Greedy Algorithms

Greedy Algorithms

- Optimization problem typically go through a sequence of steps, with a set of choices at each step (Example of Travelling Salesman Problem).

Greedy Algorithms

- Optimization problem typically go through a sequence of steps, with a set of choices at each step (Example of Travelling Salesman Problem).

- One possible way to determine the best choice is a brute-force.

Greedy Algorithms

- Optimization problem typically go through a sequence of steps, with a set of choices at each step (Example of Travelling Salesman Problem).
- One possible way to determine the best choice is a brute-force.
- Greedy algorithms are often used to solve optimization problems.

Greedy Algorithms

- Optimization problem typically go through a sequence of steps, with a set of choices at each step (Example of Travelling Salesman Problem).
- One possible way to determine the best choice is a brute-force.
- Greedy algorithms are often used to solve optimization problems.
- Greedy algorithm always makes the locally optimal choice in the hope that this choice will lead to a globally optimal solution.

Greedy Algorithms

- In other words, as their name suggest, these algorithms are shortsighted in their approach, taking decisions on the basis of information immediately at hand without worrying about the effect these decisions may have in the future.

Greedy Algorithms

- In other words, as their name suggest, these algorithms are shortsighted in their approach, taking decisions on the basis of information immediately at hand without worrying about the effect these decisions may have in the future.

- Greedy algorithms do not always obtain optimal solutions, but for many problems they do.

Greedy Algorithms

- In other words, as their name suggest, these algorithms are shortsighted in their approach, taking decisions on the basis of information immediately at hand without worrying about the effect these decisions may have in the future.

- Greedy algorithms do not always obtain optimal solutions, but for many problems they do.

- They are easy to invent, easy to implement, and-when they work-efficient.

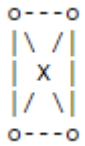
Greedy Algorithms

- Minimum Spanning Trees
  - A spanning tree of a graph is just a subgraph and more precisely a tree which consists |V| (i.e. all) vertices and |V| - 1 edges connecting all vertices of a graph.

Greedy Algorithms

- Minimum Spanning Trees
  - A spanning tree of a graph is just a subgraph and more precisely a tree which consists |V| (i.e. all) vertices and |V| - 1 edges connecting all vertices of a graph.

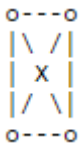
  - A graph may have many spanning trees; for instance the complete graph on four vertices

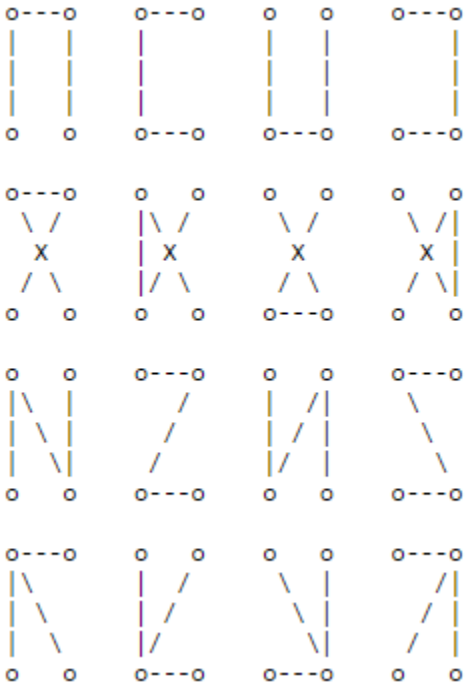


has sixteen spanning trees:

Greedy Algorithms

- Minimum Spanning Trees





Greedy Algorithms

- Minimum Spanning Trees
  - Now suppose the edges of the graph have weights or lengths.

Greedy Algorithms

- Minimum Spanning Trees
  - Now suppose the edges of the graph have weights or lengths.
  - The weight of a tree is just the sum of weights of its edges.

Greedy Algorithms

- Minimum Spanning Trees
  - Now suppose the edges of the graph have weights or lengths.
  - The weight of a tree is just the sum of weights of its edges.
  - Obviously, different trees have different lengths/weights.

Greedy Algorithms

- Minimum Spanning Trees
  - Now suppose the edges of the graph have weights or lengths.
  - The weight of a tree is just the sum of weights of its edges.
  - Obviously, different trees have different lengths/weights.
  - The spanning tree with the minimum length or weight is a minimum spanning tree.

Greedy Algorithms

- Minimum Spanning Trees - Applications
  - Electronic circuit designs often need to make the pins of several components electrically equivalent by wiring them together.

Greedy Algorithms

- Minimum Spanning Trees - Applications
  - Electronic circuit designs often need to make the pins of several components electrically equivalent by wiring them together.
  - To interconnect a set of n pins, we can use an arrangement of n - 1 wires, each connecting two pins.

Greedy Algorithms

- Minimum Spanning Trees - Applications
  - Electronic circuit designs often need to make the pins of several components electrically equivalent by wiring them together.
  - To interconnect a set of n pins, we can use an arrangement of n - 1 wires, each connecting two pins.
  - Of all such arrangements, the one that uses the least amount of wire is usually the most desirable.

Greedy Algorithms

- Minimum Spanning Trees - Applications
  - Electronic circuit designs often need to make the pins of several components electrically equivalent by wiring them together.
  - To interconnect a set of n pins, we can use an arrangement of n - 1 wires, each connecting two pins.
  - Of all such arrangements, the one that uses the least amount of wire is usually the most desirable.
  - We can model this wiring problem with a connected, undirected graph $G = (V, E)$, where V is the set of pins, E is the set of possible interconnections between pairs of pins, and for each edge $(u, v) \in E$, we have a weight $w(u, v)$ specifying the cost (amount of wire needed) to connect u and v.

