CSCB63 Winter 2021

Week 6 Lecture 1

Dijkstra's Shortest Path Algorithm

Anna Bretscher

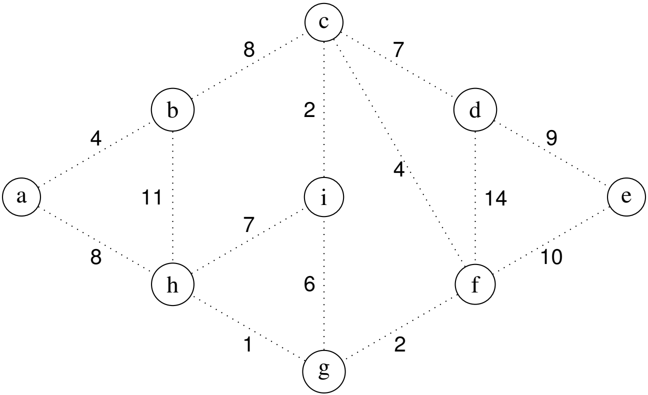
March 16, 2021

TODAY

Review Prim's Algorithm

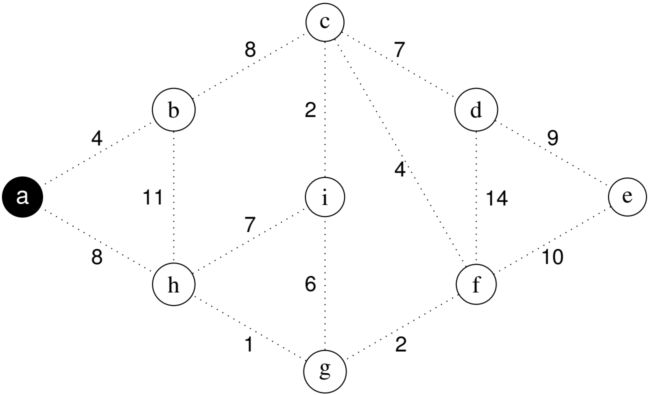
Dijkstra's Algorithm for single source shortest paths

Prim's Algorithm: A Few Example Steps



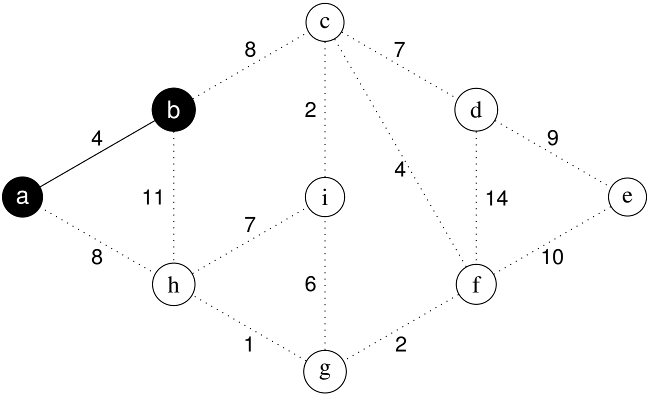
vertex	a	b	c	d	e	f	g	h	i
priority	0	∞	∞	∞	∞	∞	∞	∞	∞
pred									

Prim's Algorithm: A Few Example Steps



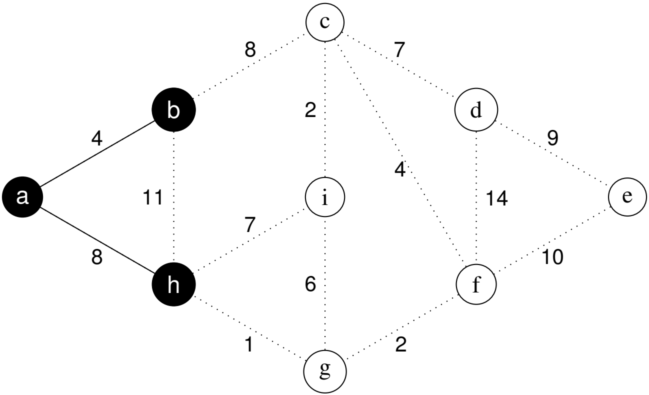
vertex	b	h	c	d	e	f	g	i
priority	4	8	∞	∞	∞	∞	∞	∞
pred	a	a						

