CS217 - Data Structures & Algorithm Analysis (DSAA)

Lecture #15

▶Minimum spanning trees and shortest paths

Prof. Pietro S. Oliveto
Department of Computer Science and Engineering
Southern University of Science and Technology (SUSTech)

olivetop@sustech.edu.cn
https://faculty.sustech.edu.cn/olivetop

Reading: Chapter 21 and Section 22.3

Aims for this lecture

- To introduce the minimum spanning tree problem.
- To see two different greedy approaches for solving it: Kruskal's algorithm and Prim's algorithm.
- To briefly review variants of shortest path problems.
- To cover Dijkstra's algorithm for solving the single-source shortest path problem.
  - Another example for greediness and dynamic programming
- To show how efficient data structures can be used to guarantee efficient runtimes.

▶Minimum spanning trees

- Suppose we want to supply n newly built houses with electricity, using the minimum length of wire.
- Given a connected undirected graph $G = (V, E)$ where vertices represent houses (imagine one being on the grid) and edges $(u, v) \in E$ represent possible connections between houses. Each edge has a weight $w(u, v) > 0$ that gives the cost (amount of wire needed) to connect u and v.
- Looking for a subset $T \subseteq E$ of edges that connect all houses minimising the total weight $w(T) = \sum_{(u,v) \in T} w(u, v)$.
- Cycles are unhelpful, so T must be a tree!
  Call it a **spanning tree** as it spans all vertices.
  Looking for a **minimum(-weight) spanning tree (MST)**.

▶Growing a minimum spanning tree

- Let's try to construct a minimum spanning tree iteratively by adding edges (wiring houses) to a selection $A \subseteq E$.
- This works so long as at each step the current set A is a **subset of some minimum spanning tree**.
- If we can add an edge (u, v) to A such that afterwards A is still a subset of some minimum spanning tree, the edge is called a **safe edge**.
  - Remember from correctness of greedy algorithms: the greedy choice is always safe.
  - We'll see how to determine which edges are safe.

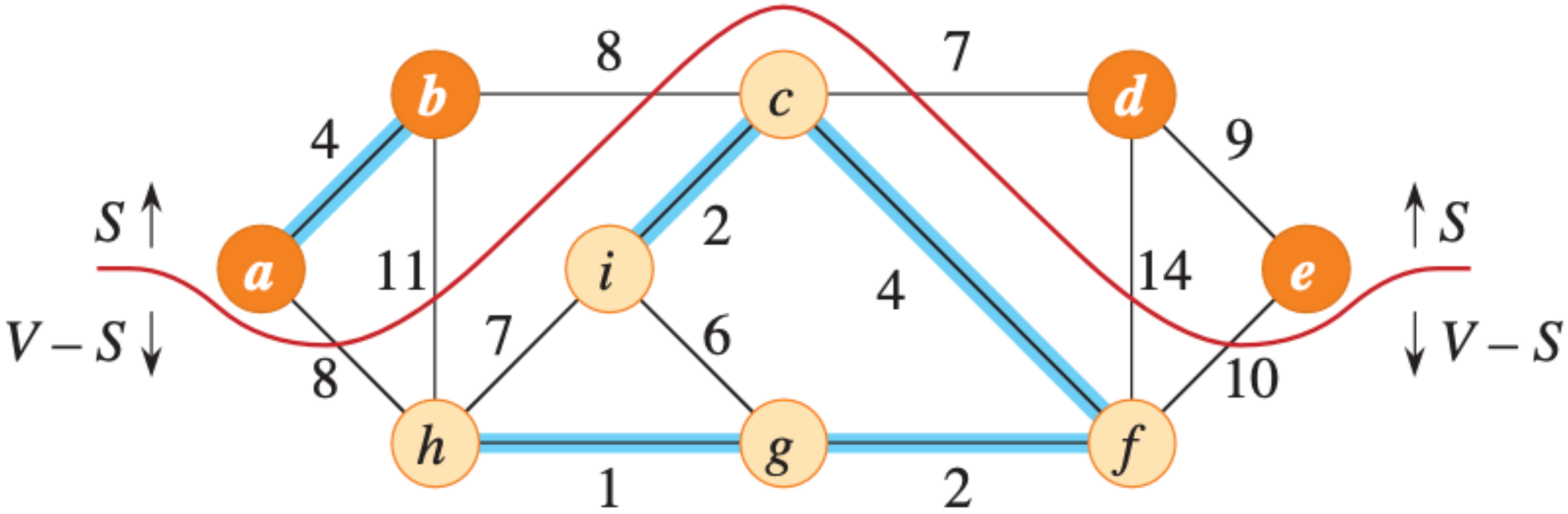
“Abstract”/Generic MST algorithm

```
GENERIC-MST(G, w)
A = ∅
while A does not form a spanning tree
    find an edge (u, v) that is safe for A
    A = A ∪ {(u, v)}
return A
```

- Correctness of this approach by loop invariant:
  - Loop invariant: *Prior to each step, A is a subset of some MST*.
  - Initialisation: $A = \emptyset$ is a subset of some minimum spanning tree.
  - Maintenance: adding a safe edge maintains the loop invariant.
  - Termination: A is a spanning tree and a subset of an MST, so it must be an MST.
- Fair enough. But how to find a safe edge?