Greedy Algorithms

- Minimum Spanning Trees
  - We then wish to find an acyclic subset T ⊆ E that connects all of the vertices and whose total weight

$$w(T) = \sum_{(u,v)\in T} w(u,v)$$

  is minimized.

Greedy Algorithms

- Minimum Spanning Trees
  - We then wish to find an acyclic subset T ⊆ E that connects all of the vertices and whose total weight

$$w(T) = \sum_{(u,v)\in T} w(u,v)$$

is minimized.

- Since T is acyclic and connects all of the vertices, it must form a tree, which we call a spanning tree since it "spans" the graph G.

Greedy Algorithms

- Minimum Spanning Trees
  - We then wish to find an acyclic subset T ⊆ E that connects all of the vertices and whose total weight

$$w(T) = \sum_{(u,v)\in T} w(u,v)$$

is minimized.

- Since T is acyclic and connects all of the vertices, it must form a tree, which we call a spanning tree since it "spans" the graph G.

- We call the problem of determining the tree T the minimum-spanning-tree problem.

Greedy Algorithms

- Minimum Spanning Trees
  - We then wish to find an acyclic subset T ⊆ E that connects all of the vertices and whose total weight

$$w(T) = \sum_{(u,v)\in T} w(u,v)$$

is minimized.

- Since T is acyclic and connects all of the vertices, it must form a tree, which we call a spanning tree since it "spans" the graph G.

- We call the problem of determining the tree T the minimum-spanning-tree problem.

- Figure 23.1 shows an example of a connected graph and a minimum spanning tree.

Greedy Algorithms

- Minimum Spanning Trees

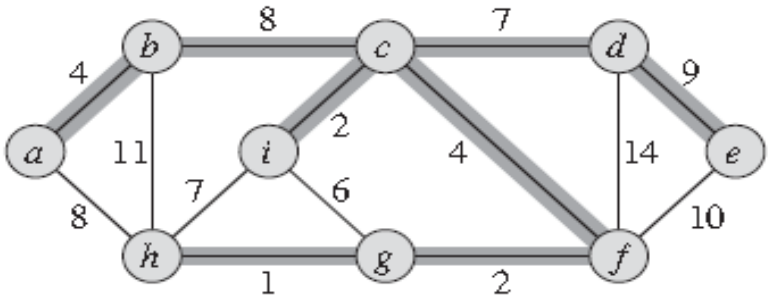


Figure 23.1 A minimum spanning tree for a connected graph. The weights on edges are shown, and the edges in a minimum spanning tree are shaded. The total weight of the tree shown is 37. This minimum spanning tree is not unique: removing the edge (b, c) and replacing it with the edge (a, h) yields another spanning tree with weight 37.

- We will examine two algorithms for solving the minimum-spanning-tree problem: Kruskal's algorithm and Prim's algorithm.
- We can easily make each of them run in time O(E lgV) using ordinary binary heaps.
- By using Fibonacci heaps, Prim's algorithm runs in time O(E + V lgV), which improves over the binary-heap implementation if |V| is much smaller than |E|.

Greedy Algorithms

- Minimum Spanning Trees
  - MAKE-SET(x)
    - New Set whose only member is x
    - Disjoint Property
  - UNION(x, y)
    - Unites S_x and S_y
    - Representative
    - Disjoint Property
  - FIND-SET(x)
    - Returns a pointer to the representative of the unique set containing x.

