

# CS-203 – LESSON 10 – GREEDY ALGORITHMS

**GIUSEPPE TURINI**

# TABLE OF CONTENTS

## The Greedy Approach

- The Change-Making Problem

- Optimization Problems and the Greedy Approach

## Minimum Spanning Trees

- Spanning Trees, Minimum Spanning Trees, and the MST Problem

- Prim's Algorithm to Build a MST

- Kruskal's Algorithm to Build a MST

## Single-Source Shortest-Paths

- Single-Source Shortest-Paths Problem, and All-Pairs Shortest-Paths Problem

- Dijkstra's Algorithm for the SSSP Problem, and Floyd's Algorithm for the APSP Problem

## Huffman Trees and Codes

# STUDY GUIDE

## Study Material:

- These slides.
- Your notes.
- \* "Introduction to the Design and Analysis of Algorithms (3<sup>rd</sup> Ed.)", chap. 9, pp. 315-344.

## Practice Exercises:

- Exercises from \*: 9.1.1-5, 9.1.9-15, 9.2.1-4, 9.2.10, 9.2.12, 9.3.1-5.

## Additional Resources:

- "Introduction to Algorithms (3<sup>rd</sup> Ed.)", chap. 16, 23-24, pp. 414-450, 624-683.
- "Foundations of Algorithms (5<sup>th</sup> Ed.)", chap. 4, pp. 151-202.
- VisuAlgo

# THE GREEDY APPROACH

The “*greedy approach*” builds a solution step-by-step using a sequence of choices that individually look like a “*best-choice*”, and it is applicable to optimization problems only.

For some problems, the “*greedy approach*” yields an optimal solution for every instance. Otherwise, it can be used to achieve fast approximate solutions.

The “*greedy approach*” yields an optimal solution for the following problems:

- Change-Making Problem (with “standard” coin denominations).
- Minimum Spanning Tree (MST), and Single-Source Shortest-Paths (SSSP).
- Simple scheduling problems, and Huffman codes.

The “*greedy approach*” yields an approximate solution for the following problems:

- Traveling Salesman Problem (TSP), Knapsack Problem.
- Other combinatorial optimization problems.

# CHANGE-MAKING PROBLEM

**Problem:** Given unlimited amounts of coins of denominations  $d_1 > \dots > d_m$ , give change for the amount  $n$  with the least number of coins.

**Example:** With these coins denominations available:  $d_1 = 25$  c,  $d_2 = 10$  c,  $d_3 = 5$  c,  $d_4 = 1$  c  
Give change for the amount  $n = 48$  c

The “greedy approach” solution is:  $1 \times d_1, 2 \times d_2, 3 \times d_4$

The “greedy approach” strategy consists in iteratively selecting the denomination that reduces the remaining amount the most, until the remaining amount is 0.

**Note:** This “greedy approach” strategy leads to an optimal solution for any amount but only for a “standard” set of denominations, but not for “non-standard” coin denominations (e.g.,  $d_1 = 25$  c,  $d_2 = 10$  c,  $d_3 = 1$  c, and  $n = 30$  c).

# THE GREEDY APPROACH

The “*greedy approach*” incrementally builds a solution step-by-step using a sequence of choices that individually look like a “*best-choice*” and must be:

- **Feasible** (doable in terms of computation and computer programming).
- **Locally optimal** (the best choice looking at the current problem status).
- **Irrevocable** (undoable).

The “*greedy approach*” is applicable to optimization problems only.

For some problems, the “*greedy approach*” yields an optimal solution for every instance.

Otherwise, it can be used to achieve fast approximate solutions.

# OPTIMIZATION PROBLEMS

**Optimization Problem:** Finding the best solution from all feasible solutions.

This usually means to **minimize/maximize a target function** (e.g., a distance function, a value function, a cost function, etc.), usually subject to some constraints (e.g., visiting all vertices in a graph, not exceeding a threshold, etc.).