Cuts

- A cut of an undirected graph $G = (V, E)$ is a partition of V in two sets $(S, V \setminus S)$.



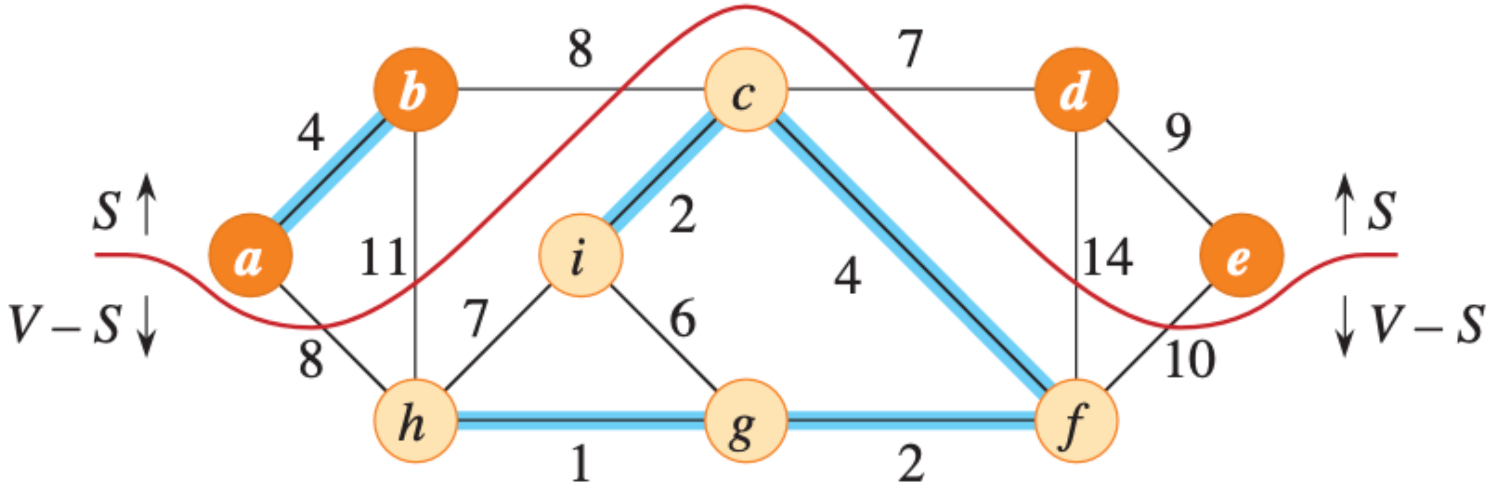
- An edge **crosses** the cut if exactly one of its endpoints is in S.
- A cut **respects** a set A of edges if no edge in A crosses the cut.
- An edge is a **light edge** if its weight is minimal among all edges with some property, e. g. for all edges crossing the cut.

Condition for safe edges

- **Theorem 23.1**: Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function w defined on E. Let A be a subset of E that is included in some MST for G. **If $(S, V \setminus S)$ is a cut of G that respects A, and (u, v) is a light edge crossing $(S, V \setminus S)$ then (u, v) is safe for A.**

- In other words: adding a crossing edge of minimal weight to a partial MST is a **safe choice**.

- Proof is similar to the correctness of greedy algorithms, where we show that a greedy choice is safe.



Proof of Theorem 23.1 (1)

