6.4: Single-Source Shortest Paths

Frank Stajano  Thomas Sauerwald

Lent 2016
Dijkstra’s Algorithm
Historical Remarks

- 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)

“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

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

Final MST is given (implicitly) by the pointers!
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

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

Final MST is given (implicitly) by the pointers!
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

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

Final MST is given (implicitly) by the pointers!
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

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

Final MST is given (implicitly) by the pointers!
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

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

Final MST is given (implicitly) by the pointers!
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

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

Final MST is given (implicitly) by the pointers!
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

---

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

Final MST is given (implicitly) by the pointers!
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

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

Final MST is given (implicitly) by the pointers!
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

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

Final MST is given (implicitly) by the pointers!
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

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

Final MST is given (implicitly) by the pointers!
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

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

Final MST is given (implicitly) by the pointers!
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

![Graph Image]
Recap: Prim’s Algorithm

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

![Diagram of a graph with labeled edges and vertices. The edges forming the minimum spanning tree are highlighted in red.]
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

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

Final MST is given (implicitly) by the pointers!
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!

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

Final MST is given (implicitly) by the pointers!

6.4: Single-Source Shortest Paths
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 $A$.

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

Final MST is given (implicitly) by the pointers!
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!

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

Final MST is given (implicitly) by the pointers!

6.4: Single-Source Shortest Paths
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 ⇔ safe edge of cut \((V \setminus Q, Q)\)
  2. update keys and pointers of its neighbors in \( Q \)

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

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 ⇔ safe edge of cut \((V \setminus Q, Q)\)
  2. update keys and pointers of its neighbors in \( Q \)

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

Final MST is given (implicitly) by the pointers!

![Graph diagram showing Prim's MST algorithm](image-url)
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 ⇔ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$

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

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

![Diagram of a graph showing Prim's Algorithm in action](image-url)
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 ⇔ 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 $\iff$ 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 $\iff$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$

6.4: Single-Source Shortest Paths
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 ⇔ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$

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

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 ⇔ 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 $\iff$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$

---

**Diagram:**

A graph with vertices numbered and edges labeled with weights. Each vertex has a pointer to the minimum-weight edge to $V \setminus Q$. The process of selecting and updating vertices is illustrated through the steps of Prim's algorithm.
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 ⇔ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$

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

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 $Q$

We computed same MST as Kruskal, but in a completely different order!
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 ⇔ 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 ⇔ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$
Recap: Prim’s Algorithm

- 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 ⇔ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in Q

Implementation will be based on vertices!

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

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 ⇔ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$

![Graph Diagram]
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 $\iff$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$
Recap: Prim’s Algorithm

- 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 $Q$

![Graph diagram]

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

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 ⇔ 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 ⇔ **safe edge** of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$

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

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 ⇔ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$

We computed same MST as Kruskal, but in a completely different order!
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 $\iff$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$

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

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 $\iff$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$

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

![Graph diagram](image-url)
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 ⇔ safe edge of cut \((V \setminus Q, Q)\)
  2. update keys and pointers of its neighbors in \( Q \)

Implementation will be based on vertices!

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!

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

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 $\iff$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$

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

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 $\iff$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$

---

![Graph representation of Prim’s Algorithm](image)

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

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 $Q$

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

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 ⇔ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$

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

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 $Q$

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

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 $\iff$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$

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

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 $\iff$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$

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

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:
  - extract vertex from $Q$ with smallest key $\iff$ safe edge of cut $(V \setminus Q, Q)$
  - 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 \( Q \)

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

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 ⇔ **safe** edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$

---

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

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 $\iff$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$

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

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 $\iff$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$

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

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 $\iff$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$

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

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 $\iff$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$

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

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 ⇔ **safe** edge of cut \( (V \setminus Q, Q) \)
  2. **update** keys and pointers of its neighbors in \( Q \)

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

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 $\iff$ 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 $\iff$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$

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

Final MST is given (implicitly) by the pointers!
Prim’s Algorithms vs. Dijkstra’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 \( \infty \)).