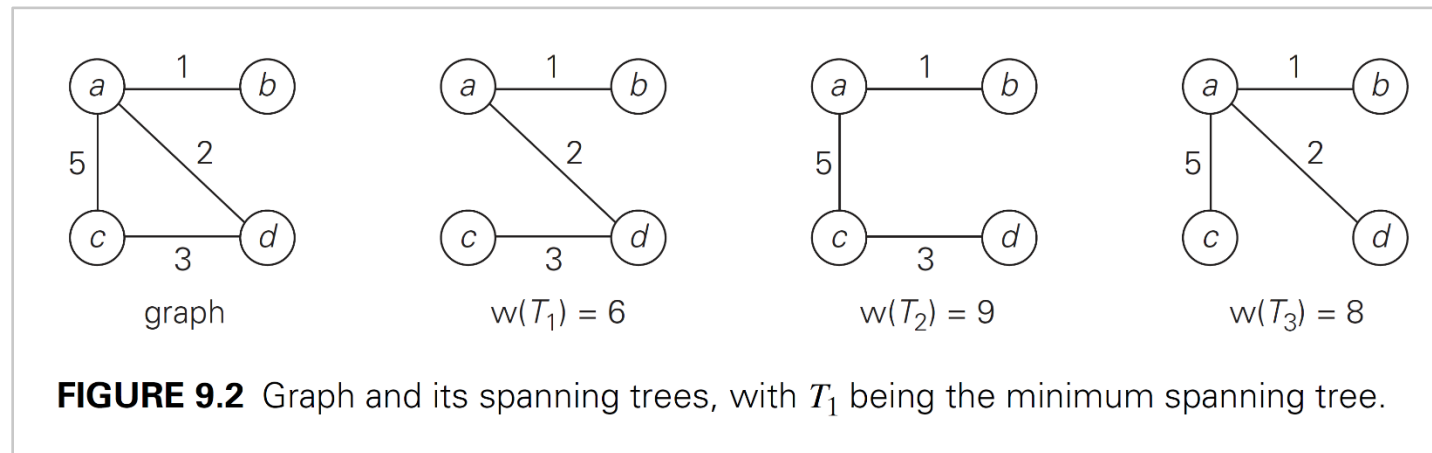
Optimization problems can be divided into 2 categories:

- **Discrete Optimization:** An optimization problem with discrete variables, in which an object (e.g., an integer value, a permutation, a graph, etc.) must be found from a countable set.
- **Continuous Optimization:** A problem with continuous variables, in which an optimal value from a continuous function must be found. These can include “*constrained problems*” and “*multimodal problems*”.

# MINIMUM SPANNING TREES

**Spanning Tree:** A spanning tree of an undirected connected graph  $G$  is its connected acyclic subgraph (*i.e.*, a tree) that contains all the vertices of the graph  $G$ .

**Minimum Spanning Tree:** A minimum spanning tree of a weighted undirected connected graph  $G$  is a spanning tree of  $G$  with minimum total weight (*i.e.*, the sum of all its edge weights).





# THE MST TREE PROBLEM

The minimum spanning tree (MST) problem is the problem of finding a MST for a given weighted connected graph.

To build a MST of a given graph  $G$  by using exhaustive search there are two obstacles:

- The number of spanning trees of a given graph grows exponentially with the graph size.
- Generating all spanning trees of a given graph is difficult.

# PRIM'S ALGORITHM FOR MST

Prim's algorithm builds a MST using a **sequence of expanding subtrees**:

- The initial tree  $T$  ( $T = \langle V_T, E_T \rangle$ ) in such a sequence consists of a single graph vertex, selected arbitrarily from the set of vertices  $V$  of the given graph  $G$  ( $G = \langle V_G, E_G \rangle$ ).
- On each iteration, Prim's algorithm expand the tree  $T$  by attaching to it its "nearest" vertex (connected by an edge with smallest weight) in graph  $G$  but not in tree  $T$ .
- Prim's algorithm stops after all vertices in graph  $G$  have been included in tree  $T$ .

**Note:** Because Prim's algorithm expands the tree 1 vertex at time, it runs exactly  $n-1$  iterations (with  $n$  being the number of vertices in the given graph  $G$ ).

**Note:** The MST generated by Prim's algorithm is represented just by the set of edges used for the tree expansions (*i.e.*,  $E_T$ ).