Prim's Algorithm: A Few Example Steps



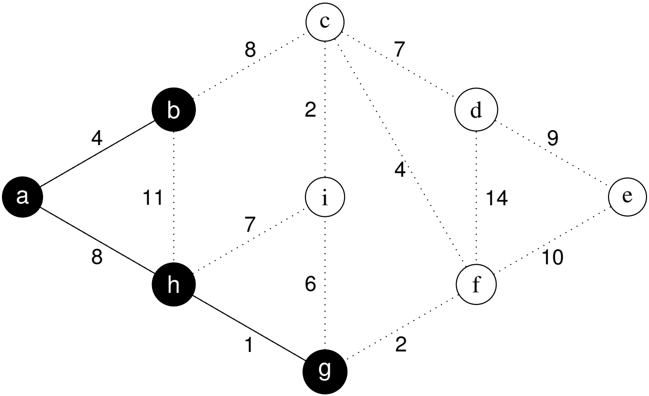
vertex	h	c	d	e	f	g	i
priority	8	8	∞	∞	∞	∞	∞
pred	a	b					

Prim's Algorithm: A Few Example Steps



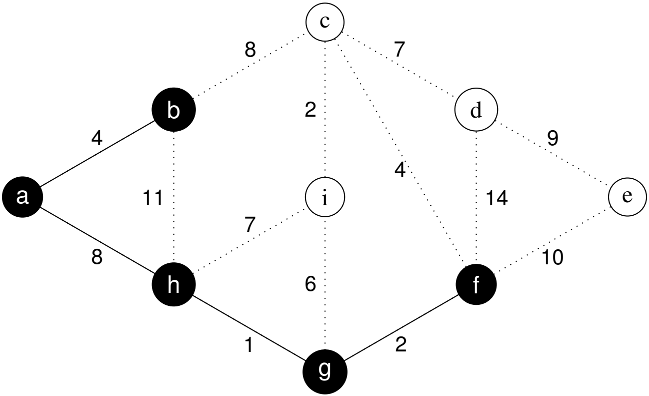
vertex	g	i	c	d	e	f
priority	1	7	8	∞	∞	∞
pred	h	h	b			

Prim's Algorithm: A Few Example Steps



vertex	f	i	c	d	e
priority	2	6	8	∞	∞
pred	g	g	b		

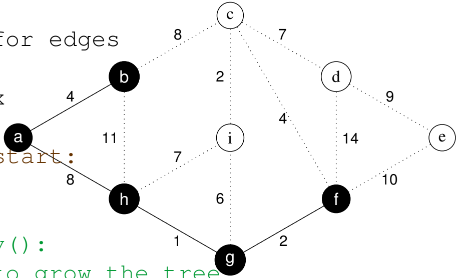
Prim's Algorithm: A Few Example Steps



vertex	c	i	e	d
priority	4	6	10	14
pred	f	g	f	f

PRIM'S ALGORITHM

```
Prim(V, E)
    S := new container() for edges
    PQ := new min-heap()
    start := pick a vertex
    PQ.insert(start, 0)
     for each vertex v ≠ start:
        # initialize pq
        PQ.insert(v, ∞)
     while not PQ.is_empty():
        # add least edge to grow the tree
         u := PQ.extract_min()
         S.add({u.pred, u})
         for each z in u's adjacency list:
            # update priorities based on u now in S
            if z in PQ && weight(u,z) < priority of z:
                PQ.decrease_priority(z, weight(u,z))
                z.pred := u
    return S
```



Prim's Algorithm Time Complexity

Q. How many times does a *vertex* enter/leave the *min-heap*?

Prim's Algorithm Time Complexity

Q. How many times does a *vertex* enter/leave the *min-heap*?

A. Every *vertex* enters and leaves min-heap *once*: $\Theta(\lg n)$ per vertex, totalling $\Theta(n \lg n)$

Prim's Algorithm Time Complexity

Q. How many times does a *vertex* enter/leave the *min-heap*?

A. Every *vertex* enters and leaves min-heap *once*: $\Theta(\lg n)$ per vertex, totalling $\Theta(n \lg n)$

Q. How many times can a *vertex's priority* decrease?

Prim's Algorithm Time Complexity

Q. How many times does a *vertex* enter/leave the *min-heap*?

A. Every *vertex* enters and leaves min-heap *once*: $\Theta(\lg n)$ per vertex, totalling $\Theta(n \lg n)$

Q. How many times can a *vertex's priority* decrease?

A. Every edge may trigger a change of priority: so $\forall v \in V, O(deg(v))$ which is $O(m)$ and takes $O(\lg n)$ for a total of $O(mlogn)$.