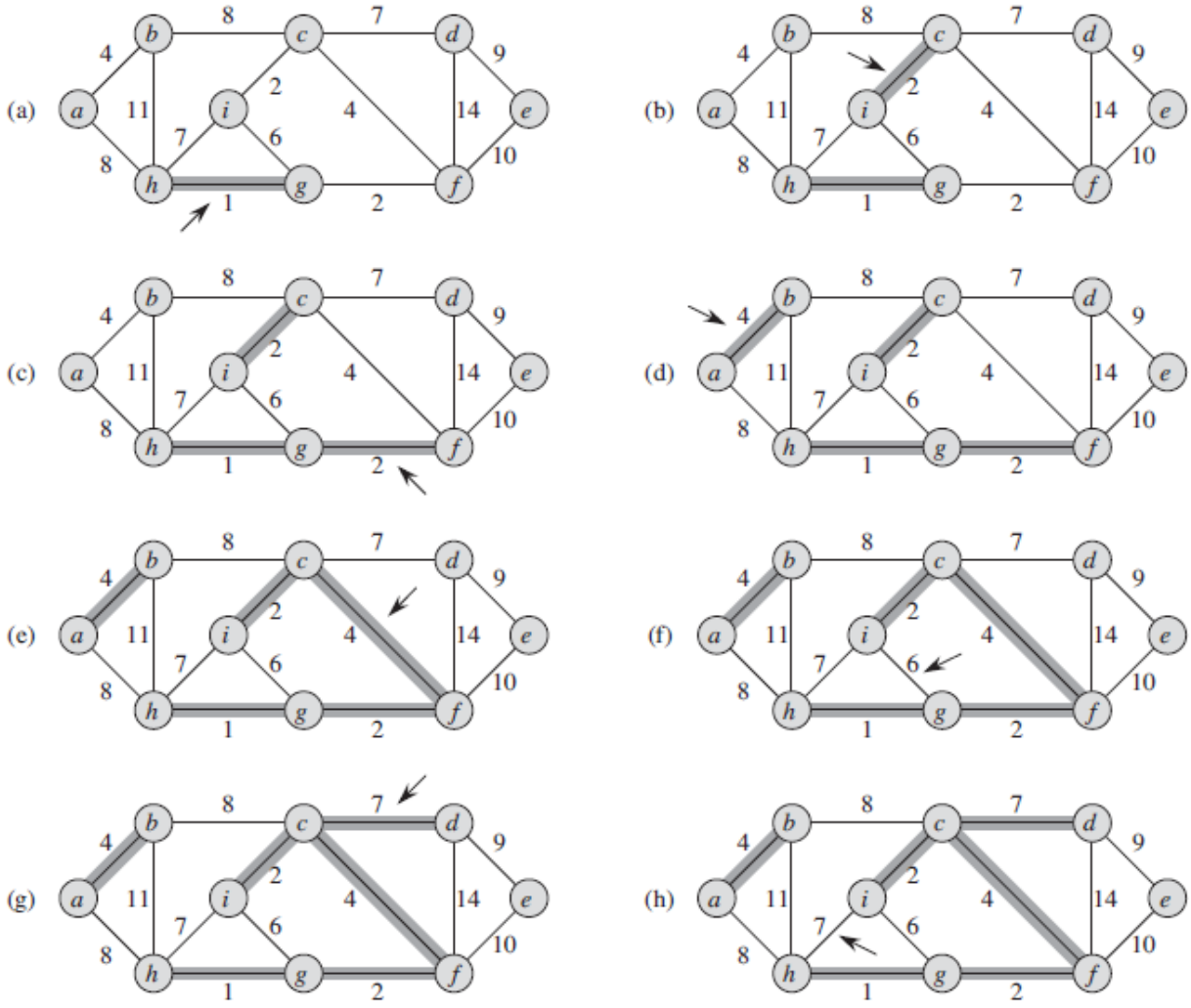
Greedy Algorithms

- Minimum Spanning Trees
  - We have been given connected, undirected and weighted graph G = (V, E). Weight function is w: E -> R
  - Kruskal's Algorithm

```
MST-KRUSKAL(G, w)
1  A = ∅
2  for each vertex v ∈ G.V
3       MAKE-SET(v)
4  sort the edges of G.E into nondecreasing order by weight w
5  for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
6       if FIND-SET(u) ≠ FIND-SET(v)
7            A = A ∪ {(u, v)}
8            UNION(u, v)
9  return A
```

Greedy Algorithms

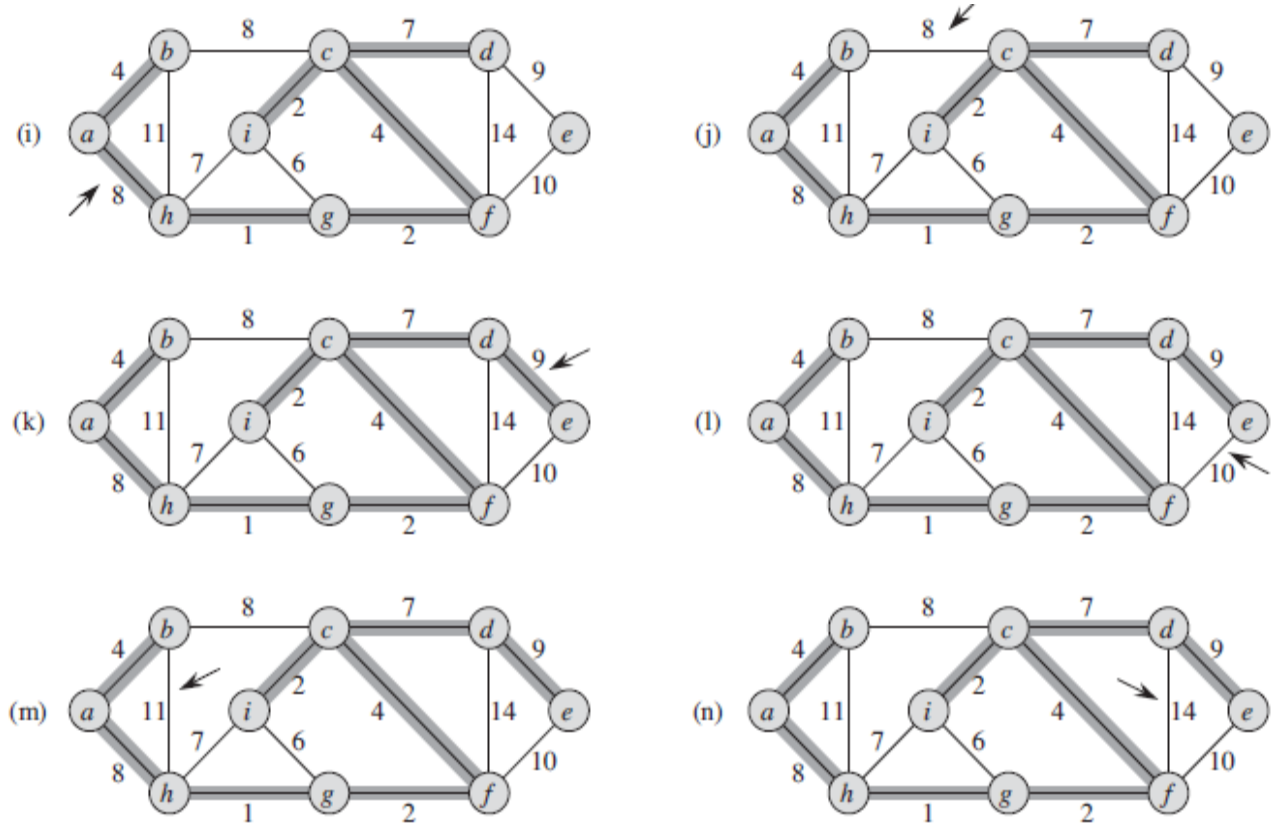
- Minimum Spanning Trees
  - Kruskal's Algorithm



