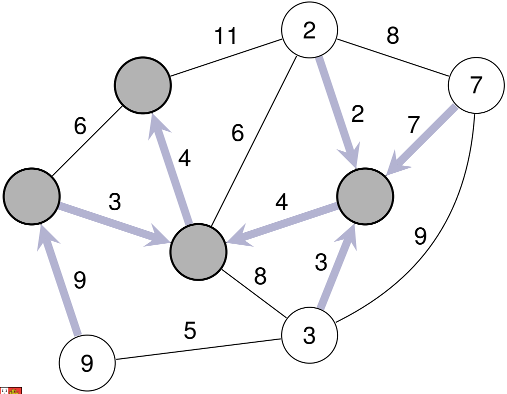
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Recap: Prim's Algorithm

Implementation

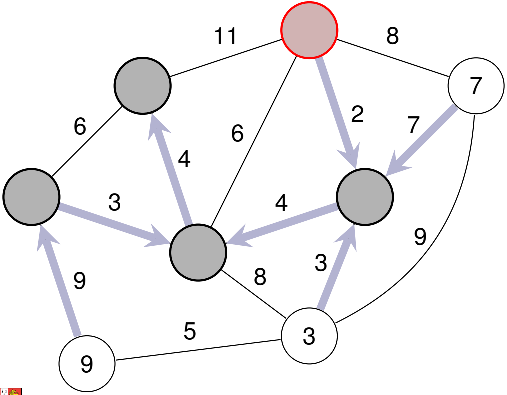
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Recap: Prim's Algorithm

Implementation

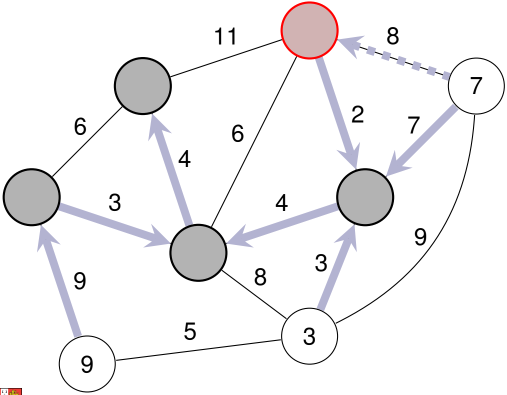
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Recap: Prim's Algorithm

Implementation

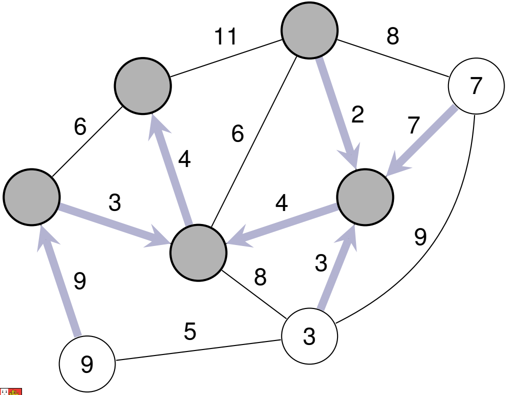
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Recap: Prim's Algorithm

Implementation

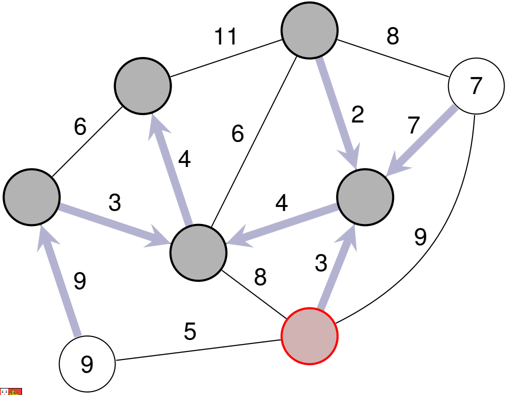
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Recap: Prim's Algorithm

Implementation

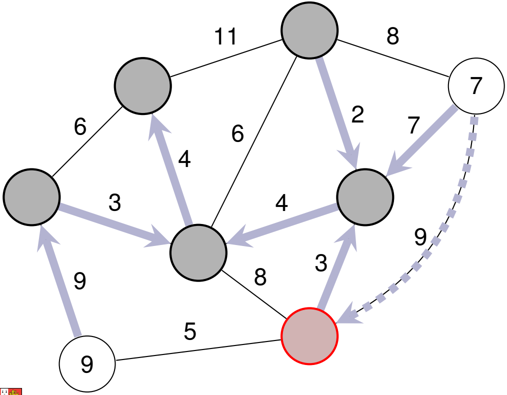
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Recap: Prim's Algorithm

Implementation

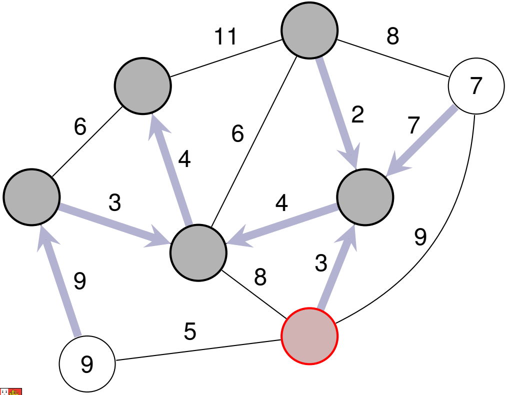
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Recap: Prim's Algorithm

Implementation

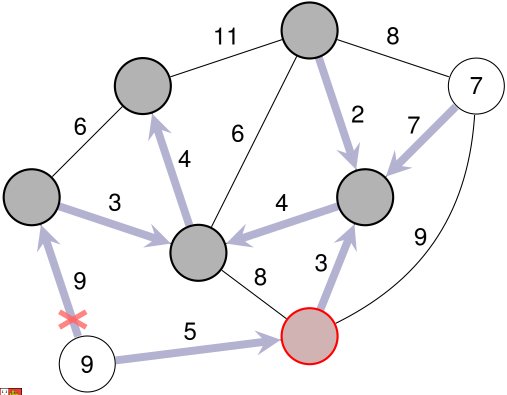
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Recap: Prim's Algorithm

Implementation

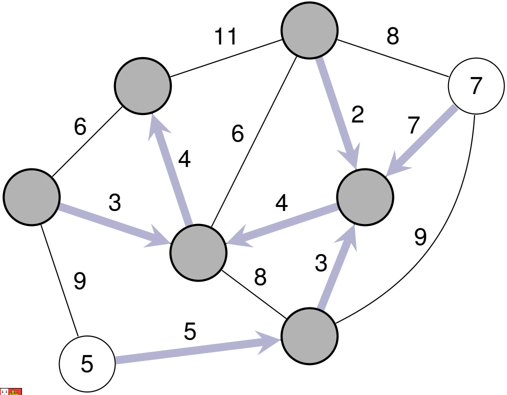
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Recap: Prim's Algorithm

Implementation

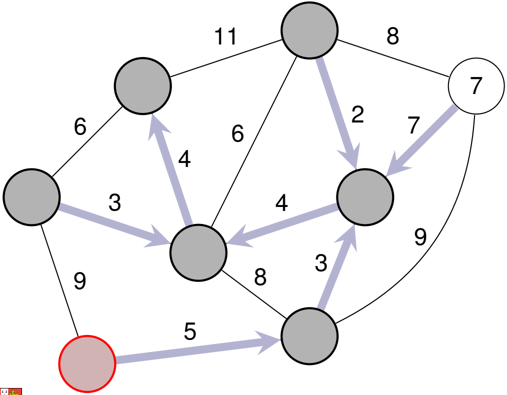
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Recap: Prim's Algorithm

Implementation

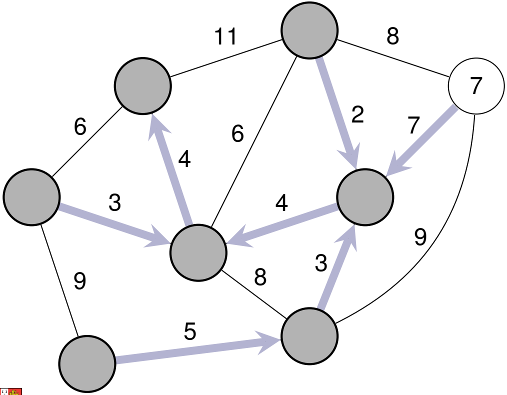
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Recap: Prim's Algorithm

Implementation

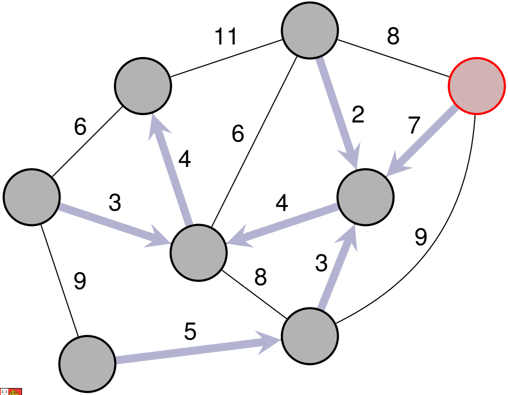
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Recap: Prim's Algorithm

Implementation

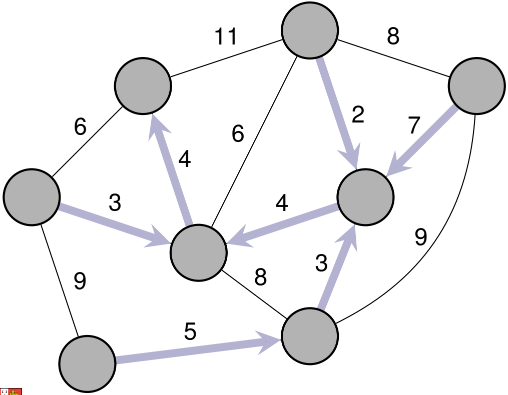
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Recap: Prim's Algorithm

Implementation

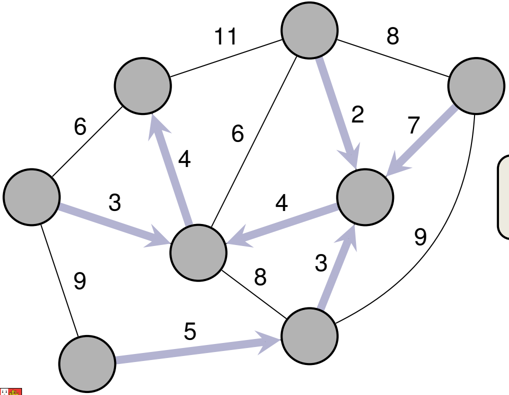
- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Recap: Prim's Algorithm

Implementation

- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q

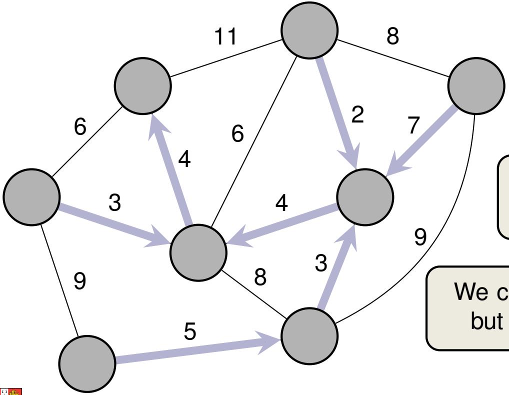


Final MST is given (implicitly) by the pointers!