Prim's Algorithm Time Complexity

Q. How many times does a *vertex* enter/leave the *min-heap*?

A. Every *vertex* enters and leaves min-heap *once*: $\Theta(\lg n)$ per vertex, totalling $\Theta(n \lg n)$

Q. How many times can a *vertex's priority* decrease?

A. Every edge may trigger a change of priority: so $\forall v \in V, O(deg(v))$ which is $O(m)$ and takes $O(\lg n)$ for a total of $O(mlogn)$.

- Everything else, can be done in $\Theta(1)$ per *vertex* or per *edge*

Prim's Algorithm Time Complexity

Q. How many times does a *vertex* enter/leave the *min-heap*?

A. Every *vertex* enters and leaves min-heap *once*: $\Theta(\lg n)$ per vertex, totalling $\Theta(n \lg n)$

Q. How many times can a *vertex's priority* decrease?

A. Every edge may trigger a change of priority: so $\forall v \in V, O(deg(v))$ which is $O(m)$ and takes $O(\lg n)$ for a total of $O(mlogn)$.

- Everything else, can be done in $\Theta(1)$ per *vertex* or per *edge*
- Total $O((n+m)\lg n)$ time worst case.

Prim's Correctness Proof

To begin with we will first prove a useful property:

Cut Property: Let S be a nontrivial subset of V in G (i.e. $S \neq \emptyset$ and $S \neq V$). If (u, v) is the *lowest-cost edge* crossing $(S, V - S)$, then (u, v) is in *every MST* of G.

Proof.

- Suppose there exists an *MST* T that does not contain (u, v).

Prim's Correctness Proof

To begin with we will first prove a useful property:

Cut Property: Let S be a nontrivial subset of V in G (i.e. $S \neq \emptyset$ and $S \neq V$). If (u, v) is the *lowest-cost edge* crossing $(S, V - S)$, then (u, v) is in *every MST* of G.

Proof.

- Suppose there exists an *MST* T that does not contain (u, v).
- Consider the sets S and $V - S$.

Prim's Correctness Proof

To begin with we will first prove a useful property:

Cut Property: Let S be a nontrivial subset of V in G (i.e. $S \neq \emptyset$ and $S \neq V$). If (u, v) is the *lowest-cost edge* crossing $(S, V - S)$, then (u, v) is in *every MST* of G.

Proof.

- Suppose there exists an *MST* T that does not contain (u, v).
- Consider the sets S and $V - S$.
- There must exist a *path* from u to v.

Prim's Correctness Proof

To begin with we will first prove a useful property:

Cut Property: Let S be a nontrivial subset of V in G (i.e. $S \neq \emptyset$ and $S \neq V$). If (u, v) is the *lowest-cost edge* crossing $(S, V - S)$, then (u, v) is in *every MST* of G.

Proof.

- Suppose there exists an *MST* T that does not contain (u, v).
- Consider the sets S and $V - S$.
- There must exist a *path* from u to v.
- On this path, there must exist an *edge* e that *crosses* between $V - S$ into S.

To begin with we will first prove a useful property:

Cut Property: Let S be a nontrivial subset of V in G (i.e. $S \neq \emptyset$ and $S \neq V$). If (u, v) is the *lowest-cost edge* crossing $(S, V - S)$, then (u, v) is in *every MST* of G.

Proof.

- Suppose there exists an *MST* T that does not contain (u, v).
- Consider the sets S and $V - S$.
- There must exist a *path* from u to v.
- On this path, there must exist an *edge* e that *crosses* between $V - S$ into S.
- Since (u, v) is the *least weight edge* crossing between V and $S - V$, swapping (u, v) with e will reduce the weight of T.

Prim's Correctness Proof

To begin with we will first prove a useful property:

Cut Property: Let S be a nontrivial subset of V in G (i.e. $S \neq \emptyset$ and $S \neq V$). If (u, v) is the *lowest-cost edge* crossing $(S, V - S)$, then (u, v) is in *every MST* of G.

Proof.