1. {a}, {b}, {c}, {d}, {e}, {f}, {g}, {h}, {i}
a. {h,g}, {a}, {b}, {c}, {d}, {e}, {f}, {i}
b. {h,g}, {a}, {b}, {i,c}, {d}, {e}, {f}
c. {h,g,f}, {a}, {b}, {i,c}, {d}, {e}
d. {h,g,f}, {a, b}, {i,c}, {d}, {e}
e. {h,g,f,i,c}, {a,b}, {d}, {e}
f. No change
g. {h,g,f,i,c,d}, {a,b}, {e}
h. No change

Greedy Algorithms

- Minimum Spanning Trees
  - Kruskal's Algorithm



1. {a}, {b}, {c}, {d}, {e}, {f}, {g}, {h}, {i}
a. {h,g}, {a}, {b}, {c}, {d}, {e}, {f}, {i}
b. {h,g}, {a}, {b}, {i,c}, {d}, {e}, {f}
c. {h,g,f}, {a}, {b}, {i,c}, {d}, {e}
d. {h,g,f}, {a, b}, {i,c}, {d}, {e}
e. {h,g,f,i,c}, {a,b}, {d}, {e}
f. No change
g. {h,g,f,i,c,d}, {a,b}, {e}
h. No change
i. {h,g,f,i,c,d,a,b}, {e}
j. No change
k. {h,g,f,i,c,d,a,b,e}
l. No change
m. No change
n. No change

Greedy Algorithms

- Minimum Spanning Trees
  - Kruskal's Algorithm

$O(1) + O(E \log E) + O(V+E) = O(E \log E)$
(As $|E| >= |V| - 1$ and therefor E log E is certainly asymptotically larger once $|V| >= 6$)

```
MST-KRUSKAL(G, w)
1  A = ∅
2  for each vertex v ∈ G.V
3       MAKE-SET(v)
4  sort the edges of G.E into nondecreasing order by weight w
5  for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
6       if FIND-SET(u) ≠ FIND-SET(v)
7            A = A ∪ {(u, v)}
8            UNION(u, v)
9  return A
```

Greedy Algorithms

- Minimum Spanning Trees
  - Prim's Algorithm
    - Prim's algorithm has the property that the edges in the set A always form a single tree.

Greedy Algorithms

- Minimum Spanning Trees
  - Prim's Algorithm
    - Prim's algorithm has the property that the edges in the set A always form a single tree.
    - As Figure shows, the tree starts from an arbitrary root vertex r and grows until the tree spans all the vertices in V .

Greedy Algorithms

- Minimum Spanning Trees
  - Prim's Algorithm
    - Prim's algorithm has the property that the edges in the set A always form a single tree.
    - As Figure shows, the tree starts from an arbitrary root vertex r and grows until the tree spans all the vertices in V .
    - Each step adds to the tree A, a light edge that connects A to an isolated vertex—one on which no edge of A is incident.

Greedy Algorithms

- Minimum Spanning Trees
  - Prim's Algorithm
    - Prim's algorithm has the property that the edges in the set A always form a single tree.
    - As Figure shows, the tree starts from an arbitrary root vertex r and grows until the tree spans all the vertices in V .
    - Each step adds to the tree A, a light edge that connects A to an isolated vertex—one on which no edge of A is incident.
    - This strategy qualifies as greedy since at each step it adds to the tree an edge that contributes the minimum amount possible to the tree's weight.

Greedy Algorithms

- Minimum Spanning Trees
  - Prim's Algorithm
    - In order to implement Prim's algorithm efficiently, we need a fast way to select a new edge to add to the tree formed by the edges in A.

Greedy Algorithms

- Minimum Spanning Trees
  - Prim's Algorithm
    - In order to implement Prim's algorithm efficiently, we need a fast way to select a new edge to add to the tree formed by the edges in A.
    - In the pseudocode below, the connected graph G and the root r of the minimum spanning tree to be grown are inputs to the algorithm.

Greedy Algorithms

- Minimum Spanning Trees
  - Prim's Algorithm
    - In order to implement Prim's algorithm efficiently, we need a fast way to select a new edge to add to the tree formed by the edges in A.
    - In the pseudocode below, the connected graph G and the root r of the minimum spanning tree to be grown are inputs to the algorithm.
    - During execution of the algorithm, all vertices that are not in the tree reside in a min-priority queue Q based on a key attribute.

Greedy Algorithms

- Minimum Spanning Trees
  - Prim's Algorithm
    - In order to implement Prim's algorithm efficiently, we need a fast way to select a new edge to add to the tree formed by the edges in A.
    - In the pseudocode below, the connected graph G and the root r of the minimum spanning tree to be grown are inputs to the algorithm.
    - During execution of the algorithm, all vertices that are not in the tree reside in a min-priority queue Q based on a key attribute.
    - For each vertex v, the attribute v.key is the minimum weight of any edge connecting v to a vertex in the tree; by convention, v.key = ∞, if there is no such edge.

Greedy Algorithms

- Minimum Spanning Trees
  - Prim's Algorithm
    - In order to implement Prim's algorithm efficiently, we need a fast way to select a new edge to add to the tree formed by the edges in A.
    - In the pseudocode below, the connected graph G and the root r of the minimum spanning tree to be grown are inputs to the algorithm.
    - During execution of the algorithm, all vertices that are not in the tree reside in a min-priority queue Q based on a key attribute.
    - For each vertex v, the attribute v.key is the minimum weight of any edge connecting v to a vertex in the tree; by convention, v.key = ∞, if there is no such edge.
    - The attribute v.π names the parent of v in the tree.