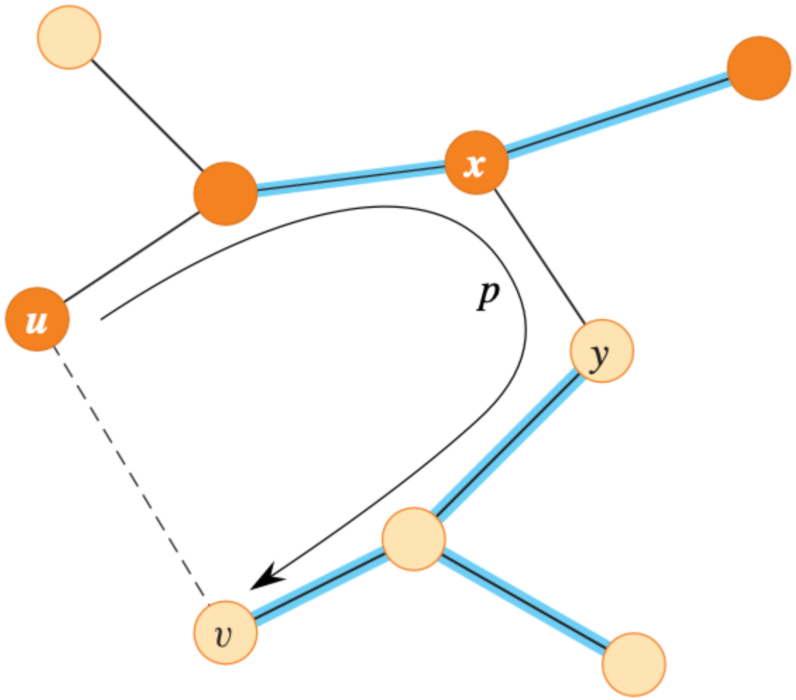
- **Theorem 23.1**: Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function w defined on E. Let A be a subset of E that is included in some MST for G. **If $(S, V \setminus S)$ is a cut of G that respects A, and (u, v) is a light edge crossing $(S, V \setminus S)$ then (u, v) is safe for A.**

Proof:

- Let T be a minimum spanning tree that includes A.
- If T includes (u, v), we are done.
- Now assume that T does not include (u, v). Then we create another minimum spanning tree T' that does include (u, v).
- We do this by cutting an edge and pasting in (u, v).

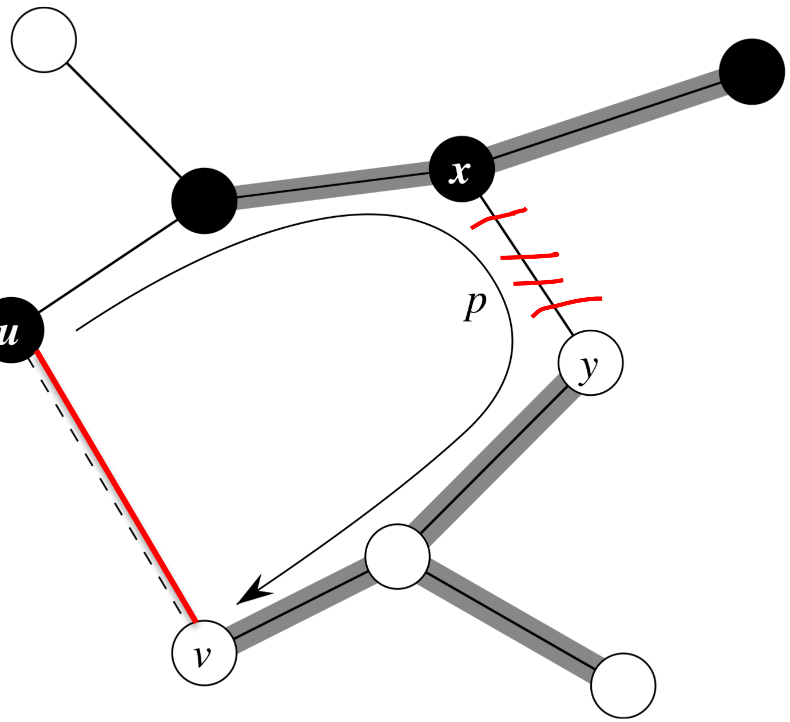
▶Proof of Theorem 23.1 (2)

- Since T is a spanning tree, the edge (u, v) forms a cycle with the simple path p from u to v in T.
- Since u and v are on different sides of the cut (S, V – S), at least one edge (x, y) of p crosses the cut.
- The edge (x, y) is not in A as the cut respects A.
- Since (x, y) is on the unique simple path from u to v in T, removing (x, y) breaks T into two components.
- Adding (u, v) reconnects them to form a new spanning tree $T' = T - \{(x, y)\} \cup \{(u, v)\}$.



Proof of Theorem 23.1 (3)

- We show that T’ is a minimum spanning tree.
- Since (u, v) is a light edge crossing (S, V – S), and (x, y) also crosses the cut,
$$w(u,v) \leq w(x,y).$$
- Hence T’ has weight
$w(T') = w(T) - w(x,y) + w(u,v) \leq w(T).$
- But T is a minimum spanning tree, hence T’ must also be a minimum spanning tree.
- Why is (u, v) safe for A? We have $A \subseteq T'$, $A \subseteq T$ (by assumption on T) and $(x,y) \notin A$, thus
$$A \cup \{(u,v)\} \subseteq T'.$$
- Adding (u, v) to A is a safe choice as we can still construct a minimum spanning tree T’.