- Suppose there exists an *MST* T that does not contain (u, v).
- Consider the sets S and $V - S$.
- There must exist a *path* from u to v.
- On this path, there must exist an *edge* e that *crosses* between $V - S$ into S.
- Since (u, v) is the *least weight edge* crossing between V and $S - V$, swapping (u, v) with e will reduce the weight of T.
- Therefore, T is not an *MST*.

Correctness of Prim's Algorithm

The correctness of *Prim's* Algorithm follows...from the **Cut Property**.

Q. How does the argument go?

A.

Correctness of Prim's Algorithm

The correctness of *Prim's* Algorithm follows...from the **Cut Property**.

Q. How does the argument go?

A.

- Consider *optimal* MST O and *Prim's Algorithm* tree T.

Correctness of Prim's Algorithm

The correctness of *Prim's* Algorithm follows...from the **Cut Property**.

Q. How does the argument go?

A.

- Consider *optimal* MST O and *Prim's Algorithm* tree T.
- Order edges of T according to *order* they are selected.

Correctness of Prim's Algorithm

The correctness of *Prim's* Algorithm follows...from the **Cut Property**.

Q. How does the argument go?

A.

- Consider *optimal* MST O and *Prim's Algorithm* tree T.
- Order edges of T according to *order* they are selected.
- Consider the *first edge* $e = (u, v)$ in the ordering that is in T but not in O.

Correctness of Prim's Algorithm

The correctness of *Prim's* Algorithm follows...from the **Cut Property**.

Q. How does the argument go?

A.

- Consider *optimal* MST O and *Prim's Algorithm* tree T.
- Order edges of T according to *order* they are selected.
- Consider the *first edge* $e = (u, v)$ in the ordering that is in T but not in O.
- At the stage of *Prim's* when e was added there was a set S of vertices such that $u \in S$, $v \in V - S$.

Correctness of Prim's Algorithm

The correctness of *Prim's* Algorithm follows...from the **Cut Property**.

Q. How does the argument go?

A.

- Consider *optimal* MST O and *Prim's Algorithm* tree T.
- Order edges of T according to *order* they are selected.
- Consider the *first edge* $e = (u, v)$ in the ordering that is in T but not in O.
- At the stage of *Prim's* when e was added there was a set S of vertices such that $u \in S$, $v \in V - S$.

Case 1: Edge weights are *unique* so by the **Cut Property**, e must belong to O. Therefore consider when *edge weights* are *not unique*.

CORRECTNESS OF PRIM'S ALGORITHM

CASE 2: Edge weights not *unique*.

Correctness of Prim's Algorithm

Case 2: Edge weights not *unique*.

- $e \notin O$, there exists a path p from u to v such that an edge $e' = (x, y)$ exists on p and $x \in S$ and $y \in V - S$.

Correctness of Prim's Algorithm

Case 2: Edge weights not *unique*.

- $e \notin O$, there exists a path p from u to v such that an edge $e' = (x, y)$ exists on p and $x \in S$ and $y \in V - S$.

Case 2a: $w(e') = w(e)$, so swap e' with e and the tree will still span tree G and be minimal.

Correctness of Prim's Algorithm

Case 2: Edge weights not *unique*.

- $e \notin O$, there exists a path p from u to v such that an edge $e' = (x, y)$ exists on p and $x \in S$ and $y \in V - S$.

Case 2a: $w(e') = w(e)$, so swap e' with e and the tree will still span tree G and be minimal.

Case 2b: $w(e') \neq w(e)$. Must be that $w(e') \nless w(e)$ since then *Prim's* algorithm would have chosen it.

Correctness of Prim's Algorithm

Case 2: Edge weights not *unique*.

- $e \notin O$, there exists a path p from u to v such that an edge $e' = (x, y)$ exists on p and $x \in S$ and $y \in V - S$.

Case 2a: $w(e') = w(e)$, so swap e' with e and the tree will still span tree G and be minimal.

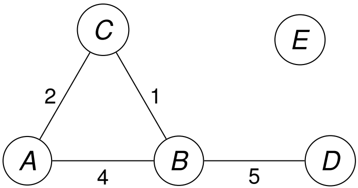
Case 2b: $w(e') \neq w(e)$. Must be that $w(e') \nless w(e)$ since then *Prim's* algorithm would have chosen it.

- If $w(e') > w(e)$ then swapping e' with e reduces the weight of O, which is a contradiction.