Greedy Algorithms

- Minimum Spanning Trees
  - Prim's Algorithm
    - The algorithm implicitly maintains the set A as

$$A = \{(v, v.\pi) : v \in V - \{r\} - Q\}$$

- When the algorithm terminates, the min-priority queue Q is empty; the minimum spanning tree A for G is thus $A = \{(v, v.\pi) : v \in V - \{r\}\}$

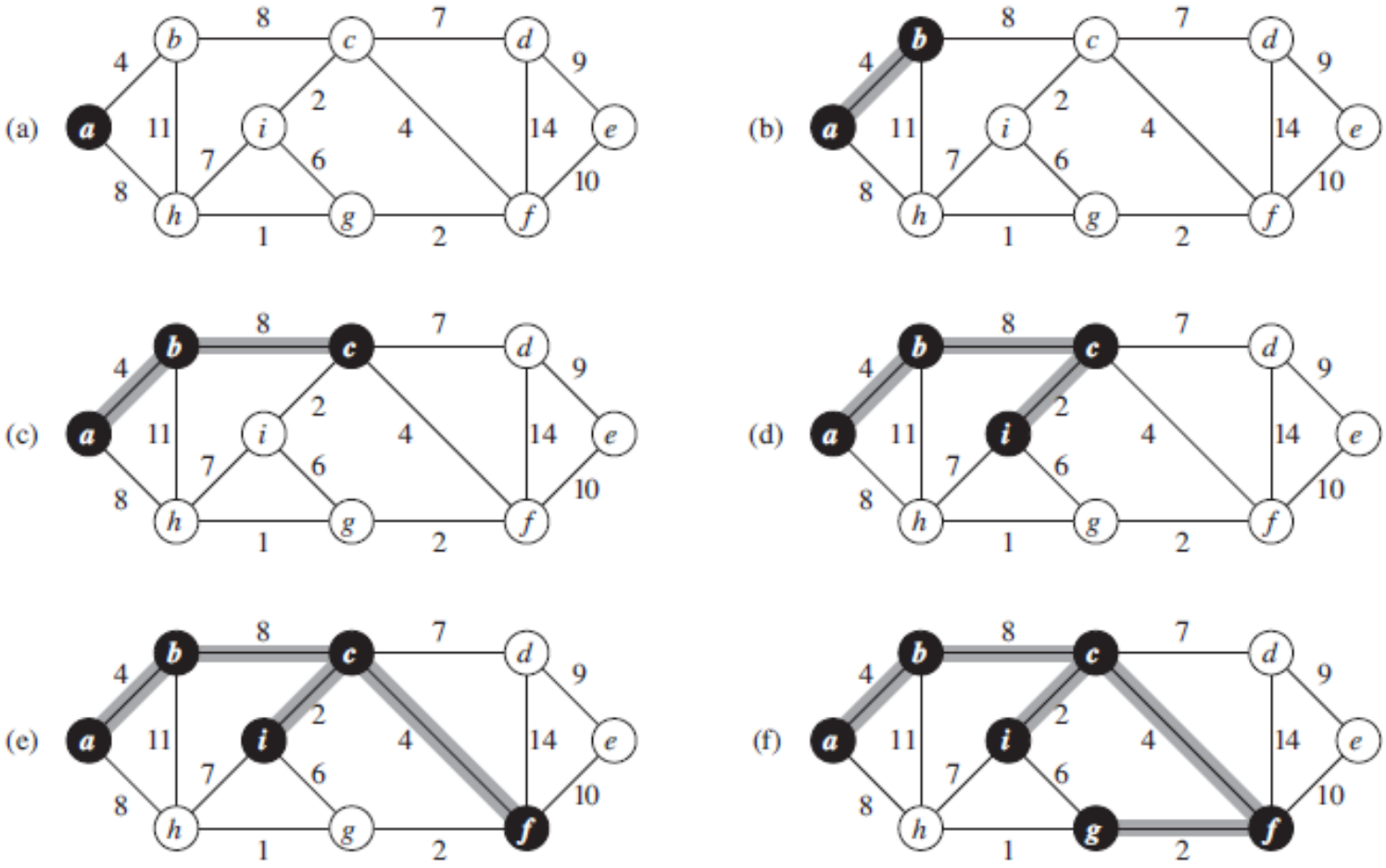
```
MST-PRIM(G, w, r)
 1  for each u ∈ G.V
 2      u.key = ∞
 3      u.π = NIL
 4  r.key = 0
 5  Q = G.V
 6  while Q ≠ ∅
 7      u = EXTRACT-MIN(Q)
 8      for each v ∈ G.Adj[u]
 9          if v ∈ Q and w(u, v) < v.key
10              v.π = u
11              v.key = w(u, v)
```

Line 7 grows MST with u and its parent

Line 8-11 updates adjacent vertices (which are not in MST and whose new connection to MST is lighter than the one already existing) of u which is already a part of MST.