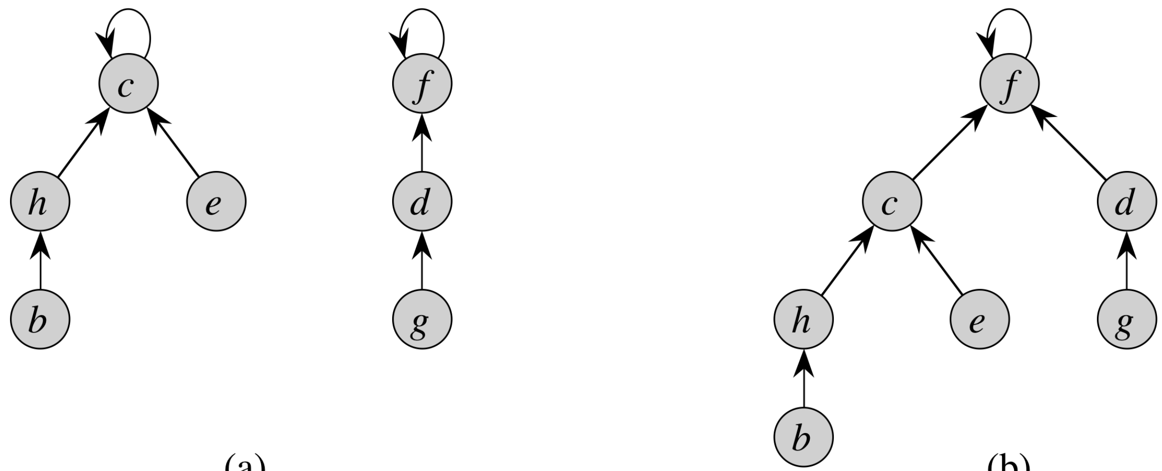
Edges connecting components are safe

```
GENERIC-MST(G, w)
A = Ø
while A does not form a spanning tree
    find an edge (u, v) that is safe for A
    A = A ∪ {(u, v)}
return A
```

- The "abstract" MST algorithm (adding safe edges to A) constructs **a forest**.
- Note that initially all vertices are isolated and form their own trees.
- Theorem 23.1 implies that **for any tree T in that forest**, a light edge from T to the union of other trees is a safe edge.
  – Why? The cut $(T, V \setminus T)$ respects the forest A, so the theorem applies.

Kruskal's algorithm

- Idea: connect two trees adding an edge with minimum weight.
- Need a way of finding out which tree a vertex belongs to.
- **Union-Find data structures** store names of sets:
  - **Find-Set(u)** returns the name of a set that element u belongs to.
  - **Union(u, v)** merges the two sets u and v belong to (if different)
  - Can be implemented efficiently with trees where the root contains the name of the set (details in Chapter 19).



(a) (b)

Kruskal's algorithm (2)

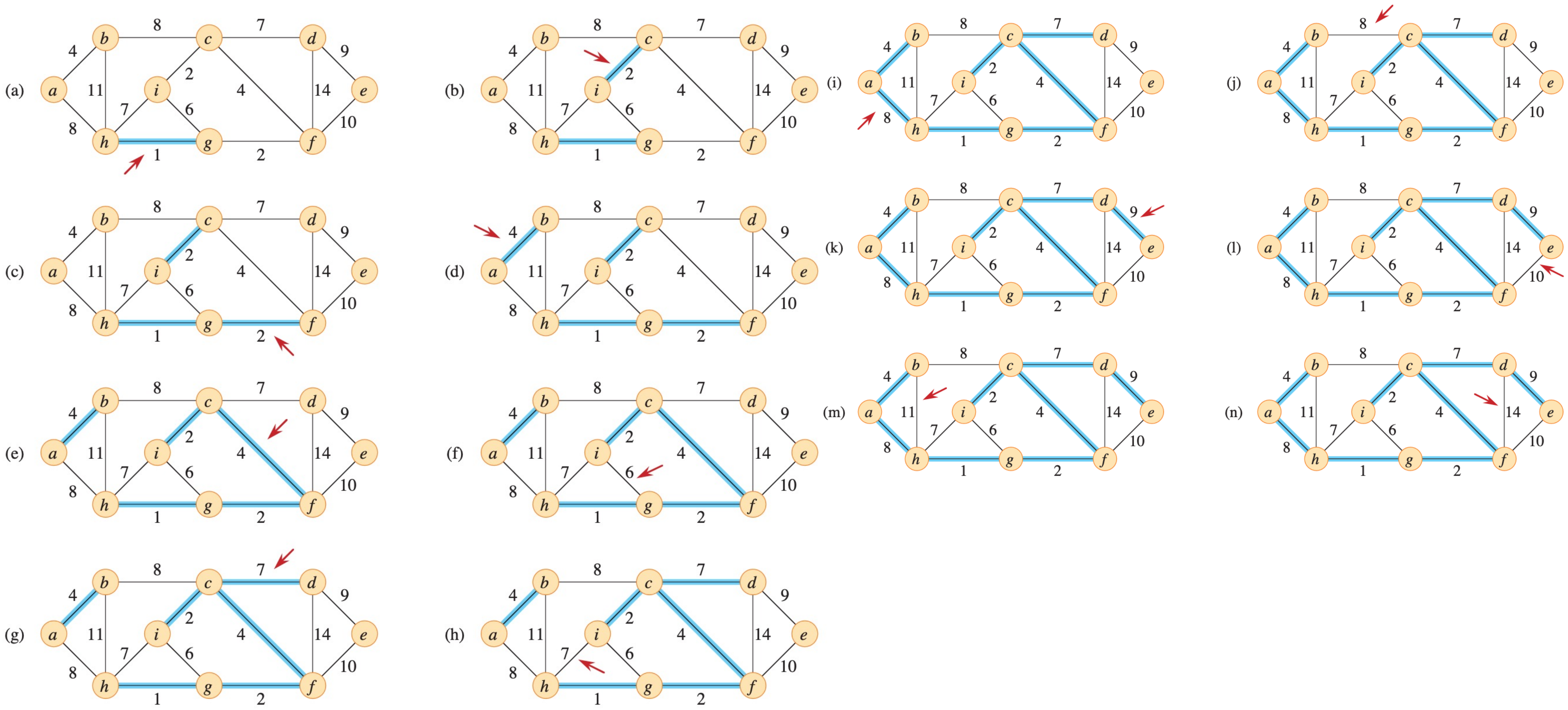
Ideas:

- sort all edges according to weight and process edges in this order.
- If both ends belong to different trees, add the edge and join the trees.

```
KRUSKAL(G, w)
A = ∅
for each vertex v ∈ V do
    make a set {v}
sort the edges of E in nondecreasing order by weight w
for each edge (u, v) in this order do
    if FIND-SET(u) ≠ FIND-SET(v) then
        A = A ∪ {(u, v)}
        UNION(u, v)
return A
```

- **Runtime?**
- With efficient data structures for unions and finds, the runtime is dominated by the time for sorting: $O\big(|E|\log(|E|)\big)$.
- Since $\log(|E|) \leq \log(|V|^2) = 2\log(|V|) = O(\log(|V|))$ we may write the runtime as $O\big(|E|\log(|V|)\big)$.

Kruskal's Algorithm: Example



Prim's algorithm

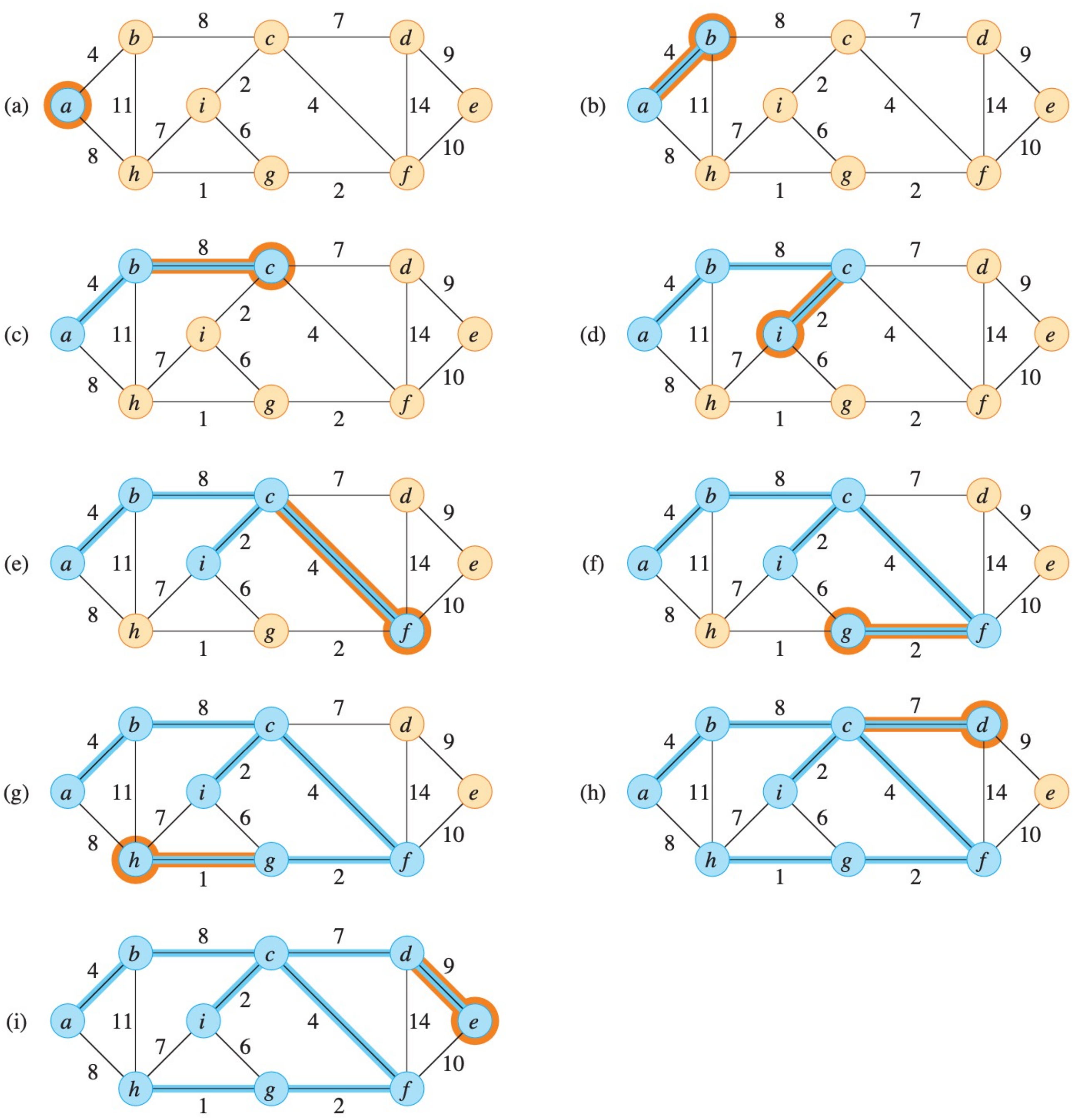
- Alternative implementation of the "abstract" MST algorithm
- Idea: **grow a single tree** A by adding a minimum-weight edge leading away from the tree (a light edge to an isolated vertex).
- Since isolated vertices are trees, such a light edge is safe.
- How to implement Prim's algorithm efficiently?
  - Need to find a light (minimum-weight) edge to add to the tree.
  - We maintain a distance of each node to the tree (similar to BFS)
  - Initially all distances are ∞.
  - Distances may decrease when new vertices are added to the tree.
  - Use a **Priority Queue** to keep track of the nodes with shortest distance to the current tree (light edges)