Recap: Prim's Algorithm

Implementation

- Every vertex in Q has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from Q with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q



Final MST is given (implicitly) by the pointers!

We computed same MST as Kruskal, but in a completely different order!

Prim's Algorithms vs. Dijsktra's Algorithm

Prim's Algorithm

- Grows a tree that will eventually become a (minimum) spanning tree
- A is the set of vertices which have been connected so far
- Value of a vertex:
  - If $u \in A$, then it has no value.
  - If $u \notin A$, then it is equal to the smallest weight of an edge connecting to A (if such edge exists, otherwise ∞.)

Dijsktra's Algorithm

- Grows a tree that will eventually become a shortest-path tree
- S is the set of vertices in the (current) shortest-path tree
- Value of a vertex:
  - If $u \in S$, then it is the actual distance from the source s to u.
  - If $u \notin S$, then it may be any value (including ∞) that is at least the distance from the source s.

Dijkstra's Algorithm

Overview of Dijkstra

- Requires that all edges have non-negative weights
- Use a special order for relaxing edges

Dijkstra's Algorithm

Overview of Dijkstra

- Requires that all edges have non-negative weights
- Use a special order for relaxing edges
- The order follows a greedy-strategy (similar to Prim's algorithm):

Dijkstra's Algorithm

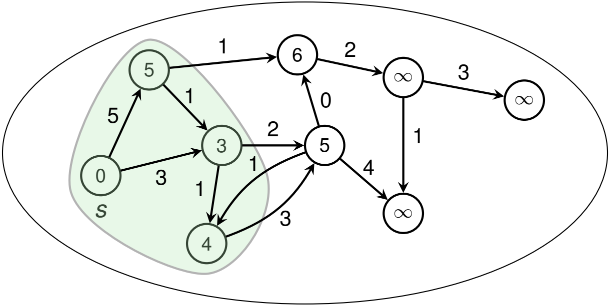
Overview of Dijkstra

- Requires that all edges have non-negative weights
- Use a special order for relaxing edges
- The order follows a greedy-strategy (similar to Prim's algorithm):
  1. Maintain set S of vertices u with $u.\delta = u.d$

Dijkstra's Algorithm

Overview of Dijkstra

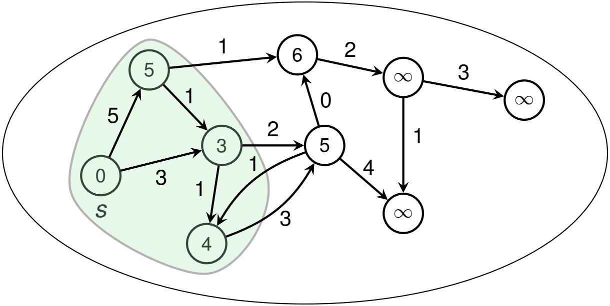
- Requires that all edges have non-negative weights
- Use a special order for relaxing edges
- The order follows a greedy-strategy (similar to Prim's algorithm):
  1. Maintain set S of vertices u with $u.\delta = u.d$



Dijkstra's Algorithm

Overview of Dijkstra

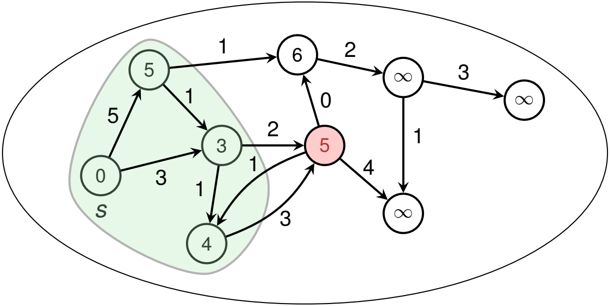
- Requires that all edges have non-negative weights
- Use a special order for relaxing edges
- The order follows a greedy-strategy (similar to Prim's algorithm):
  1. Maintain set S of vertices u with $u.\delta = u.d$
  2. At each step, add a vertex $v \in V \setminus S$ with minimal $v.\delta$



Dijkstra's Algorithm

Overview of Dijkstra

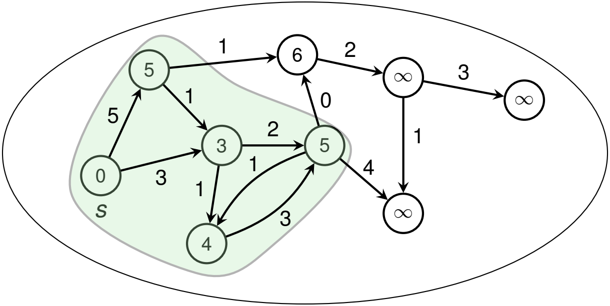
- Requires that all edges have non-negative weights
- Use a special order for relaxing edges
- The order follows a greedy-strategy (similar to Prim's algorithm):
  1. Maintain set S of vertices u with $u.\delta = u.d$
  2. At each step, add a vertex $v \in V \setminus S$ with minimal $v.\delta$



Dijkstra's Algorithm

Overview of Dijkstra

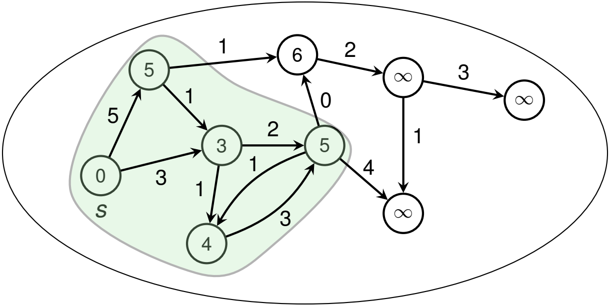
- Requires that all edges have non-negative weights
- Use a special order for relaxing edges
- The order follows a greedy-strategy (similar to Prim's algorithm):
  1. Maintain set S of vertices u with $u.\delta = u.d$
  2. At each step, add a vertex $v \in V \setminus S$ with minimal $v.\delta$



Dijkstra's Algorithm

Overview of Dijkstra

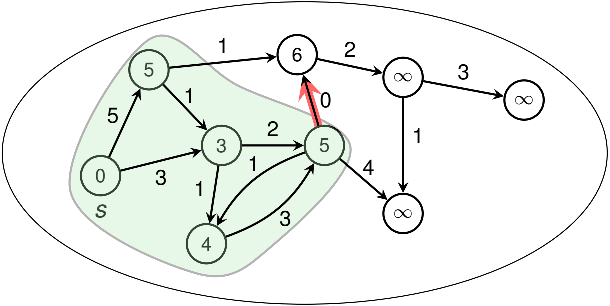
- Requires that all edges have non-negative weights
- Use a special order for relaxing edges
- The order follows a greedy-strategy (similar to Prim's algorithm):
  1. Maintain set S of vertices u with $u.\delta = u.d$
  2. At each step, add a vertex $v \in V \setminus S$ with minimal $v.\delta$
  3. Relax all edges leaving v



Dijkstra's Algorithm

Overview of Dijkstra

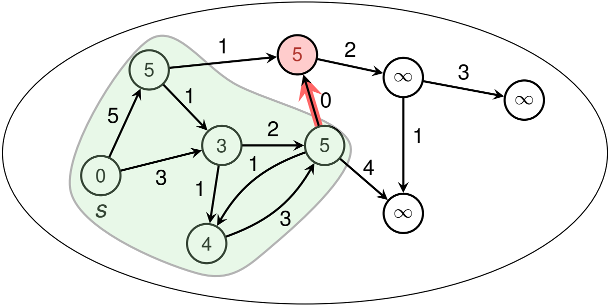
- Requires that all edges have non-negative weights
- Use a special order for relaxing edges
- The order follows a greedy-strategy (similar to Prim's algorithm):
  1. Maintain set S of vertices u with $u.\delta = u.d$
  2. At each step, add a vertex $v \in V \setminus S$ with minimal $v.\delta$
  3. Relax all edges leaving v



Dijkstra's Algorithm

Overview of Dijkstra

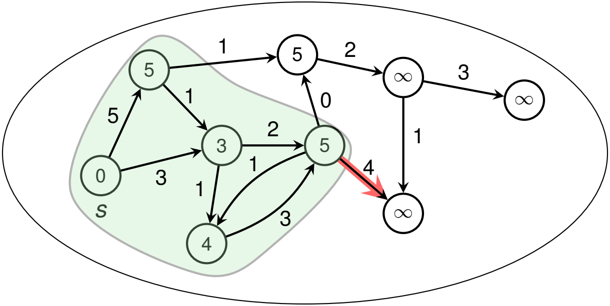
- Requires that all edges have non-negative weights
- Use a special order for relaxing edges
- The order follows a greedy-strategy (similar to Prim's algorithm):
  1. Maintain set S of vertices u with $u.\delta = u.d$
  2. At each step, add a vertex $v \in V \setminus S$ with minimal $v.\delta$
  3. Relax all edges leaving v