Greedy Algorithms

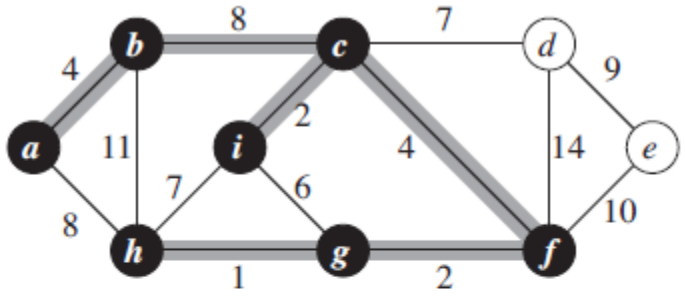
- {a, b, c, d, e, f, g, h, i} - {0, INF, INF, INF, INF, INF, INF, INF, INF}
- {0, 4, INF, INF, INF, INF, INF, 8, INF}, parent of b, h updated, a in tree
- {0, 4, 8, INF, INF, INF, INF, 8, INF}, parent of c updated, b in tree with (b, a)
- {0, 4, 8, 7, INF, 4, INF, 8, 2}, parent of d, f, i updated, c in tree with (c, b) (Alternatively h could have been in the tree)
- {0, 4, 8, 7, INF, 4, 6, 7, 2}, parent of g and h updated, i in tree with (i, c)
- {0, 4, 8, 7, 10, 4, 2, 7, 2}, parent of g and e updated, f in tree with (f, c)
- {0, 4, 8, 7, 10, 4, 2, 1, 2}, parent of h updated, g in tree with (g, f).



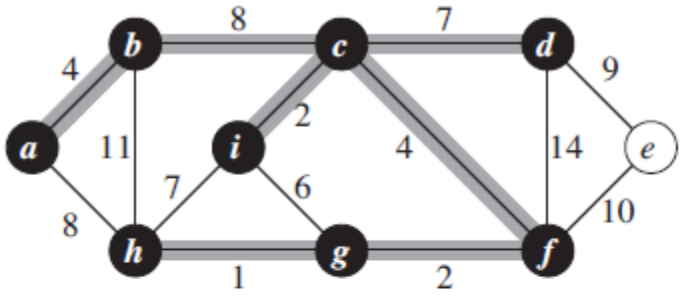
Greedy Algorithms

- Minimum Spanning Trees
  - Prim's Algorithm
    - {0, 4, 8, 7, 10, 4, 2, 1, 2}, h in tree with (h, g)
    - {0, 4, 8, 7, 9, 4, 2, 1, 2}, parent of e updated, d in tree with (d, c),
    - {0, 4, 8, 7, 9, 4, 2, 1, 2}, e in tree with (e, d)

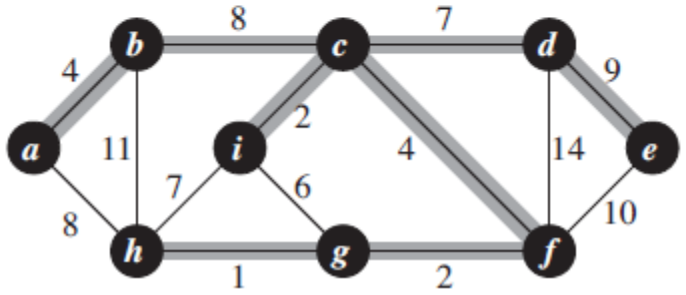
(g)



(h)



(i)



Greedy Algorithms

- Minimum Spanning Trees
  - Prim's Algorithm

```
MST-PRIM(G, w, r)
 1  for each u ∈ G.V
 2      u.key = ∞
 3      u.π = NIL
 4  r.key = 0
 5  Q = G.V
 6  while Q ≠ ∅
 7      u = EXTRACT-MIN(Q)
 8      for each v ∈ G.Adj[u]
 9          if v ∈ Q and w(u, v) < v.key
10              v.π = u
11              v.key = w(u, v)
```

1-3 in O(V)
4 in O(1)
5 is build heap, so in O(V)
6-7 in O(V logV)
8-11 in O(E log V) times as the for loop in totality iterates O(E) times and it involves implicit decrease key operation.

Greedy Algorithms

- Minimum Spanning Trees
  - Prim's Algorithm

The running time of Prim's algorithm depends on how we implement the min-priority queue Q. If we implement Q as a binary min-heap (see Chapter 6), we can use the BUILD-MIN-HEAP procedure to perform lines 1–5 in $O(V)$ time. The body of the **while** loop executes $|V|$ times, and since each EXTRACT-MIN operation takes $O(\lg V)$ time, the total time for all calls to EXTRACT-MIN is $O(V \lg V)$. The **for** loop in lines 8–11 executes $O(E)$ times altogether, since the sum of the lengths of all adjacency lists is $2|E|$. Within the **for** loop, we can implement the test for membership in Q in line 9 in constant time by keeping a bit for each vertex that tells whether or not it is in Q, and updating the bit when the vertex is removed from Q. The assignment in line 11 involves an implicit DECREASE-KEY operation on the min-heap, which a binary min-heap supports in $O(\lg V)$ time. Thus, the total time for Prim's algorithm is $O(V \lg V + E \lg V) = O(E \lg V)$, which is asymptotically the same as for our implementation of Kruskal's algorithm.

We can improve the asymptotic running time of Prim's algorithm by using Fibonacci heaps. Chapter 19 shows that if a Fibonacci heap holds $|V|$ elements, an EXTRACT-MIN operation takes $O(\lg V)$ amortized time and a DECREASE-KEY operation (to implement line 11) takes $O(1)$ amortized time. Therefore, if we use a Fibonacci heap to implement the min-priority queue Q, the running time of Prim's algorithm improves to $O(E + V \lg V)$.

Greedy Algorithms

- Single Source Shortest Paths
  - In single-source shortest-paths problem, given a graph G(V, E), we want to find a shortest path from a given source vertex s ϵ V to each vertex v ϵ V .