Implementing Prim's algorithm

- Need to find a light (minimum-weight) edge to add to the tree.
- We maintain a distance "key" of each node to the tree (similar to BFS)
- Initially all distances are ∞.
- Distances may decrease when new vertices are added to the tree.
- MST given by predecessors π (as for BFS)

```
PRIM(G, w, r)
for each vertex u ∈ V do
    u.key = ∞
    u.π = NIL
r.key = 0
Q = V
while Q ≠ ∅ do
    u = EXTRACT-MIN(Q)
    for each v ∈ Adj[u] do
        if v ∈ Q and w(u, v) < v.key then
            v.π = u
            v.key = w(u, v)
```

Prim: Example



Priority Queue based on min-heap

- A data structure for maintaining a set S of elements with an associated element called key.
- **Min-priority queue** based on min-heap defined as follows:

Operation	Time
Insert(S, x) – insert x into S	$O(\log n)$
Minimum(S) – returns smallest element in S	$O(1)$
Extract-Min(S) – removes and returns smallest element in S	$O(\log n)$
Decrease-Key(S, x, k) – decreases x's value to smaller value k (element may float up in the heap)	$O(\log n)$

▶Runtime of Prim's algorithm w/ Min-Heaps

- Runtime exclusive of red lines (as for BFS):
  $$O(|V| + |E|)$$
  (store a bit in each vertex to make the test $v \in Q$ run in O(1) time)
- Building a Min-Heap: O(|V|).
- Runtime for all calls to Extract-Min is $O(|V| \log(|V|))$.
- Runtime for at most $|E|$ Decrease-Keys is $O(|E| \log(|V|))$.
- **Total**: $\boldsymbol{O(|E| \log(|V|))}$ as (since G is connected) $|V| = O(|E|)$.

```
PRIM(G, w, r)
for each vertex u ∈ V do
    u.key = ∞
    u.π = NIL
r.key = 0
Q = V
BUILD-MIN-HEAP(Q)
while Q ≠ ∅ do
    u = EXTRACT-MIN(Q)
    for each v ∈ Adj[u] do
        if v ∈ Q and w(u, v) < v.key then
            v.π = u
            DECREASE-KEY(Q, v.key, w(u, v))
```

Shortest Path Problems

- Given a directed graph with edge weights representing distances, what is the shortest path between two vertices?
- To find the shortest path from ShenZhen 深圳 to ShangHai 上海, exploring all paths (e.g. via BeiJing 北京) is not helpful. Need a smarter approach.
- Breadth-first search finds shortest paths when all distances are 1, but can't deal with weights.
- Assume that all distances are non-negative.
- Note that shortest paths exhibit **optimal substructure**: a shortest path from s to u going through v is composed of a shortest path from s to v and a shortest path from v to u.