Dijkstra's Algorithm

Overview of Dijkstra

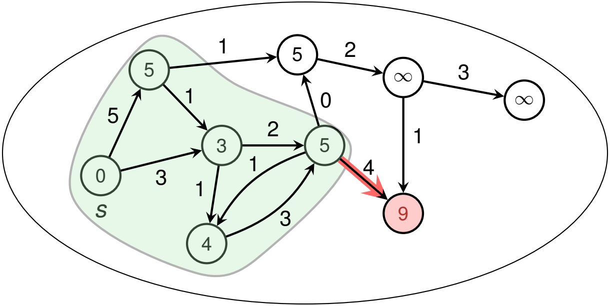
- Requires that all edges have non-negative weights
- Use a special order for relaxing edges
- The order follows a greedy-strategy (similar to Prim's algorithm):
  1. Maintain set S of vertices u with $u.\delta = u.d$
  2. At each step, add a vertex $v \in V \setminus S$ with minimal $v.\delta$
  3. Relax all edges leaving v



Dijkstra's Algorithm

Overview of Dijkstra

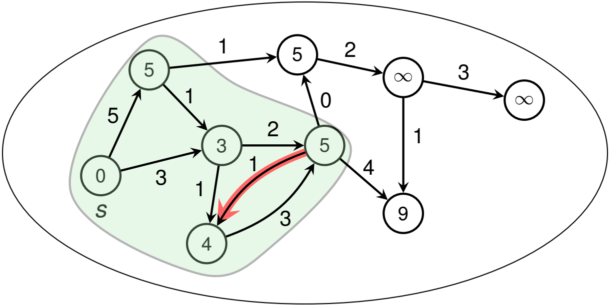
- Requires that all edges have non-negative weights
- Use a special order for relaxing edges
- The order follows a greedy-strategy (similar to Prim's algorithm):
  1. Maintain set S of vertices u with $u.\delta = u.d$
  2. At each step, add a vertex $v \in V \setminus S$ with minimal $v.\delta$
  3. Relax all edges leaving v



Dijkstra's Algorithm

Overview of Dijkstra

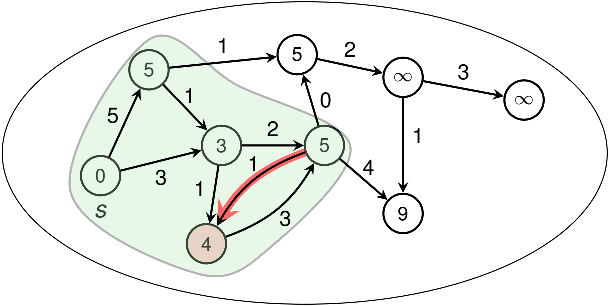
- Requires that all edges have non-negative weights
- Use a special order for relaxing edges
- The order follows a greedy-strategy (similar to Prim's algorithm):
  1. Maintain set S of vertices u with $u.\delta = u.d$
  2. At each step, add a vertex $v \in V \setminus S$ with minimal $v.\delta$
  3. Relax all edges leaving v



Dijkstra's Algorithm

Overview of Dijkstra

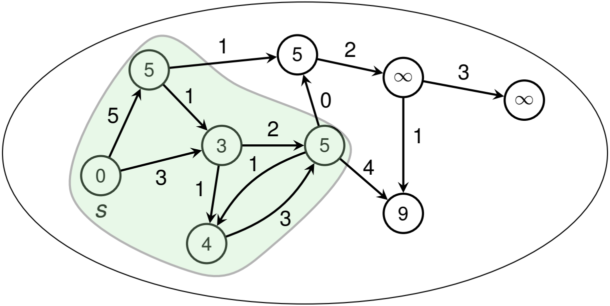
- Requires that all edges have non-negative weights
- Use a special order for relaxing edges
- The order follows a greedy-strategy (similar to Prim's algorithm):
  1. Maintain set S of vertices u with $u.\delta = u.d$
  2. At each step, add a vertex $v \in V \setminus S$ with minimal $v.\delta$
  3. Relax all edges leaving v



Dijkstra's Algorithm

Overview of Dijkstra

- Requires that all edges have non-negative weights
- Use a special order for relaxing edges
- The order follows a greedy-strategy (similar to Prim's algorithm):
  1. Maintain set S of vertices u with $u.\delta = u.d$
  2. At each step, add a vertex $v \in V \setminus S$ with minimal $v.\delta$
  3. Relax all edges leaving v



Dijkstra's Algorithm

Overview of Dijkstra

- Requires that all edges have non-negative weights
- Use a special order for relaxing edges
- The order follows a greedy-strategy (similar to Prim's algorithm):
  1. Maintain set S of vertices u with $u.\delta = u.d$
  2. At each step, add a vertex $v \in V \setminus S$ with minimal $v.\delta$
  3. Relax all edges leaving v

Dijkstra's Algorithm

Overview of Dijkstra

- Requires that all edges have non-negative weights
- Use a special order for relaxing edges
- The order follows a greedy-strategy (similar to Prim's algorithm):
  1. Maintain set S of vertices u with $u.\delta = u.d$
  2. At each step, add a vertex $v \in V \setminus S$ with minimal $v.\delta$
  3. Relax all edges leaving v

```
DIJKSTRA(G,w,s)
INITIALIZE(G,s)
S = ∅
Q = V
while Q ≠ ∅ do
    u = Extract-Min(Q)
    S = S ∪ {u}
    for each v ∈ G.Adj[u] do
        RELAX(u, v, w)
    end for
end while
```

Details of Dijkstra's Algorithm

As in Prim, use **priority queue** Q to keep track of the vertices' values.

```
DIJKSTRA(G,w,s)
INITIALIZE(G,s)
S = ∅
Q = V
while Q ≠ ∅ do
    u = Extract-Min(Q)
    S = S ∪ {u}
    for each v ∈ G.Adj[u] do
        RELAX(u, v, w)
    end for
end while
```

Details of Dijkstra's Algorithm

As in Prim, use **priority queue** Q to keep track of the vertices' values.

DIJKSTRA(G,w,s)

0: INITIALIZE(G,s)
1: $S = \emptyset$
2: $Q = V$
3: **while** $Q \neq \emptyset$ **do**
4: $\quad u =$ Extract-Min(Q)
5: $\quad S = S \cup \{u\}$
6: $\quad$ **for** each $v \in G.Adj[u]$ **do**
7: $\quad\quad$ RELAX(u, v, w)
8: $\quad$ **end for**
9: **end while**

Runtime w. Fibonacci Heaps

Details of Dijkstra's Algorithm

As in Prim, use **priority queue** Q to keep track of the vertices' values.

DIJKSTRA(G,w,s)

INITIALIZE(G,s)
$S = \emptyset$
$Q = V$
while $Q \neq \emptyset$ **do**
    $u =$ Extract-Min(Q)
    $S = S \cup \{u\}$
    for each $v \in G.Adj[u]$ **do**
        RELAX(u, v, w)
    end for
end while

Runtime w. Fibonacci Heaps

Details of Dijkstra's Algorithm

As in Prim, use **priority queue** Q to keep track of the vertices' values.

```
DIJKSTRA(G,w,s)
0: INITIALIZE(G,s)
1: S = ∅
2: Q = V
3: while Q ≠ ∅ do
4:     u = Extract-Min(Q)
5:     S = S ∪ {u}
6:     for each v ∈ G.Adj[u] do
7:         RELAX(u, v, w)
8:     end for
9: end while
```

Runtime w. Fibonacci Heaps

- Initialization (l. 0-2): $\mathcal{O}(V)$

Details of Dijkstra's Algorithm

As in Prim, use **priority queue** Q to keep track of the vertices' values.

DIJKSTRA(G,w,s)

0: INITIALIZE(G,s)
1: $S = \emptyset$
2: $Q = V$
3: **while** $Q \neq \emptyset$ **do**
4: $\quad u =$ Extract-Min(Q)
5: $\quad S = S \cup \{u\}$
6: $\quad$ **for** each $v \in G.Adj[u]$ **do**
7: $\qquad$ RELAX(u, v, w)
8: $\quad$ **end for**
9: **end while**

Runtime w. Fibonacci Heaps

- Initialization (l. 0-2): $\mathcal{O}(V)$

Details of Dijkstra's Algorithm

> As in Prim, use **priority queue** Q to keep track of the vertices' values.

```
DIJKSTRA(G,w,s)
0: INITIALIZE(G,s)
1: S = ∅
2: Q = V
3: while Q ≠ ∅ do
4:     u = Extract-Min(Q)
5:     S = S ∪ {u}
6:     for each v ∈ G.Adj[u] do
7:         RELAX(u, v, w)
8:     end for
9: end while
```

Runtime w. Fibonacci Heaps

- Initialization (l. 0-2): $\mathcal{O}(V)$
- ExtractMin (l. 4): $\mathcal{O}(V \cdot \log V)$

Details of Dijkstra's Algorithm

As in Prim, use **priority queue** Q to keep track of the vertices' values.

```
DIJKSTRA(G,w,s)
0: INITIALIZE(G,s)
1: S = ∅
2: Q = V
3: while Q ≠ ∅ do
4:     u = Extract-Min(Q)
5:     S = S ∪ {u}
6:     for each v ∈ G.Adj[u] do
7:         RELAX(u, v, w)
8:     end for
9: end while
```

Runtime w. Fibonacci Heaps

- Initialization (l. 0-2): $\mathcal{O}(V)$
- ExtractMin (l. 4): $\mathcal{O}(V \cdot \log V)$

Details of Dijkstra's Algorithm

As in Prim, use **priority queue** Q to keep track of the vertices' values.

```
DIJKSTRA(G,w,s)
0: INITIALIZE(G,s)
1: S = ∅
2: Q = V
3: while Q ≠ ∅ do
4:     u = Extract-Min(Q)
5:     S = S ∪ {u}
6:     for each v ∈ G.Adj[u] do
7:         RELAX(u, v, w)
8:     end for
9: end while
```

Runtime w. Fibonacci Heaps

- Initialization (l. 0-2): $\mathcal{O}(V)$
- ExtractMin (l. 4): $\mathcal{O}(V \cdot \log V)$
- DecreaseKey (l. 7): $\mathcal{O}(E \cdot 1)$

Details of Dijkstra's Algorithm

As in Prim, use **priority queue** Q to keep track of the vertices' values.