Common Task #2 on Weighted Graphs

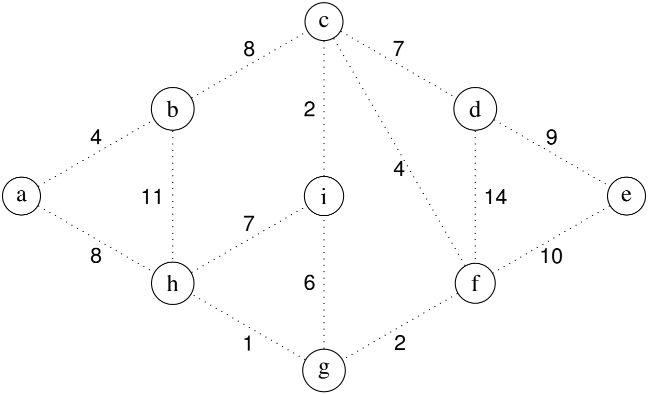
Single Source Shortest Paths

- Find a *simple path* between two vertices (if any).
- Minimize the *sum of the weights* of the edges used.



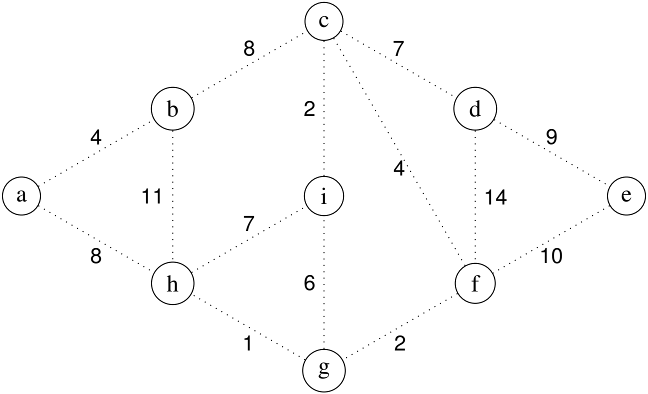
From A to D:
$\langle A, C, B, D \rangle$ is a shortest path. Total weight 8. ✓
$\langle A, B, D \rangle$ is *not* a shortest path. Total weight 9. ✗

Brain Storming: Single Source Shortest Paths



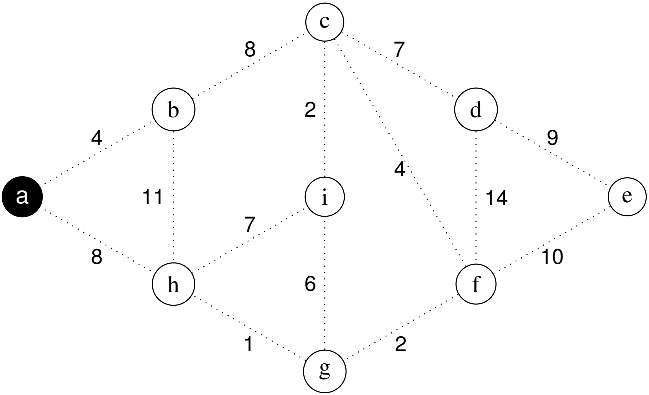
Given a *start vertex*, find the *shortest paths* to all other *vertices*. Ideas?

DIJKSTRA'S ALGORITHM: A FEW ITERATIONS



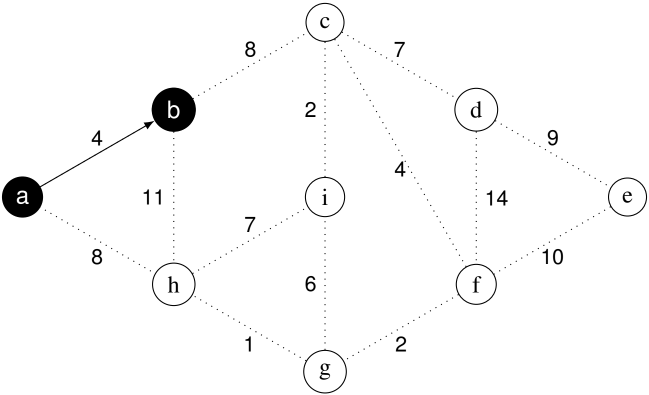
vertex	a	b	c	d	e	f	g	h	i
priority	0	∞	∞	∞	∞	∞	∞	∞	∞
pred									