Variants of Shortest Path Problems

- **Single-source shortest paths problem (SSSP)**: find shortest paths from a source vertex to all other vertices.
- **Single-destination shortest paths problem (SDSP)**: find shortest paths from all vertices to a destination vertex.
  - Like single-source shortest paths, simply invert all edges.
- **Single-pair shortest-paths problem (SPSP)**: find a shortest path between two vertices.
  - Actually not much easier than single-source shortest paths!
- **All-pairs shortest paths problem (APSP)**: find shortest paths between all pairs of vertices.
  - Trivial: solve single-source shortest paths for all vertices. More clever solutions are more efficient.

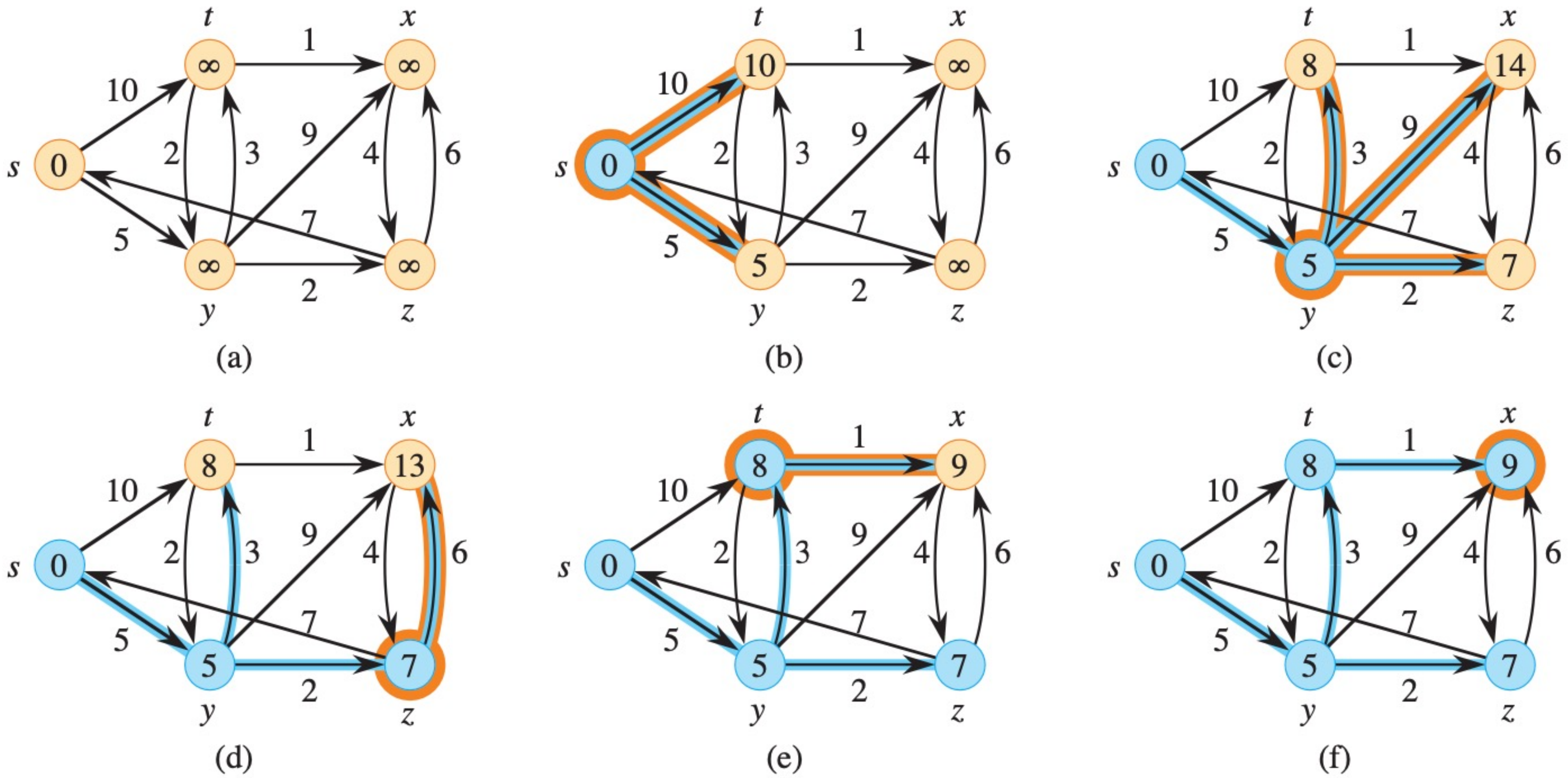
Dijkstra's algorithm for the SSSP

- **Idea from BFS:** Maintain **distance estimates .d** that are no smaller than shortest-path distances.
- Grow a set S of vertices whose **final shortest-path distances** from source s have been found.
- **Idea from Prim:** In each step, **add the closest vertex** from $V \setminus S$ (smallest distance estimate .d → greedy choice).
- Refine distance estimates after each expansion of S.