DIJKSTRA(G,w,s)

0: INITIALIZE(G,s)
1: $S = \emptyset$
2: $Q = V$
3: **while** $Q \neq \emptyset$ **do**
4: $\quad u =$ Extract-Min(Q)
5: $\quad S = S \cup \{u\}$
6: $\quad$ **for** each $v \in G.Adj[u]$ **do**
7: $\quad\quad$ RELAX(u, v, w)
8: $\quad$ **end for**
9: **end while**

Runtime w. Fibonacci Heaps

- Initialization (l. 0-2): $\mathcal{O}(V)$
- ExtractMin (l. 4): $\mathcal{O}(V \cdot \log V)$
- DecreaseKey (l. 7): $\mathcal{O}(E \cdot 1)$

Details of Dijkstra's Algorithm

As in Prim, use **priority queue** Q to keep track of the vertices' values.

```
DIJKSTRA(G,w,s)
0: INITIALIZE(G,s)
1: S = ∅
2: Q = V
3: while Q ≠ ∅ do
4:     u = Extract-Min(Q)
5:     S = S ∪ {u}
6:     for each v ∈ G.Adj[u] do
7:         RELAX(u, v, w)
8:     end for
9: end while
```

Runtime w. Fibonacci Heaps

- Initialization (l. 0-2): $\mathcal{O}(V)$
- ExtractMin (l. 4): $\mathcal{O}(V \cdot \log V)$
- DecreaseKey (l. 7): $\mathcal{O}(E \cdot 1)$

$\Rightarrow$ Overall: $\mathcal{O}(V \log V + E)$

Details of Dijkstra's Algorithm

As in Prim, use **priority queue** Q to keep track of the vertices' values.

```
DIJKSTRA(G,w,s)
0: INITIALIZE(G,s)
1: S = ∅
2: Q = V
3: while Q ≠ ∅ do
4:     u = Extract-Min(Q)
5:     S = S ∪ {u}
6:     for each v ∈ G.Adj[u] do
7:         RELAX(u, v, w)
8:     end for
9: end while
```

Runtime w. Fibonacci Heaps

- Initialization (l. 0-2): $\mathcal{O}(V)$
- ExtractMin (l. 4): $\mathcal{O}(V \cdot \log V)$
- DecreaseKey (l. 7): $\mathcal{O}(E \cdot 1)$

$\Rightarrow$ Overall: $\mathcal{O}(V \log V + E)$

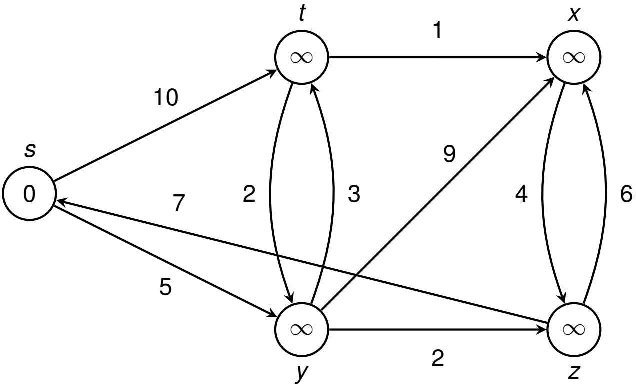
With a binary heap instead, the overall runtime would be $O(E \cdot \log V)$!

Prim's algorithm has the same runtime!

Execution of Dijkstra (Figure 24.6)

Priority Queue Q:

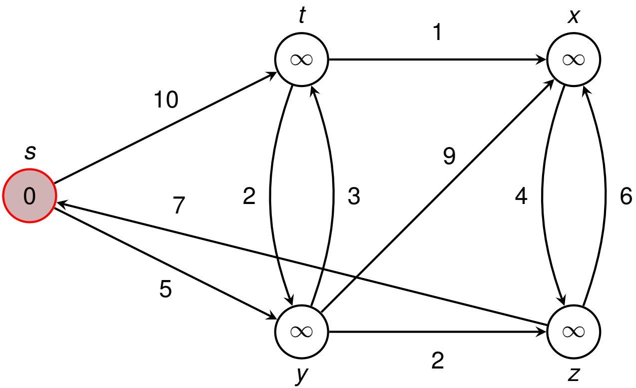
$(s, 0), (t, \infty), (x, \infty), (y, \infty), (z, \infty)$



Execution of Dijkstra (Figure 24.6)

Priority Queue Q:

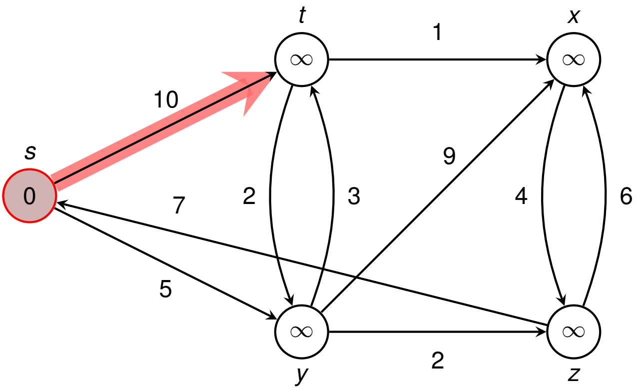
$\cancel{(s, 0)}, (t, \infty), (x, \infty), (y, \infty), (z, \infty)$



Execution of Dijkstra (Figure 24.6)

Priority Queue Q:

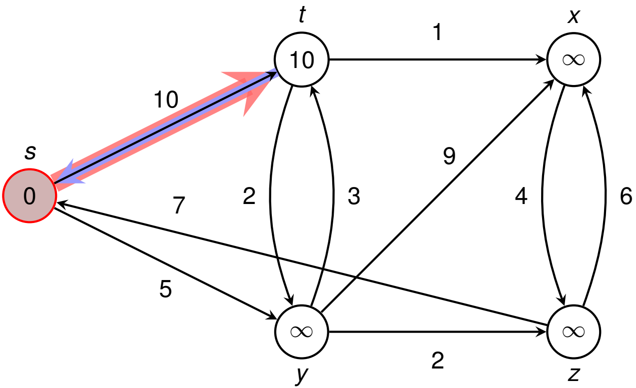
$\cancel{(s,0)}, (t,\infty), (x,\infty), (y,\infty), (z,\infty)$



Execution of Dijkstra (Figure 24.6)

Priority Queue Q:

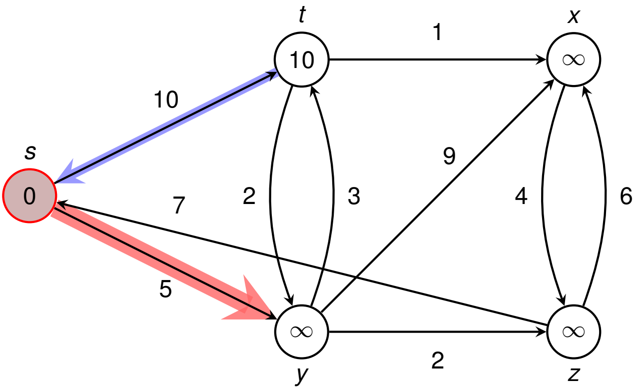
$\cancel{(s, 0)}, (t, 10), (x, \infty), (y, \infty), (z, \infty)$



Execution of Dijkstra (Figure 24.6)

Priority Queue Q:

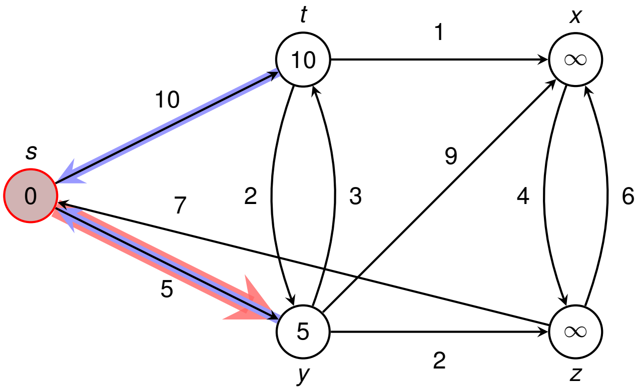
$\cancel{(s, 0)}, (t, 10), (x, \infty), (y, \infty), (z, \infty)$



Execution of Dijkstra (Figure 24.6)

Priority Queue Q:

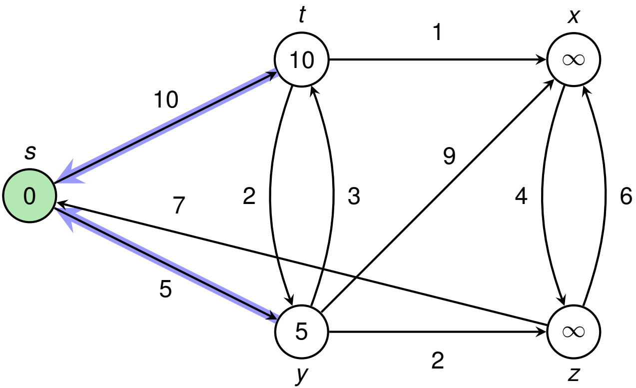
$\cancel{(s,0)}, (t,10), (x,\infty), (y,5), (z,\infty)$



Execution of Dijkstra (Figure 24.6)

Priority Queue Q:

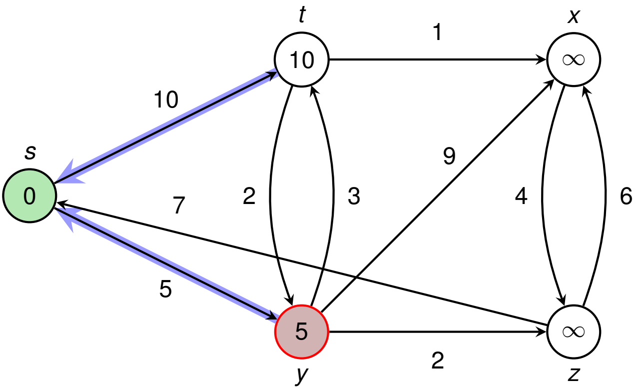
$\cancel{(s,0)}, (t,10), (x,\infty), (y,5), (z,\infty)$



Execution of Dijkstra (Figure 24.6)

Priority Queue Q:

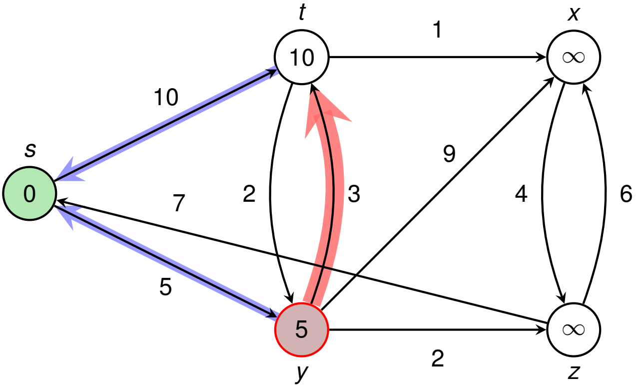
$(t, 10), (x, \infty), \cancel{(y, 5)}, (z, \infty)$