DIJKSTRA'S ALGORITHM: A FEW ITERATIONS



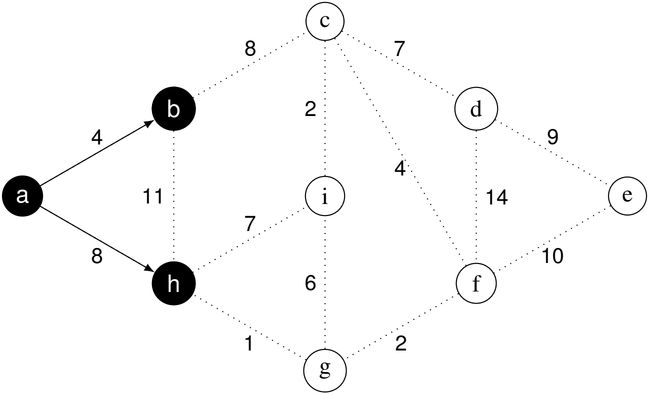
vertex	b	h	c	d	e	f	g	i
priority	4	8	∞	∞	∞	∞	∞	∞
pred	a	a						

Dijkstra's Algorithm: A Few Iterations



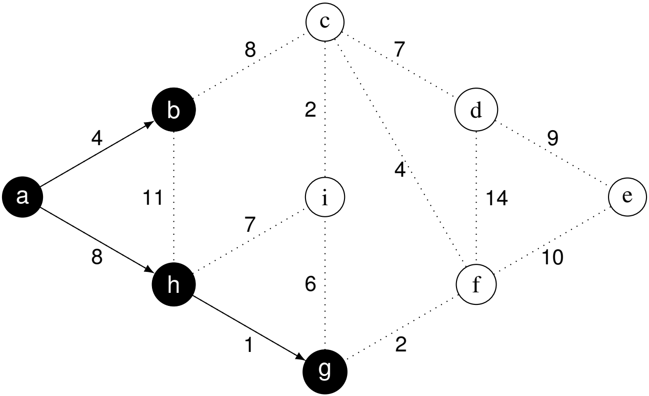
vertex	h	c	d	e	f	g	i
priority	8	12	∞	∞	∞	∞	∞
pred	a	b					

Dijkstra's Algorithm: A Few Iterations



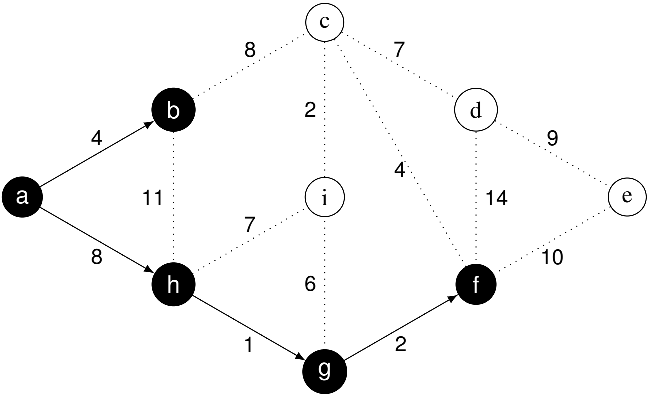
vertex	g	c	i	d	e	f
priority	9	12	15	∞	∞	∞
pred	h	b	h			

Dijkstra's Algorithm: A Few Iterations



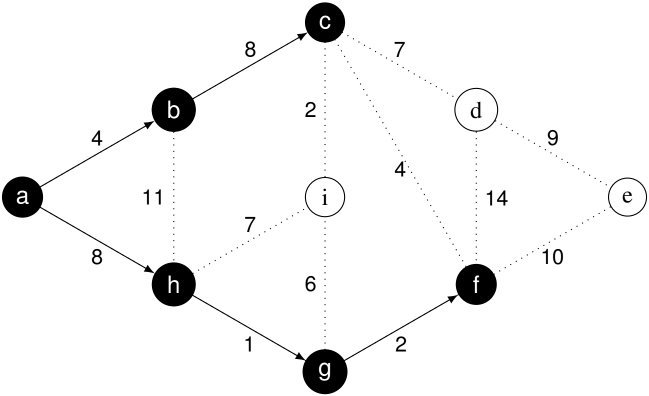
vertex	f	c	i	d	e
priority	11	12	15	∞	∞
pred	g	b	h		

Dijkstra's Algorithm: A Few Iterations



vertex	c	i	e	d
priority	12	15	21	25
pred	b	h	f	f

Dijkstra's Algorithm: A Few Iterations



vertex	i	d	e
priority	14	19	21
pred	c	c	f

Greedy Algorithm #3

We add our *start vertex s* to the set of *reached vertices S* and give it *distance* $d[s] = 0$.