Greedy Algorithms

- Single Source Shortest Paths
  - In single-source shortest-paths problem, given a graph G(V, E), we want to find a shortest path from a given source vertex s ϵ V to each vertex v ϵ V .
  - The algorithm use the technique of relaxation. For each vertex v ϵ V , we maintain an attribute v.d, which is an upper bound on the weight/length of a shortest path from source s to v. We call v.d a shortest-path estimate.

Greedy Algorithms

- Single Source Shortest Paths
  - We initialize the shortest-path estimates and predecessors by the following θ (V) -time procedure:

```
INITIALIZE-SINGLE-SOURCE(G, s)
1  for each vertex v ∈ G.V
2      v.d = ∞
3      v.π = NIL
4  s.d = 0
```

Greedy Algorithms

- Single Source Shortest Paths
  - The process of relaxing an edge (u, v) consists of testing whether we can improve the shortest path to v found so far by going through u and, if so, updating v.d and v.π.

Greedy Algorithms

- Single Source Shortest Paths
  - The process of relaxing an edge (u, v) consists of testing whether we can improve the shortest path to v found so far by going through u and, if so, updating v.d and v.π.
  - A relaxation step may decrease the value of the shortest-path estimate v.d and update v's predecessor attribute v.π.

Greedy Algorithms

- Single Source Shortest Paths
  - The process of relaxing an edge (u, v) consists of testing whether we can improve the shortest path to v found so far by going through u and, if so, updating v.d and v.π.
  - A relaxation step may decrease the value of the shortest-path estimate v.d and update v's predecessor attribute v.π.
  - The following code performs a relaxation step on edge (u, v) in O(1) time:

Greedy Algorithms

- Single Source Shortest Paths

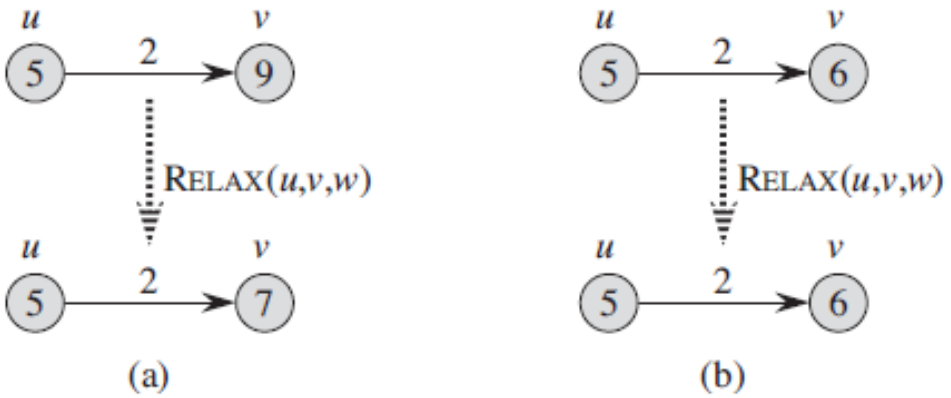


Figure 24.3 Relaxing an edge (u, v) with weight $w(u, v) = 2$. The shortest-path estimate of each vertex appears within the vertex. **(a)** Because $v.d > u.d + w(u, v)$ prior to relaxation, the value of $v.d$ decreases. **(b)** Here, $v.d \leq u.d + w(u, v)$ before relaxing the edge, and so the relaxation step leaves $v.d$ unchanged.

```
RELAX(u, v, w)
1  if v.d > u.d + w(u, v)
2        v.d = u.d + w(u, v)
3        v.π = u
```

Greedy Algorithms

- Single Source Shortest Paths - Dijkstra's Algorithm

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```

1. O(V)
2. O(1)
3. O(|V|)
4. O(|V|+1)
5. O(|V|*lg |V|)
6. O(V)
7 & 8. O(|E|*lg |V|)

O(E+V)*lg V
Or
O(E)*lg V

S is a set of vertices whose final shortest-path weights from the source s have already been determined.