Dijkstra’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 \( \infty \)) 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(G,w,s)
0: INITIALIZE(G,s)
1: S = \emptyset
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
```
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 $v.\delta$

```plaintext
DIJKSTRA(G,w,s)
  0: INITIALIZE(G,s)
  1: S = Ø
  2: Q = V
  3: while Q ≠ Ø 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
```

6.4: Single-Source Shortest Paths T.S. 20
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(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
```
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)
0: INITIALIZE(G,s)
1: S = ∅
2: Q = V
3: while Q \neq ∅ do
4: u = Extract-Min(Q)
5: S = S \cup {u}
6: for each v \in G.Adj[u] do
7: RELAX(u, v, w)
8: end for
9: end while
```

6.4: Single-Source Shortest Paths
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 $v$

![Diagram of Dijkstra'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 $u.\delta = u.d$
  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 Diagram]
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)

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

6.4: Single-Source Shortest Paths T.S. 20
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 $v$

6.4: Single-Source Shortest Paths T.S. 20
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)
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
```

6.4: Single-Source Shortest Paths  T.S. 20
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)
0: INITIALIZE(G,s)
1: S = \emptyset
2: Q = V
3: while Q \neq \emptyset do
4:  u = Extract-Min(Q)
5:  S = S \cup \{u\}
6:  for each v ∈ G.\text{Adj}[u] do
7:    RELAX(u,v,w)
8:  end for
9: end while
```
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)

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

Runtime w. Fibonacci Heaps

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

Prim’s algorithm has the same runtime!

6.4: Single-Source Shortest Paths
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: $u = \text{Extract-Min}(Q)$
5: $S = S \cup \{u\}$
6: for each $v \in G.\text{Adj}[u]$ do
7: $\text{RELAX}(u, v, w)$
8: end for
9: end while

**Runtime w. Fibonacci Heaps**

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

Prim’s algorithm has the same runtime!
Details of Dijkstra’s Algorithm

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

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

**Runtime w. Fibonacci Heaps**
- **Initialization** (l. 0-2): $\mathcal{O}(V)$

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

Prim’s algorithm has the same runtime!
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:     \( u = \text{Extract-Min}(Q) \)
5:     \( S = S \cup \{u\} \)
6:     for each \( v \in G.Adj[u] \) do
7:         \( \text{RELAX}(u,v,w) \)
8:     end for
9: end while
```

**Runtime w. Fibonacci Heaps**
- **Initialization** (l. 0-2): \( O(V) \)

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

Prim’s algorithm has the same runtime!
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: $u = $ Extract-Min($Q$)
   5: $S = S \cup \{u\}$
   6: for each $v \in G.\text{Adj}[u]$ 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): $O(V)$
- ExtractMin (l. 4): $O(V \cdot \log V)$
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: $S = \emptyset$
2: $Q = V$
3: while $Q \neq \emptyset$ do
4: $u = \text{Extract-Min}(Q)$
5: $S = S \cup \{u\}$
6: for each $v \in G.\text{Adj}[u]$ do
7: RELAX($u, v, w$)
8: end for
9: end while

**Runtime w. Fibonacci Heaps**
- Initialization (l. 0-2): $O(V)$
- ExtractMin (l. 4): $O(V \cdot \log V)$
- DecreaseKey (l. 7): $O(E \cdot 1)$

With a binary heap instead, the overall runtime would be $O(E \cdot \log V)$!
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: $S = \emptyset$
2: $Q = V$
3: while $Q \neq \emptyset$ do
4: $u = \text{Extract-Min}(Q)$
5: $S = S \cup \{u\}$
6: for each $v \in G.Adj[u]$ do
7: $\text{RELAX}(u, v, w)$
8: end for
9: end while

Runtime w. Fibonacci Heaps
- Initialization (l. 0-2): $O(V)$
- ExtractMin (l. 4): $O(V \cdot \log V)$
- DecreaseKey (l. 7): $O(E \cdot 1)$
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: $u = \text{Extract-Min}(Q)$
5: $S = S \cup \{u\}$
6: for each $v \in G.\text{Adj}[u]$ do
7: RELAX($u, v, w$)
8: end for
9: end while

**Runtime w. Fibonacci Heaps**

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

$\Rightarrow$ Overall: $O(V \log V + E)$
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: $S = \emptyset$
2: $Q = V$
3: while $Q \neq \emptyset$ do
   4: $u = \text{Extract-Min}(Q)$
   5: $S = S \cup \{u\}$
   6: for each $v \in G.\text{Adj}[u]$ do
      7: RELAX($u, v, w$)
   8: end for
9: end while

Runtime w. Fibonacci Heaps
- Initialization (l. 0-2): $O(V)$
- ExtractMin (l. 4): $O(V \cdot \log V)$
- DecreaseKey (l. 7): $O(E \cdot 1)$
$\Rightarrow$ Overall: $O(V \log V + E)$

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 $Q$: 
$(s, 0), (t, \infty), (x, \infty), (y, \infty), (z, \infty)$
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 $Q$: $(s, 0), (t, 10), (x, \infty), (y, \infty), (z, \infty)$
Execution of Dijkstra (Figure 24.6)

Priority Queue $Q$:

$(s, 0), (t, 10), (x, \infty), (y, \infty), (z, \infty)$
Execution of Dijkstra (Figure 24.6)

Priority Queue $Q$:

$(s, 0), (t, 10), (x, \infty), (y, 5), (z, \infty)$
Execution of Dijkstra (Figure 24.6)

Priority Queue $Q$:

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

![Graph showing execution of Dijkstra algorithm]
Execution of Dijkstra (Figure 24.6)

Priority Queue $Q$: 

$(t, 10), (x, \infty), (y, 5), (z, \infty)$
Execution of Dijkstra (Figure 24.6)

Priority Queue \( Q \):

\((t, 10), (x, \infty), (y, 5), (z, \infty)\)
Execution of Dijkstra (Figure 24.6)

Priority Queue $Q$:

$(t, 8), (x, \infty), (y, 5), (z, \infty)$
Execution of Dijkstra (Figure 24.6)

Priority Queue \( Q \):

\((t, 8), (x, \infty), (y, 5), (z, \infty)\)
Execution of Dijkstra (Figure 24.6)

Priority Queue $Q$:

$(t, 8), (x, 14), (y, 5), (z, \infty)$
Execution of Dijkstra (Figure 24.6)

Priority Queue $Q$:

$(t, 8), (x, 14), (y, 5), (z, \infty)$
Execution of Dijkstra (Figure 24.6)

Priority Queue $Q$:

$(t, 8), (x, 14), (y, 5), (z, 7)$
Execution of Dijkstra (Figure 24.6)

Priority Queue $Q$: $(t, 8), (x, 14), (y, 5), (z, 7)$
Execution of Dijkstra (Figure 24.6)

Priority Queue $Q$: 

$((t, 8), (x, 14), (z, 7))$
Execution of Dijkstra (Figure 24.6)

Priority Queue $Q$: 

$(t, 8), (x, 14), (z, 7)$
Execution of Dijkstra (Figure 24.6)

Priority Queue $Q$: 

$(t, 8), (x, 14), (z, 7)$
Execution of Dijkstra (Figure 24.6)

Priority Queue $Q$:

$(t, 8), (x, 14), (z, 7)$
Execution of Dijkstra (Figure 24.6)

Priority Queue $Q$:

$(t, 8), (x, 13), (z, 7)$
Execution of Dijkstra (Figure 24.6)

Priority Queue $Q$:

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

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

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

$((s, 0), (t, 10), (x, \infty), (y, 5), (z, 7))$

$((s, 0), (t, 10), (x, 13), (y, 5), (z, 7))$

$((s, 0), (t, 8), (x, 13), (y, 5), (z, 7))$
Execution of Dijkstra (Figure 24.6)

Priority Queue $Q$:

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

$(t, 8), (x, 13)$
Execution of Dijkstra (Figure 24.6)

Priority Queue $Q$:

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

$(s, 0), (t, 8), (x, 13)$
Execution of Dijkstra (Figure 24.6)

Priority Queue $Q$: 

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

$(((s, 0), (t, 10), (x, \infty), (y, 5), (z, \infty))), ((t, 8), (x, 13))$
Execution of Dijkstra (Figure 24.6)

Priority Queue $Q$:

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

- $(s, 0)$, $(t, 10)$
- $(x, \infty)$
- $(y, 5)$
- $(z, \infty)$

- $(t, 8)$
- $(x, 9)$

Graph showing the execution of Dijkstra's algorithm with nodes and edges labeled with weights.
Execution of Dijkstra (Figure 24.6)

Priority Queue $Q$:

$((t, 8), (x, 9))$
Execution of Dijkstra (Figure 24.6)

Priority Queue $Q$:

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

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

$(s, 0), (t, 10), (x, \infty), (y, 5), (z, 7)$

$(s, 0), (t, 8), (x, 14), (y, 5), (z, 7)$
Execution of Dijkstra (Figure 24.6)

Priority Queue $Q$: $(s, 0)$, $(t, \infty)$, $(x, \infty)$, $(y, \infty)$, $(z, \infty)$

(s, 0), (t, 10), (x, \infty), (y, 5), (z, \infty)
Execution of Dijkstra (Figure 24.6)

Priority Queue $Q$:

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

- $(s, 0)$, $(t, 10)$, $(x, \infty)$, $(y, 5)$, $(z, \infty)$
- $(t, 1)$, $(x, 6)$, $(y, 5)$, $(z, 7)$
- $(x, 8)$, $(y, 8)$, $(z, 7)$
- $(z, 9)$, $(x, 9)$, $(y, 9)$
Execution of Dijkstra (Figure 24.6)

Priority Queue $Q$: 

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

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

$$((s, 0), (t, 10), (x, \infty), (y, 5), (z, 7))$$

$$((t, 8), (x, 14), (y, 5), (z, 7))$$

$$((t, 8), (x, 14), (y, 5), (z, 7))$$
Execution of Dijkstra (Figure 24.6)