Execution of Dijkstra (Figure 24.6)

Priority Queue Q:

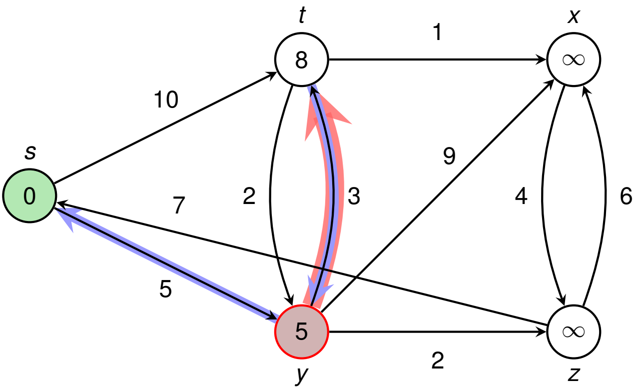
$(t, 10), (x, \infty), \cancel{(y, 5)}, (z, \infty)$



Execution of Dijkstra (Figure 24.6)

Priority Queue Q:

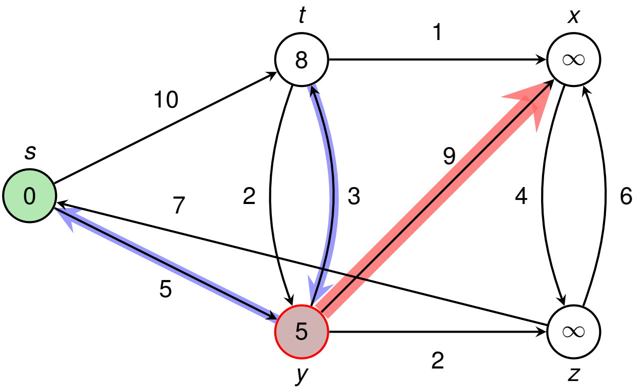
$(t, 8), (x, \infty), \cancel{(y, 5)}, (z, \infty)$



Execution of Dijkstra (Figure 24.6)

Priority Queue Q:

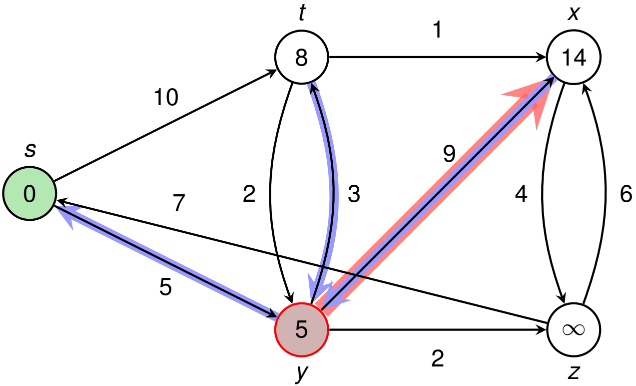
$(t, 8), (x, \infty), \cancel{(y, 5)}, (z, \infty)$



Execution of Dijkstra (Figure 24.6)

Priority Queue Q:

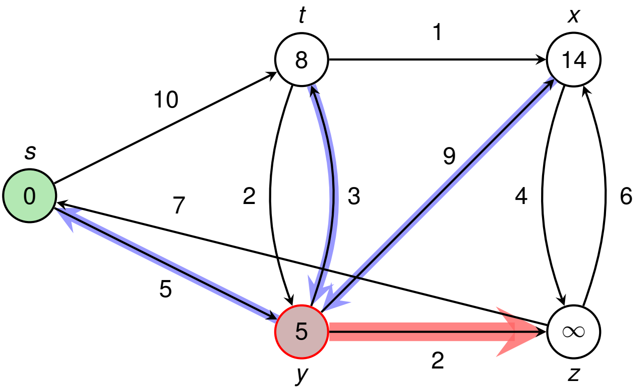
$(t, 8), (x, 14), \cancel{(y, 5)}, (z, \infty)$



Execution of Dijkstra (Figure 24.6)

Priority Queue Q:

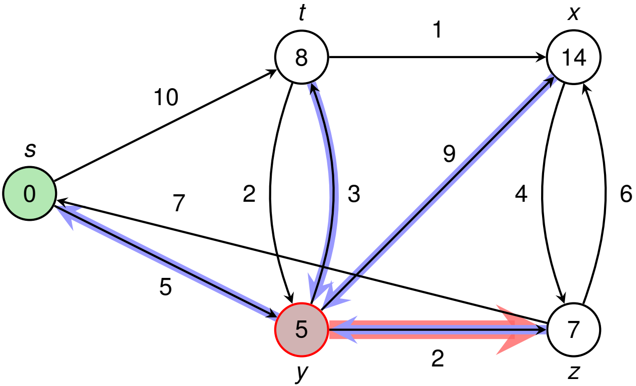
$(t, 8), (x, 14), \cancel{(y, 5)}, (z, \infty)$



Execution of Dijkstra (Figure 24.6)

Priority Queue Q:

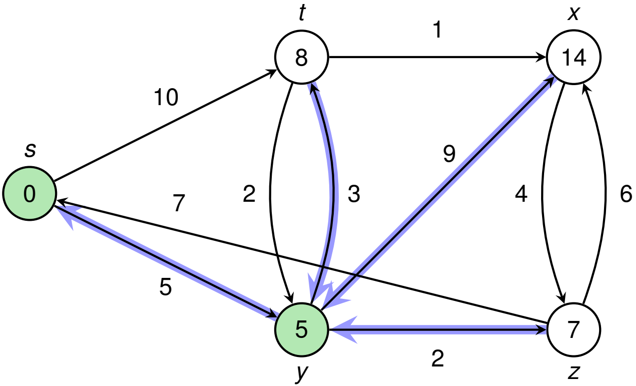
$(t, 8), (x, 14), \cancel{(y, 5)}, (z, 7)$



Execution of Dijkstra (Figure 24.6)

Priority Queue Q:

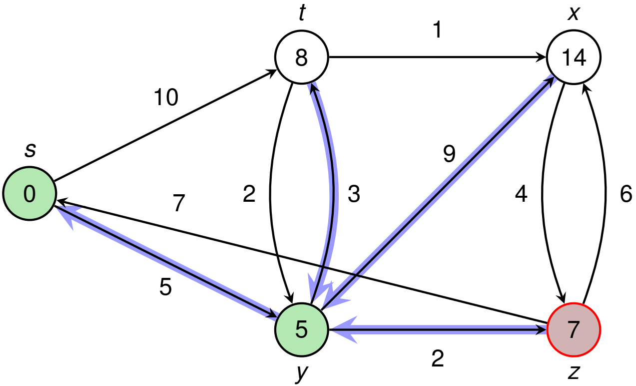
$(t, 8), (x, 14), \cancel{(y, 5)}, (z, 7)$



Execution of Dijkstra (Figure 24.6)

Priority Queue Q:

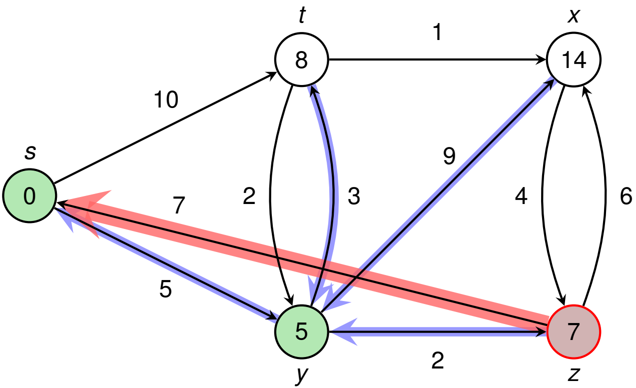
$(t, 8), (x, 14), \cancel{(z, 7)}$



Execution of Dijkstra (Figure 24.6)

Priority Queue Q:

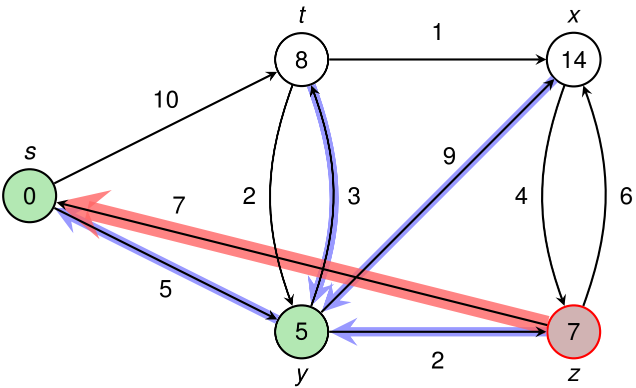
$(t, 8), (x, 14), \cancel{(z, 7)}$



Execution of Dijkstra (Figure 24.6)

Priority Queue Q:

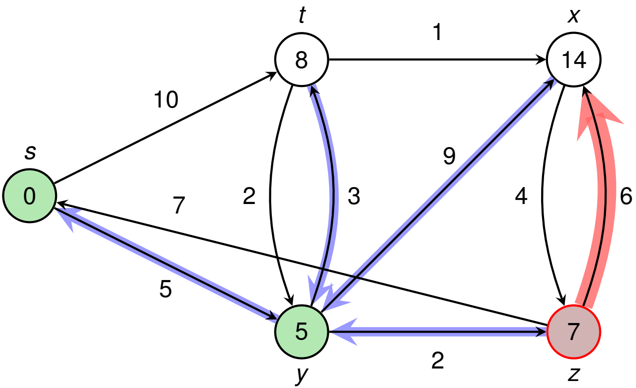
$(t, 8), (x, 14), \cancel{(z, 7)}$



Execution of Dijkstra (Figure 24.6)

Priority Queue Q:

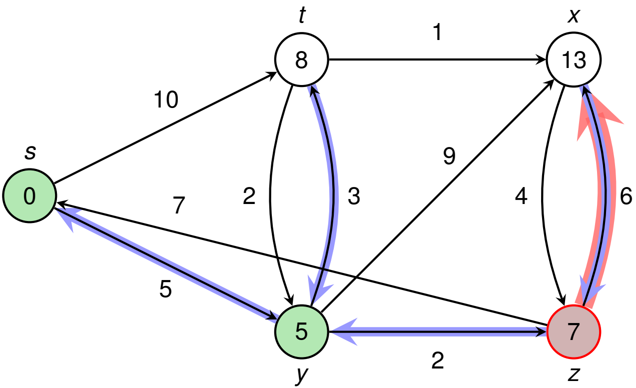
$(t, 8), (x, 14), \cancel{(z, 7)}$



Execution of Dijkstra (Figure 24.6)

Priority Queue Q:

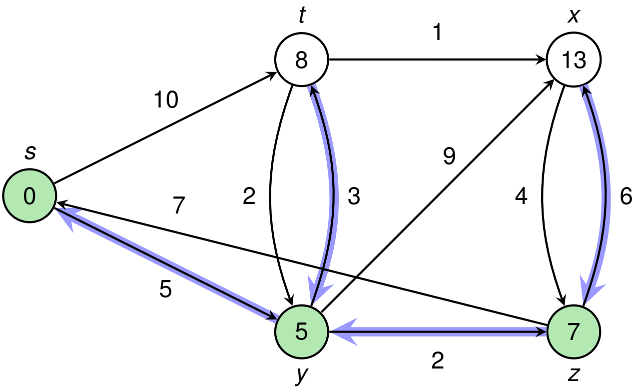
$(t, 8), (x, 13), \cancel{(z, 7)}$



Execution of Dijkstra (Figure 24.6)

Priority Queue Q:

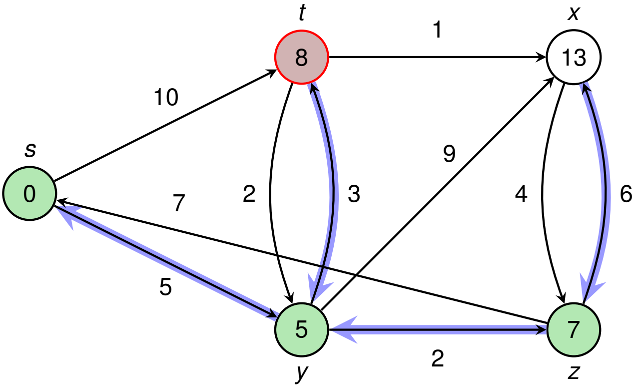
$(t, 8), (x, 13), \cancel{(z, 7)}$



Execution of Dijkstra (Figure 24.6)

Priority Queue Q:

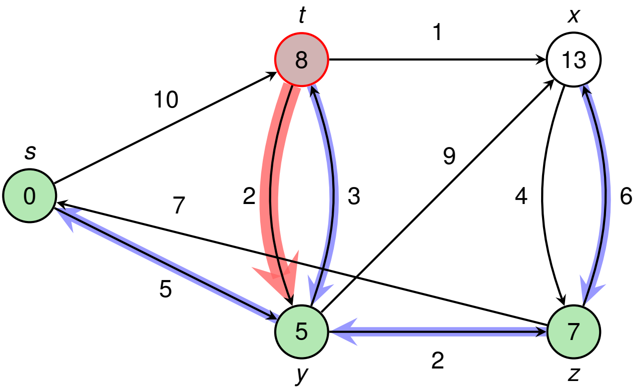
~~$(t, 8)$~~, $(x, 13)$



Execution of Dijkstra (Figure 24.6)

Priority Queue Q:

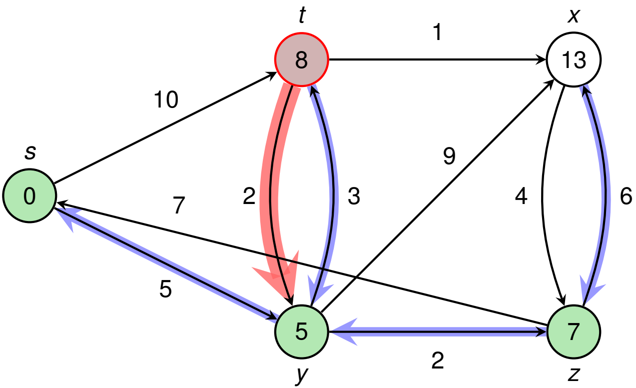
$\cancel{(t, 8)}, (x, 13)$



Execution of Dijkstra (Figure 24.6)

Priority Queue Q:

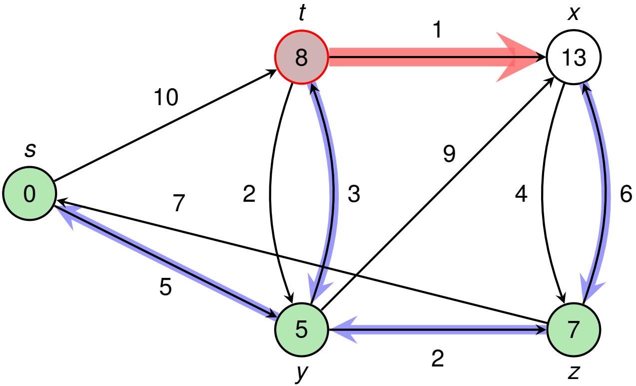
$\cancel{(t, 8)}, (x, 13)$



Execution of Dijkstra (Figure 24.6)

Priority Queue Q:

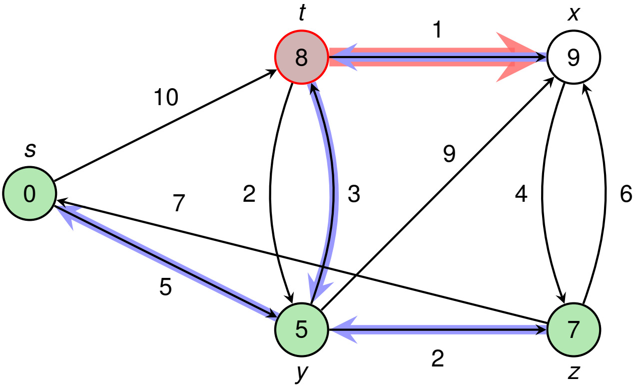
$\cancel{(t, 8)}, (x, 9)$



Execution of Dijkstra (Figure 24.6)

Priority Queue Q:

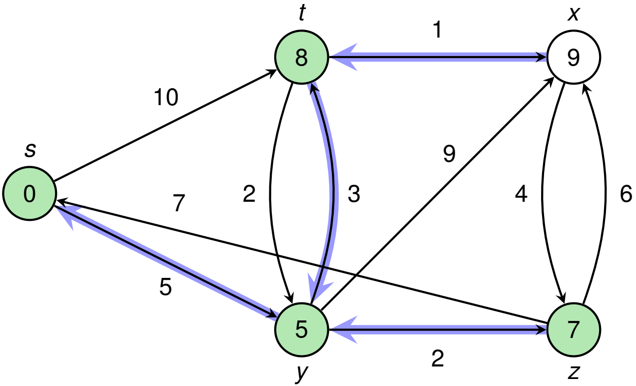
$\cancel{(t, 8)}, (x, 9)$



Execution of Dijkstra (Figure 24.6)

Priority Queue Q:

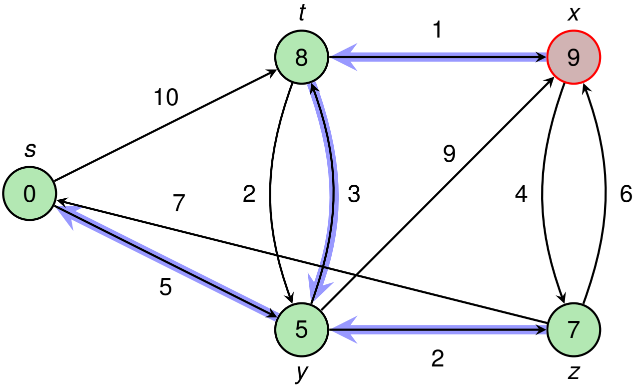
$\cancel{(t, 8)}, (x, 9)$



Execution of Dijkstra (Figure 24.6)

Priority Queue Q:

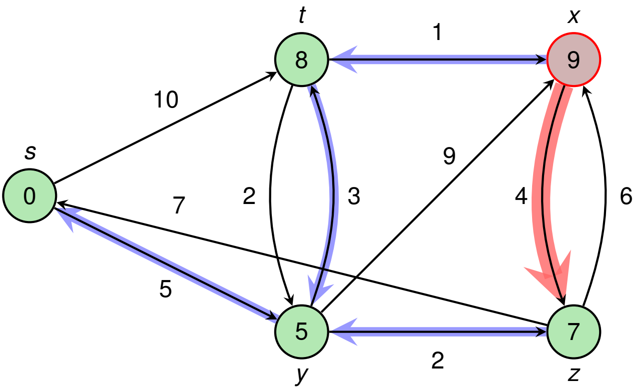
$\cancel{(x, 9)}$



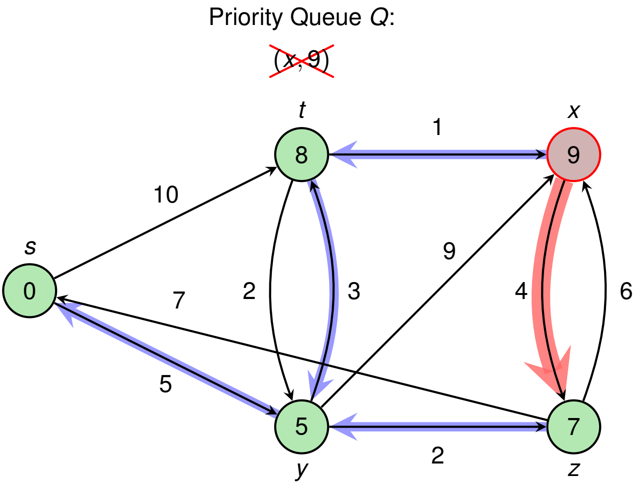
Execution of Dijkstra (Figure 24.6)

Priority Queue Q:

~~$(x, 9)$~~



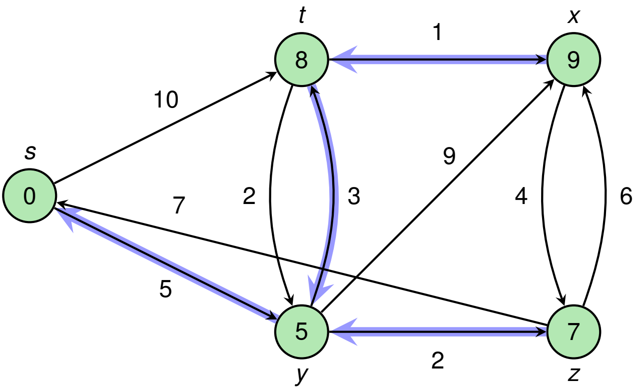
Execution of Dijkstra (Figure 24.6)



Execution of Dijkstra (Figure 24.6)

Priority Queue Q:

~~$(x, 9)$~~



Dijkstra's Algorithm: Correctness

Correctness (Theorem 24.6)

For any directed graph $G = (V, E)$ with non-negative edge weights $w : E \to \mathbb{R}^+$ and source s, Dijkstra terminates with $u.d = u.\delta$ for all $u \in V$.