# PRIM'S ALGORITHM FOR MST (2)

## **ALGORITHM** *Prim*( $G$ )

//Prim's algorithm for constructing a minimum spanning tree

//Input: A weighted connected graph  $G = \langle V, E \rangle$

//Output:  $E_T$ , the set of edges composing a minimum spanning tree of  $G$

$V_T \leftarrow \{v_0\}$  //the set of tree vertices can be initialized with any vertex

$E_T \leftarrow \emptyset$

**for**  $i \leftarrow 1$  **to**  $|V| - 1$  **do**

    find a minimum-weight edge  $e^* = (v^*, u^*)$  among all the edges  $(v, u)$

    such that  $v$  is in  $V_T$  and  $u$  is in  $V - V_T$

$V_T \leftarrow V_T \cup \{u^*\}$

$E_T \leftarrow E_T \cup \{e^*\}$

**return**  $E_T$

# PRIM'S ALGORITHM FOR MST (3)

Prim's algorithm requires that each vertex in  $G$  but not in  $T$  is associated with the shortest edge connecting it to the current tree  $T$ .

This is done by labeling each graph vertex not selected yet with 2 data pieces:

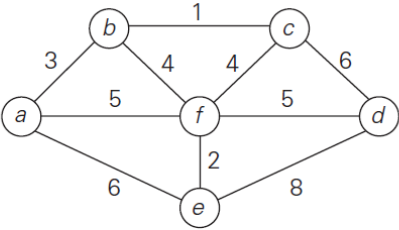
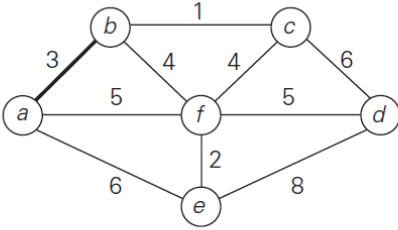
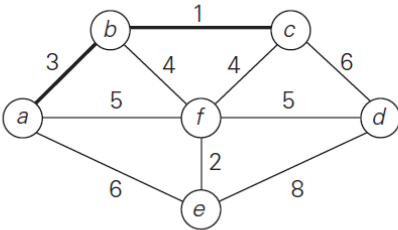
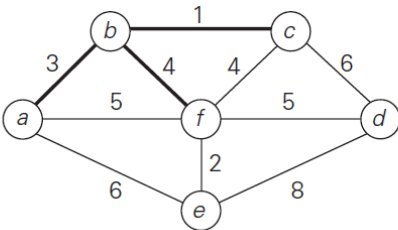
- The nearest tree vertex.
- The weight of the corresponding edge.

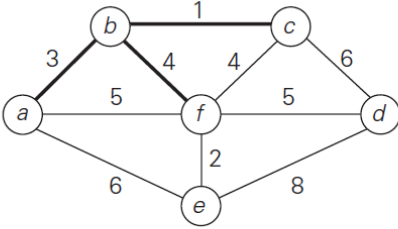
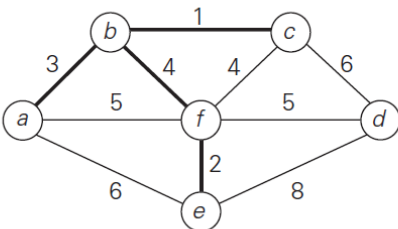
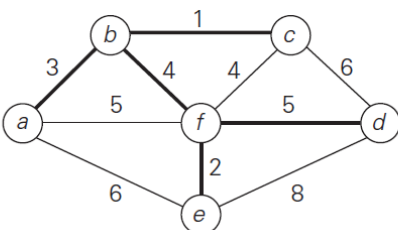
Graph vertices not selected yet, and not adjacent to any tree vertex are labeled with special markers (e.g., null, infinity, etc.).

With such labels, find the next graph vertex  $u^*$  to expand the current tree, and then:

- Move  $u^*$  from the graph vertices not selected yet to the tree vertices.
- For each remaining graph vertex  $u$  not selected yet, that is connected to  $u^*$  by a shorter edge than its current label, update its label considering  $u^*$ .

# PRIM'S ALGORITHM FOR MST (4)

		
Tree vertices	Remaining vertices	Illustration
a(−, −)	<b>b(a, 3)</b> c(−, ∞) d(−, ∞) e(a, 6) f(a, 5)	
b(a, 3)	<b>c(b, 1)</b> d(−, ∞) e(a, 6) f(b, 4)	
c(b, 1)	d(c, 6) e(a, 6) <b>f(b, 4)</b>	

c(b, 1)	d(c, 6) e(a, 6) <b>f(b, 4)</b>	
f(b, 4)	d(f, 5) <b>e(f, 2)</b>	
e(f, 2)	<b>d(f, 5)</b>	
d(f, 5)		

**FIGURE 9.3** Application of Prim’s algorithm. The parenthesized labels of a vertex in the middle column indicate the nearest tree vertex and edge weight; selected vertices and edges are shown in bold.