Priority Queue $Q$: 

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

---

**Figure 24.6:** Single-Source Shortest Paths
Dijkstra’s Algorithm: Correctness

Correctness (Theorem 24.6)

For any directed graph $G = (V, E)$ with non-negative edge weights $w : E \rightarrow \mathbb{R}^+$ 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 $w : E \rightarrow \mathbb{R}^+$ 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,
  
  $u.d > u.\delta$
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
Dijkstra’s Algorithm: Correctness

Correctness (Theorem 24.6)

For any directed graph \( G = (V, E) \) with non-negative edge weights \( w : E \rightarrow \mathbb{R}^+ \) 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 \( V \setminus S \)
Dijkstra’s Algorithm: Correctness

Correctness (Theorem 24.6)

For any directed graph \( G = (V, E) \) with non-negative edge weights \( w : E \rightarrow \mathbb{R}^+ \) 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 \( V \setminus S \)

\[ u.d \]
Correctness (Theorem 24.6)

For any directed graph \( G = (V, E) \) with non-negative edge weights \( w : E \rightarrow \mathbb{R}^{+} \) 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 \( V \setminus S \)

\[ u.d \leq y.d \]
Dijkstra’s Algorithm: Correctness

Correctness (Theorem 24.6)

For any directed graph $G = (V, E)$ with non-negative edge weights $w : E \rightarrow \mathbb{R}^+$ 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 $V \setminus S$

  $u.d \leq y.d$

  $u$ is extracted before $y$
Dijkstra’s Algorithm: Correctness

Correctness (Theorem 24.6)

For any directed graph $G = (V, E)$ with non-negative edge weights $w : E \rightarrow \mathbb{R}^+$ 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 $V \setminus S$

  \[ u.d \leq y.d = y.\delta \]
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 \( V \setminus S \)
  \[ u.d \leq y.d = y.\delta \]
  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 $w : E \rightarrow \mathbb{R}^+$ 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: 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 \( V \setminus S \)

\[ u.\delta < u.d \leq y.d = y.\delta \]

This contradicts that \( y \) is on a shortest path from \( s \) to \( u \).
Dijkstra’s Algorithm: Correctness

Correctness (Theorem 24.6)

For any directed graph $G = (V, E)$ with non-negative edge weights $w : E \rightarrow \mathbb{R}^+$ 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 $V \setminus S$

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

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

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

---

The distance from $s$ to $t$ is not correct!
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 $Q$: 
$(s, 0), (t, \infty), (x, 5), (y, \infty)$

The distance from $s$ to $t$ is not correct!
Why negative-weight edges don’t work

Priority Queue $Q$: 

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

The distance from $s$ to $t$ is not correct!
Why negative-weight edges don’t work

Priority Queue $Q$:

$\not(s, 0), (t, \infty), (x, 5), (y, 3)$
Why negative-weight edges don’t work

Priority Queue $Q$: 

$(s, 0), (t, \infty), (x, 5), (y, 3)$
Why negative-weight edges don’t work

Priority Queue \( Q \):

\((t, \infty), (x, 4), (y, 3)\)

The distance from \( s \) to \( t \) is not correct!
Why negative-weight edges don’t work

Priority Queue $Q$: 

$(t, \infty), (x, 4), (y, 3)$

The distance from $s$ to $t$ is not correct!
Why negative-weight edges don’t work

Priority Queue $Q$: $(t, 4), (x, 5), (y, 3)$

The distance from $s$ to $t$ is not correct!
Why negative-weight edges don’t work

Priority Queue $Q$: 

$(t, 4), (x, 5), (y, 3)$

The distance from $s$ to $t$ is not correct!
Why negative-weight edges don’t work

Priority Queue $Q$:

$$(t, 4), (x, 5)$$

The distance from $s$ to $t$ is not correct!
Why negative-weight edges don’t work

Priority Queue $Q$:

$(t, 4), (x, 5)$

The distance from $s$ to $t$ is not correct!
Why negative-weight edges don’t work

Priority Queue $Q$: 

$(x, 5)$

The distance from $s$ to $t$ is not correct!
Why negative-weight edges don’t work

Priority Queue $Q$:

$(x, 5)$

The distance from $s$ to $t$ is not correct!
Priority Queue $Q$: 

$((x, 5))$
Why negative-weight edges don’t work

Priority Queue $Q$:

![Diagram showing a graph with nodes s, 0, 1, 4, and t, and edges with weights 5, 3, -4, 1, and 1, highlighting the distance from s to t is not correct!]

The distance from s to t is not correct!
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 $O(V \cdot E)$

Dijkstra’s Algorithm

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