Proof: Invariant: If v is extracted, $v.d = v.\delta$

Dijkstra's Algorithm: Correctness

Correctness (Theorem 24.6)

For any directed graph $G = (V, E)$ with non-negative edge weights $w : E \to \mathbb{R}^+$ and source s, Dijkstra terminates with $u.d = u.\delta$ for all $u \in V$.

Proof: $\boxed{\text{Invariant: If } v \text{ is extracted, } v.d = v.\delta}$

- Suppose there is $u \in V$, when extracted,

$$u.d > u.\delta$$

Dijkstra's Algorithm: Correctness

Correctness (Theorem 24.6)

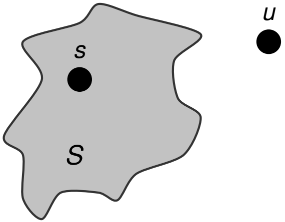
For any directed graph $G = (V, E)$ with non-negative edge weights $w : E \to \mathbb{R}^+$ and source s, Dijkstra terminates with $u.d = u.\delta$ for all $u \in V$.

Proof: $\boxed{\text{Invariant: If } v \text{ is extracted, } v.d = v.\delta}$

- Suppose there is $u \in V$, when extracted,

$$u.d > u.\delta$$

- Let u be the first vertex with this property



Dijkstra's Algorithm: Correctness

Correctness (Theorem 24.6)

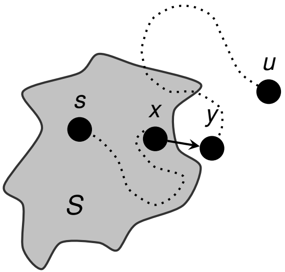
For any directed graph $G = (V, E)$ with non-negative edge weights $w : E \to \mathbb{R}^+$ and source s, Dijkstra terminates with $u.d = u.\delta$ for all $u \in V$.

Proof: $\boxed{\text{Invariant: If } v \text{ is extracted, } v.d = v.\delta}$

- Suppose there is $u \in V$, when extracted,

$$u.d > u.\delta$$

- Let u be the first vertex with this property
- Take a shortest path from s to u and let (x, y) be the first edge from S to $V \setminus S$



Dijkstra's Algorithm: Correctness

Correctness (Theorem 24.6)

For any directed graph $G = (V, E)$ with non-negative edge weights $w : E \to \mathbb{R}^+$ and source s, Dijkstra terminates with $u.d = u.\delta$ for all $u \in V$.

Proof: $\boxed{\text{Invariant: If } v \text{ is extracted, } v.d = v.\delta}$

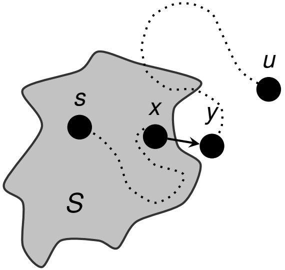
- Suppose there is $u \in V$, when extracted,

$$u.d > u.\delta$$

- Let u be the first vertex with this property
- Take a shortest path from s to u and let (x, y) be the first edge from S to $V \setminus S$

$\Rightarrow$

$$u.d$$



Dijkstra's Algorithm: Correctness

Correctness (Theorem 24.6)

For any directed graph $G = (V, E)$ with non-negative edge weights $w : E \to \mathbb{R}^+$ and source s, Dijkstra terminates with $u.d = u.\delta$ for all $u \in V$.

Proof: $\boxed{\text{Invariant: If } v \text{ is extracted, } v.d = v.\delta}$

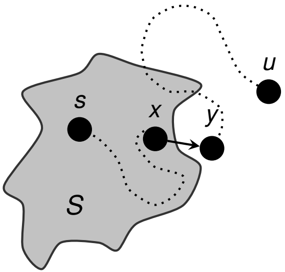
- Suppose there is $u \in V$, when extracted,

$$u.d > u.\delta$$

- Let u be the first vertex with this property
- Take a shortest path from s to u and let (x, y) be the first edge from S to $V \setminus S$

$\Rightarrow$

$$u.d \leq y.d$$



Dijkstra's Algorithm: Correctness

Correctness (Theorem 24.6)

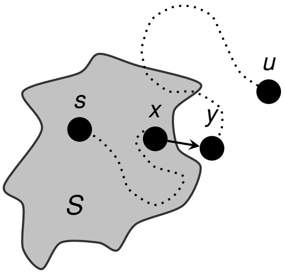
For any directed graph $G = (V, E)$ with non-negative edge weights $w : E \to \mathbb{R}^+$ and source s, Dijkstra terminates with $u.d = u.\delta$ for all $u \in V$.

Proof: $\boxed{\text{Invariant: If } v \text{ is extracted, } v.d = v.\delta}$

- Suppose there is $u \in V$, when extracted,

$$u.d > u.\delta$$

- Let u be the first vertex with this property
- Take a shortest path from s to u and let (x, y) be the first edge from S to $V \setminus S$



$\Rightarrow$

$$u.d \leq y.d$$

u is extracted before y

Dijkstra's Algorithm: Correctness

Correctness (Theorem 24.6)

For any directed graph $G = (V, E)$ with non-negative edge weights $w : E \to \mathbb{R}^+$ and source s, Dijkstra terminates with $u.d = u.\delta$ for all $u \in V$.

Proof: $\boxed{\text{Invariant: If } v \text{ is extracted, } v.d = v.\delta}$

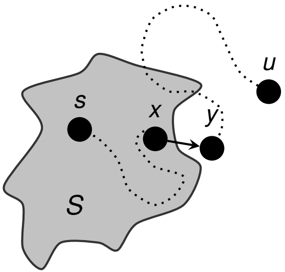
- Suppose there is $u \in V$, when extracted,

$$u.d > u.\delta$$

- Let u be the first vertex with this property
- Take a shortest path from s to u and let (x, y) be the first edge from S to $V \setminus S$

$\Rightarrow$

$$u.d \leq y.d = y.\delta$$



Dijkstra's Algorithm: Correctness

Correctness (Theorem 24.6)

For any directed graph $G = (V, E)$ with non-negative edge weights $w : E \to \mathbb{R}^+$ and source s, Dijkstra terminates with $u.d = u.\delta$ for all $u \in V$.

Proof: Invariant: If v is extracted, $v.d = v.\delta$

- Suppose there is $u \in V$, when extracted,

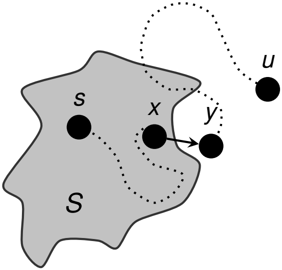
$$u.d > u.\delta$$

- Let u be the first vertex with this property
- Take a shortest path from s to u and let (x, y) be the first edge from S to $V \setminus S$

$\Rightarrow$

$$u.d \leq y.d = y.\delta$$

since $x.d = x.\delta$ when x is extracted, and then (x, y) is relaxed $\Rightarrow$ Convergence Property



Dijkstra's Algorithm: Correctness

Correctness (Theorem 24.6)

For any directed graph $G = (V, E)$ with non-negative edge weights $w : E \to \mathbb{R}^+$ and source s, Dijkstra terminates with $u.d = u.\delta$ for all $u \in V$.

Proof: Invariant: If v is extracted, $v.d = v.\delta$

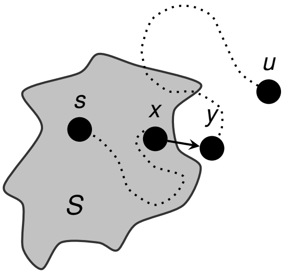
- Suppose there is $u \in V$, when extracted,

$$u.d > u.\delta$$

- Let u be the first vertex with this property
- Take a shortest path from s to u and let (x, y) be the first edge from S to $V \setminus S$

$\Rightarrow$

$$u.\delta < u.d \leq y.d = y.\delta$$



Dijkstra's Algorithm: Correctness

Correctness (Theorem 24.6)

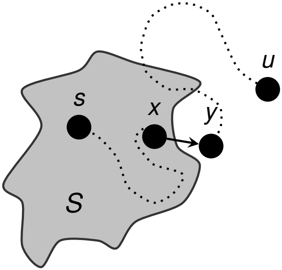
For any directed graph $G = (V, E)$ with non-negative edge weights $w : E \to \mathbb{R}^+$ and source s, Dijkstra terminates with $u.d = u.\delta$ for all $u \in V$.

Proof: $\boxed{\text{Invariant: If } v \text{ is extracted, } v.d = v.\delta}$

- Suppose there is $u \in V$, when extracted,

$$u.d > u.\delta$$

- Let u be the first vertex with this property
- Take a shortest path from s to u and let (x, y) be the first edge from S to $V \setminus S$



$\Rightarrow$

$$u.\delta < u.d \leq y.d = y.\delta$$

This contradicts that y is on a shortest path from s to u. $\square$

Dijkstra's Algorithm: Correctness

Correctness (Theorem 24.6)

For any directed graph $G = (V, E)$ with non-negative edge weights $w : E \to \mathbb{R}^+$ and source s, Dijkstra terminates with $u.d = u.\delta$ for all $u \in V$.

There are edge cases like $s = x$ and/or $y = u$!