Greedy Algorithms

- Single Source Shortest Paths - Dijkstra's Algorithm

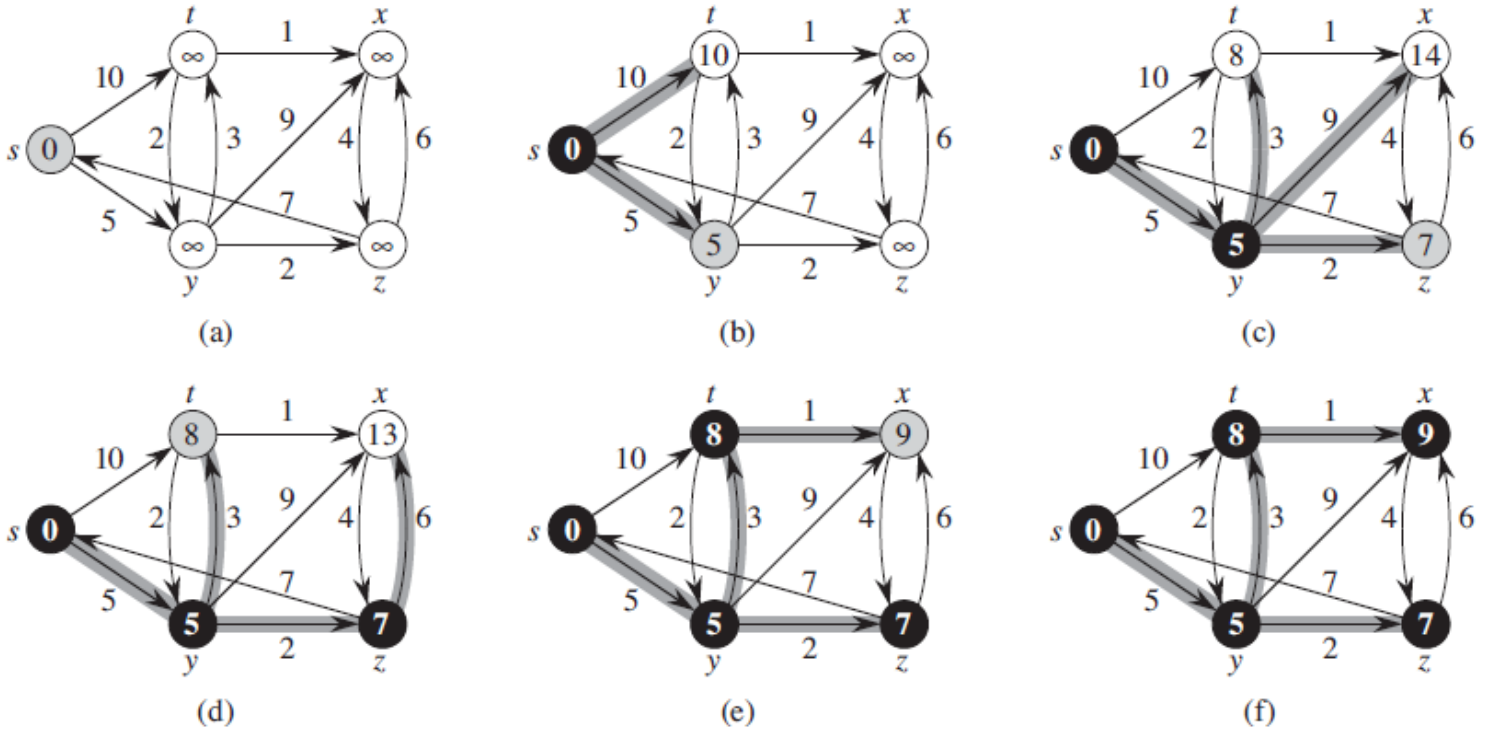


Figure 24.6 The execution of Dijkstra's algorithm. The source s is the leftmost vertex. The shortest-path estimates appear within the vertices, and shaded edges indicate predecessor values. Black vertices are in the set S, and white vertices are in the min-priority queue $Q = V - S$. **(a)** The situation just before the first iteration of the **while** loop of lines 4–8. The shaded vertex has the minimum d value and is chosen as vertex u in line 5. **(b)–(f)** The situation after each successive iteration of the **while** loop. The shaded vertex in each part is chosen as vertex u in line 5 of the next iteration. The d values and predecessors shown in part (f) are the final values.

Greedy Algorithms

- Single Source Shortest Paths - Dijkstra's Algorithm

How fast is Dijkstra's algorithm? It maintains the min-priority queue Q by calling three priority-queue operations: INSERT (implicit in line 3), EXTRACT-MIN (line 5), and DECREASE-KEY (implicit in RELAX, which is called in line 8). The algorithm calls both INSERT and EXTRACT-MIN once per vertex. Because each vertex $u \in V$ is added to set S exactly once, each edge in the adjacency list $Adj[u]$ is examined in the **for** loop of lines 7–8 exactly once during the course of the algorithm. Since the total number of edges in all the adjacency lists is $|E|$, this **for** loop iterates a total of $|E|$ times, and thus the algorithm calls DECREASE-KEY at most $|E|$ times overall. (Observe once again that we are using aggregate analysis.)

The running time of Dijkstra's algorithm depends on how we implement the min-priority queue. Consider first the case in which we maintain the min-priority queue by taking advantage of the vertices being numbered 1 to $|V|$. We simply store $v.d$ in the vth entry of an array. Each INSERT and DECREASE-KEY operation takes $O(1)$ time, and each EXTRACT-MIN operation takes $O(V)$ time (since we have to search through the entire array), for a total time of $O(V^2 + E) = O(V^2)$.

Greedy Algorithms

- Single Source Shortest Paths - Dijkstra's Algorithm

we can improve the algorithm by implementing the min-priority queue with a binary min-heap. (As discussed in Section 6.5, the implementation should make sure that vertices and corresponding heap elements maintain handles to each other.) Each EXTRACT-MIN operation then takes time $O(\lg V)$. As before, there are $|V|$ such operations. The time to build the binary min-heap is $O(V)$. Each DECREASE-KEY operation takes time $O(\lg V)$, and there are still at most $|E|$ such operations. The total running time is therefore $O((V + E) \lg V)$, which is $O(E \lg V)$ if all vertices are reachable from the source. This running time improves upon the straightforward $O(V^2)$-time implementation if $E = o(V^2/\lg V)$.

We can in fact achieve a running time of $O(V \lg V + E)$ by implementing the min-priority queue with a Fibonacci heap (see Chapter 19). The amortized cost of each of the $|V|$ EXTRACT-MIN operations is $O(\lg V)$, and each DECREASE-KEY call, of which there are at most $|E|$, takes only $O(1)$ amortized time. Historically, the development of Fibonacci heaps was motivated by the observation that Dijkstra's algorithm typically makes many more DECREASE-KEY calls than EXTRACT-MIN calls, so that any method of reducing the amortized time of each DECREASE-KEY operation to $o(\lg V)$ without increasing the amortized time of EXTRACT-MIN would yield an asymptotically faster implementation than with binary heaps.

Disclaimer

- Contents of this presentation are not original and they have been prepared from various sources just for the teaching purpose.