This creates a *distance tree* rooted at *s*.

Q. What is the *greedy rule* that we follow?

A.

Greedy Algorithm #3

We add our *start vertex* s to the set of *reached vertices* S and give it *distance* $d[s] = 0$.

This creates a *distance tree* rooted at s.

Q. What is the *greedy rule* that we follow?

A. At each stage we consider the *next closest vertex* to s from vertices *not in* S, or alternatively, the vertex with next *shortest path* to s...

Q. How do we use a *priority queue* to determine the shortest path so far?

A.

Greedy Algorithm #3

We add our *start vertex* s to the set of *reached vertices* S and give it *distance* $d[s] = 0$.

This creates a *distance tree* rooted at s.

Q. What is the *greedy rule* that we follow?

A. At each stage we consider the *next closest vertex* to s from vertices *not in* S, or alternatively, the vertex with next *shortest path* to s...

Q. How do we use a *priority queue* to determine the shortest path so far?

A. When a new vertex v is added to S,

- consider each neighbour u of v such that $u \notin S$

Greedy Algorithm #3

We add our *start vertex* s to the set of *reached vertices* S and give it *distance* $d[s] = 0$.

This creates a *distance tree* rooted at s.

Q. What is the *greedy rule* that we follow?

A. At each stage we consider the *next closest vertex* to s from vertices *not in* S, or alternatively, the vertex with next *shortest path* to s...

Q. How do we use a *priority queue* to determine the shortest path so far?

A. When a new vertex v is added to S,

- consider each neighbour u of v such that $u \notin S$
- update the current best distance (priority $p[u]$) to $d[v] + w(v, u)$ if it's better.

DIJKSTRA'S ALGORITHM

```
dijkstra(G, s)
   PQ := new min-heap()
   PQ.insert(s, 0)
   d[s] := 0
   for each vertex z ≠ s:
      # initialize priority queue
      PQ.insert(z, ∞)
      d[z] := ∞
   while PQ not empty:
      #greedy choice of vertex to grow shortest path tree
      v := Q.extract-min()
      for each u in v's adjacency list:
         #Update priorities of adjacent nodes
         if d[v] + w({v,u}) < d[u]:
            PQ.decrease-priority(u, d[v] + w({v,u}))
            d[u] := d[v] + w({v,u})
            pred[u]:= v
```

Dijkstra Correctness

- Let T_s be the *distance tree* constructed by *Dijkstra's Algorithm* starting at s.

Dijkstra Correctness

- Let T_s be the *distance tree* constructed by *Dijkstra's Algorithm* starting at s.
- Let O_s be an *optimal distance tree* rooted at s.

DIJKSTRA CORRECTNESS

- Let T_s be the *distance tree* constructed by *Dijkstra's Algorithm* starting at s.
- Let O_s be an *optimal distance tree* rooted at s.
- Order the edges $\langle e_1, e_2, \ldots, e_m \rangle$ according to how they are added to T_s.

Dijkstra Correctness

- Let T_s be the *distance tree* constructed by *Dijkstra's Algorithm* starting at s.
- Let O_s be an *optimal distance tree* rooted at s.
- Order the edges $\langle e_1, e_2, \ldots, e_m \rangle$ according to how they are added to T_s.
- Consider the first edge $e_i = (u, v)$ such that $e_i \in T_s$ and $e_i \notin O_s$.

Dijkstra Correctness

- Let T_s be the *distance tree* constructed by *Dijkstra's Algorithm* starting at s.
- Let O_s be an *optimal distance tree* rooted at s.
- Order the edges $\langle e_1, e_2, \ldots, e_m \rangle$ according to how they are added to T_s.
- Consider the first edge $e_i = (u, v)$ such that $e_i \in T_s$ and $e_i \notin O_s$.
- Then $e_1, \ldots, e_{i-1} \in T_s$. Let S be the set of vertices added so far (ie, all endpoints of $\langle e_1 \ldots e_{i-1} \rangle$).