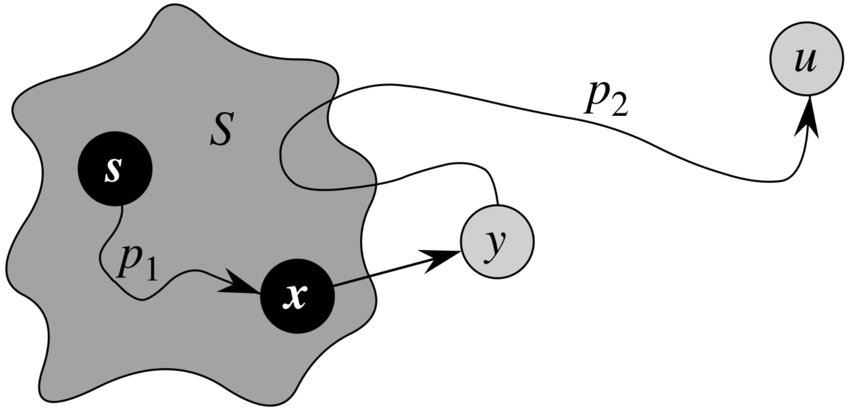
DIJKSTRA(G, w, s)

Initialise d and π in the usual way.
$S = \emptyset$
$Q = V$
while $Q \neq \emptyset$ **do**
  $u =$ EXTRACT-MIN(Q)
  $S = S \cup \{u\}$
  for each $v \in \mathrm{Adj}[u]$ **do**
    if $v.d > u.d + w(u, v)$ **then**
      $v.d = u.d + w(u, v)$
      $v.\pi = u$
      DECREASE-KEY$(Q, v, v.d)$

▶Dijkstra's algorithm: Example



Correctness of Dijkstra's algorithm



- We show that at the time a vertex u is added to S, $u.d$ is the shortest-path distance.
- This holds for s, so assume for a **contradiction** that $u \neq s$ is the **first vertex added to S** for which $u.d$ is larger than the shortest-path distance ($u.d \neq \delta(s,u)$).
- Consider a **shortest path p from s to u** and let y be the first vertex **outside of S** on this path. Let $x \in S$ be its predecessor.
- By choice of u, $x.d$ is the shortest-path distance to x, and when x was added, $y.d$ was set to $x.d + w(x,y) = \delta(s,y)$, the shortest-path distance to y (because otherwise p would not be the shortest to u).
- Since the path p_2 from y to u has non-negative distance,
$$y.d \leq \delta(s,y) \leq \delta(s,u) \leq u.d.$$
- Since u is added to S before y, $u.d \leq y.d$. Together $u.d = y.d$ and since y has the correct shortest-path distance, so has u, contradiction.

Runtime of Dijkstra w/ Min-Heaps

- Runtime exclusive of red lines : $O(|V| + |E|)$
- Building a Min-Heap: O(|V|).
- Runtime for all calls to Extract-Min is $O(|V|\log(|V|))$.
- Runtime for at most $|E|$ Decrease-Keys is $O(|E|\log(|V|))$.
- **Total**: $\boldsymbol{O((|V| + |E|) \log(|V|))}$ or $\boldsymbol{O(|E| \log(|V|))}$ if all vertices area reachable from the source.
- NB: for single-pair shortest paths we may stop when destination found.

$\textsc{Dijkstra}(G, w, s)$

```
Initialise d and π in the usual way.
S = ∅
Q = V
BUILD-MIN-HEAP(Q)
while Q ≠ ∅ do
    u = EXTRACT-MIN(Q)
    S = S ∪ {u}
    for each v ∈ Adj[u] do
        if v.d > u.d + w(u, v) then
            DECREASE-KEY(Q, v.d, u.d + w(u, v))
            v.π = u
```

Summary

- Minimum spanning trees can be solved with two greedy algorithms:
  - Kruskal's algorithm adds the lightest edge connecting two trees
  - Prim's algorithm grows one tree by adding the lightest edge
- Dijkstra's algorithm solves single-source shortest paths by expanding on the set of vertices closest to the source.
  - Combines greedy and dynamic programming approaches
- Efficient data structures (union-find and priority queues) are vital for implementing the above algorithms efficiently.
- All algorithms can be implemented in time $O(|E|\log(|V|))$.
  - Advanced data structures (Fibonacci heaps) can improve this further.