Proof: $\boxed{\text{Invariant: If } v \text{ is extracted, } v.d = v.\delta}$

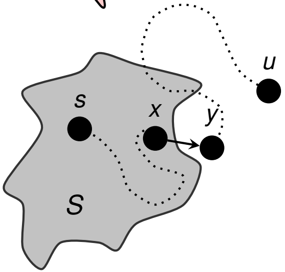
- Suppose there is $u \in V$, when extracted,

$$u.d > u.\delta$$

- Let u be the first vertex with this property
- Take a shortest path from s to u and let (x, y) be the first edge from S to $V \setminus S$

$\Rightarrow$

$$u.\delta < u.d \leq y.d = y.\delta$$

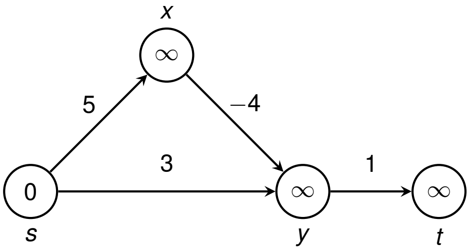


This contradicts that y is on a shortest path from s to u. $\square$

Why negative-weight edges don't work

Priority Queue Q:

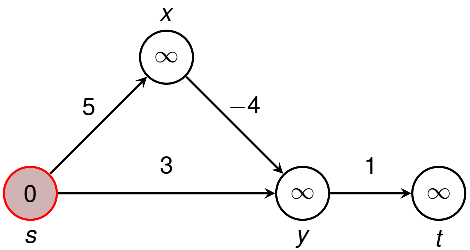
$$(s, 0), (t, \infty), (x, \infty), (y, \infty)$$



Why negative-weight edges don't work

Priority Queue Q:

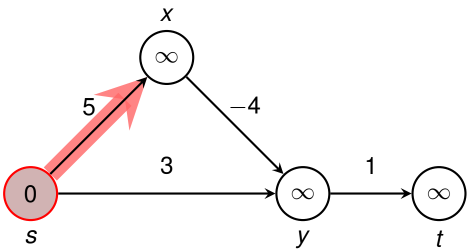
$\cancel{(s, 0)}, (t, \infty), (x, \infty), (y, \infty)$



Why negative-weight edges don't work

Priority Queue Q:

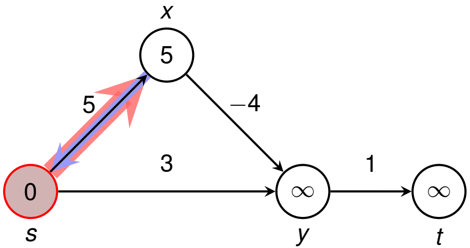
$\cancel{(s, 0)}, (t, \infty), (x, \infty), (y, \infty)$



Why negative-weight edges don't work

Priority Queue Q:

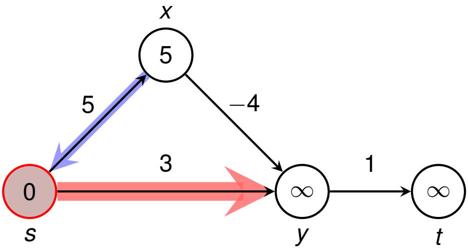
$\cancel{(s, 0)}, (t, \infty), (x, 5), (y, \infty)$



Why negative-weight edges don't work

Priority Queue Q:

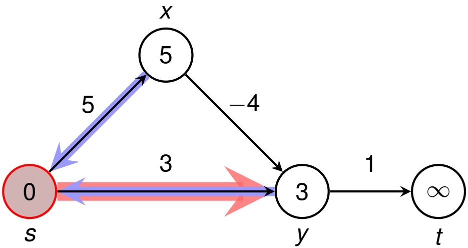
$\cancel{(s, 0)}, (t, \infty), (x, 5), (y, \infty)$



Why negative-weight edges don't work

Priority Queue Q:

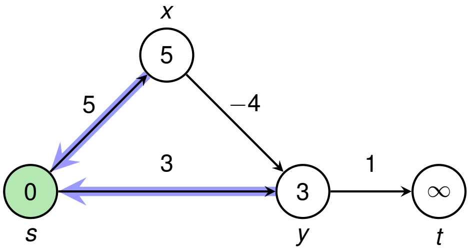
$\cancel{(s, 0)}, (t, \infty), (x, 5), (y, 3)$



Why negative-weight edges don't work

Priority Queue Q:

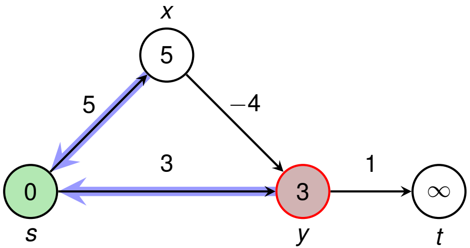
$\cancel{(s,0)}, (t,\infty), (x,5), (y,3)$



Why negative-weight edges don't work

Priority Queue Q:

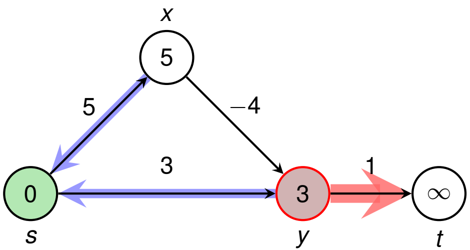
$(t, \infty), (x, 4), \cancel{(y, 3)}$



Why negative-weight edges don't work

Priority Queue Q:

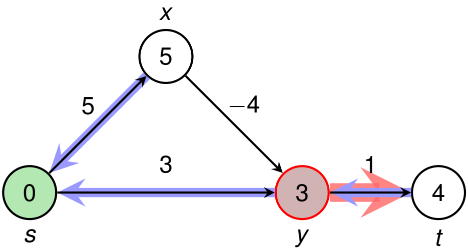
$(t, \infty), (x, 4), \cancel{(y, 3)}$



Why negative-weight edges don't work

Priority Queue Q:

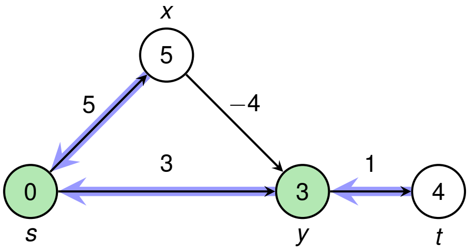
$(t, 4), (x, 5), \cancel{(y, 3)}$



Why negative-weight edges don't work

Priority Queue Q:

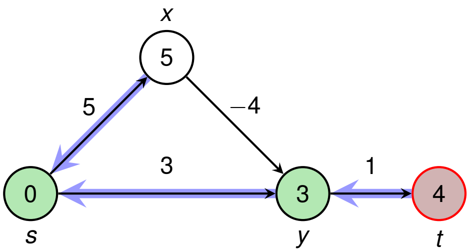
$(t, 4), (x, 5), \cancel{(y, 3)}$



Why negative-weight edges don't work

Priority Queue Q:

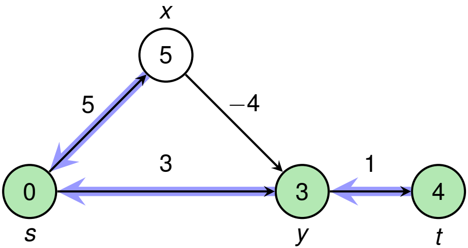
$\cancel{(t, 4)}, (x, 5)$



Why negative-weight edges don't work

Priority Queue Q:

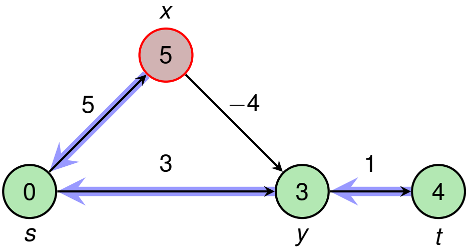
$\cancel{(t, 4)}, (x, 5)$



Why negative-weight edges don't work

Priority Queue Q:

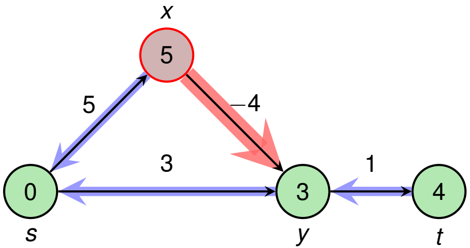
~~(x, 5)~~



Why negative-weight edges don't work

Priority Queue Q:

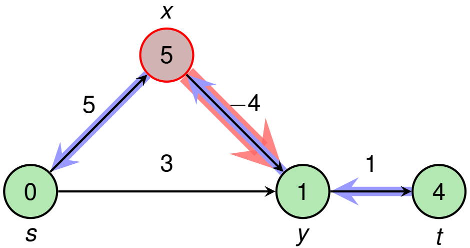
~~$(x, 5)$~~



Why negative-weight edges don't work

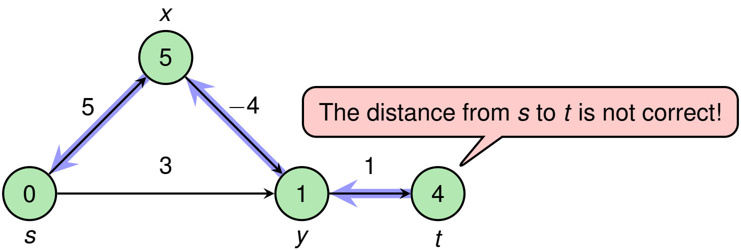
Priority Queue Q:

$\cancel{(x, 5)}$



Why negative-weight edges don't work

Priority Queue Q:



Summary of Single-Source Shortest Paths

Overview

- studied two algorithms for SSSP (single-source shortest path)
- basic operation: relaxing edges

Summary of Single-Source Shortest Paths

Overview

- studied two algorithms for SSSP (single-source shortest path)
- basic operation: relaxing edges

Bellman-Ford Algorithm

- detects negative-weight cycles
- V passes of relaxing all edges (arbitrary order)
- Runtime $\mathcal{O}(V \cdot E)$

Summary of Single-Source Shortest Paths

Overview

- studied two algorithms for SSSP (single-source shortest path)
- basic operation: relaxing edges

Bellman-Ford Algorithm

- detects negative-weight cycles
- V passes of relaxing all edges (arbitrary order)
- Runtime $\mathcal{O}(V \cdot E)$

Dijkstra's Algorithm

- requires non-negative weights
- Greeedy strategy to choose which edge to relax (similar to Prim)
- Using Fibonacci Heaps $\Rightarrow$ Runtime $\mathcal{O}(V \log V + E)$