Dijkstra Correctness

- Let T_s be the *distance tree* constructed by *Dijkstra's Algorithm* starting at s.
- Let O_s be an *optimal distance tree* rooted at s.
- Order the edges $\langle e_1, e_2, \ldots, e_m \rangle$ according to how they are added to T_s.
- Consider the first edge $e_i = (u, v)$ such that $e_i \in T_s$ and $e_i \notin O_s$.
- Then $e_1, \ldots, e_{i-1} \in T_s$. Let S be the set of vertices added so far (ie, all endpoints of $\langle e_1 \ldots e_{i-1} \rangle$).
- Each node in S has *minimum path distance* to s, the start vertex.

Dijkstra Correctness

- Let T_s be the *distance tree* constructed by *Dijkstra's Algorithm* starting at s.
- Let O_s be an *optimal distance tree* rooted at s.
- Order the edges $\langle e_1, e_2, \ldots, e_m \rangle$ according to how they are added to T_s.
- Consider the first edge $e_i = (u, v)$ such that $e_i \in T_s$ and $e_i \notin O_s$.
- Then $e_1, \ldots, e_{i-1} \in T_s$. Let S be the set of vertices added so far (ie, all endpoints of $\langle e_1 \ldots e_{i-1} \rangle$).
- Each node in S has *minimum path distance* to s, the start vertex.
- Since $(u, v) \notin O_s$ it must be that there is some other shorter path p from s to v.

Dijkstra Correctness

- Consider the edge $e_j = (x, y), j > i$ on p that has one endpoint in S and one in $V - S$.

Dijkstra Correctness

- Consider the edge $e_j = (x, y), j > i$ on p that has one endpoint in S and one in $V - S$.

Case 1: $y \neq v$, $d[y] < d[v]$ and our algorithm would have chosen e_j next and not e_i.

Dijkstra Correctness

- Consider the edge $e_j = (x, y), j > i$ on p that has one endpoint in S and one in $V - S$.

CASE 1: $y \neq v$, $d[y] < d[v]$ and our algorithm would have chosen e_j next and not e_i.

CASE 2A: If $y = v$ and $d_O[y] < d_T[v]$ Dijkstra would have selected e_j rather than e_i.

Dijkstra Correctness

- Consider the edge $e_j = (x, y), j > i$ on p that has one endpoint in S and one in $V - S$.

Case 1: $y \neq v$, $d[y] < d[v]$ and our algorithm would have chosen e_j next and not e_i.

Case 2a: If $y = v$ and $d_O[y] < d_T[v]$ Dijkstra would have selected e_j rather than e_i.

Case 2b: Therefore, $y = v$ and $d_O[y] = d_T[v]$, so we can swap (x, v) with (u, v) in O_s and O_s is now closer to T_s.

DIJKSTRA CORRECTNESS

- Consider the edge $e_j = (x, y), j > i$ on p that has one endpoint in S and one in $V - S$.

CASE 1: $y \neq v$, $d[y] < d[v]$ and our algorithm would have chosen e_j next and not e_i.

CASE 2A: If $y = v$ and $d_O[y] < d_T[v]$ Dijkstra would have selected e_j rather than e_i.

CASE 2B: Therefore, $y = v$ and $d_O[y] = d_T[v]$, so we can swap (x, v) with (u, v) in O_s and O_s is now closer to T_s.

★ One can also prove this by *induction*

Generalized Abstract Weights

Dijkstra's algorithm and *correctness* need only the following properties:

	Dijkstra	**General**	Purpose
total order	$\leq$	$\sqsubseteq$	Path weight comparison
extremes	$0 \leq \text{weight} \leq \infty$	$\bot \sqsubseteq \text{weight} \sqsubseteq \top$	Initialize values etc
associative op.	$+$	$\oplus$	sum weights $\rightarrow$ path weight
identity	$w + 0 = w = 0 + w$	$w \oplus \bot = w = \bot \oplus w$	
monotonic	if $w \leq w'$ then:	if $w \sqsubseteq w'$ then:	
	$s + w \leq s + w'$ and	$s \oplus w \sqsubseteq s \oplus w'$ and	
	$w + s \leq w' + s$	$w \oplus s \sqsubseteq w' \oplus s$	

Use creative choices of *weights*, *total order*, and *associative operator* to solve other problems!