# PRIM'S ALGORITHM FOR MST (5)

## Efficiency of Prim's Algorithm

Prim's algorithm efficiency depends on the data structure representing the graph, and on the priority queue implementation (to select the “nearest” graph vertex to the tree).

**Example:** For a graph using adjacency lists and the priority queue is a min-heap:

- The algorithm performs  $|V| - 1$  deletions of the nearest graph vertex.
- The algorithm makes  $|E|$  verifications to update graph vertices labels.
- The priority queue (min-heap) size will never exceeds  $|V|$ .
- Each priority queue (min-heap) operation (delete, update) is in  $O(\log |V|)$ .

So, in this case, the running time of Prim's algorithm is in:

$$(|V| - 1 + |E|) O(\log |V|) = O(|E| \log |V|)$$

because in a connected graph  $|V| - 1 \leq |E|$

# PRIM'S ALGORITHM FOR MST (6)

## Fringe and Unseen Vertices

During the execution of Prim's algorithm, we can categorize graph vertices not selected yet in 2 sets:

- **Fringe vertices** are unselected graph vertices adjacent to a tree vertex, and represent candidates from which the next tree vertex is selected.
- **Unseen vertices** are unselected graph vertices not-adjacent to a tree vertex, and represent vertices that will be affected by the algorithm at a later stage.

## Correctness of Prim's Algorithm

Prim's algorithm always yield a MST, and it can be proven by **mathematical induction**.

# KRUSKAL'S ALGORITHM FOR MST

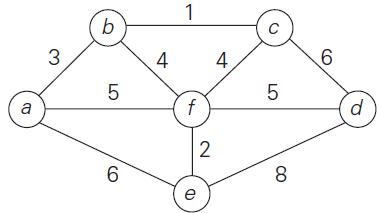
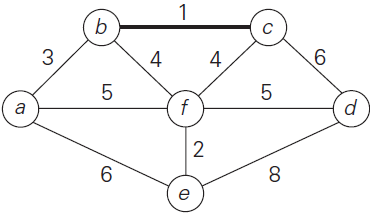
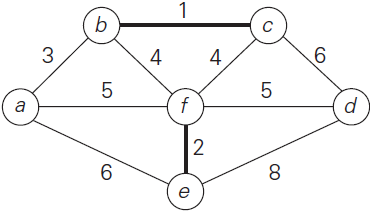
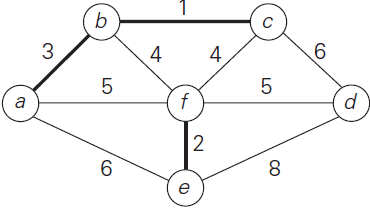
Kruskal's algorithm builds a MST selecting graph/tree 1 edge at a time:

- See a MST as an acyclic subgraph with  $|V|-1$  edges with minimal total weight.
- Sort graph edges in non-decreasing order of weight.
- Expand the MST 1 edge at a time by generating a series of  $n-1$  ( $n$  is the number of graph vertices) expanding forests  $F_1, F_2, \dots, F_{n-1}$ .
- On each iteration, select a graph edge from the sorted list that is: not selected yet, with minimum weight, and not creating a cycle in the current tree/forest (otherwise, skip that graph edge).

**Note:** During Kruskal's algorithm execution, some of the intermediate subgraphs could be disconnected.



# KRUSKAL'S ALGORITHM FOR MST (2)

		
Tree edges	Sorted list of edges	Illustration
	<b>bc</b> 1   ef 2 <b>ab</b> 3   bf 4   cf 4   af 5   df 5   ae 6   cd 6   de 8	
bc 1	bc 1 <b>ef</b> 2   ab 3   bf 4   cf 4   af 5   df 5   ae 6   cd 6   de 8	
ef 2	bc 1   ef 2 <b>ab</b> 3   bf 4   cf 4   af 5   df 5   ae 6   cd 6   de 8	

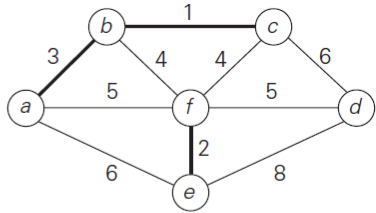
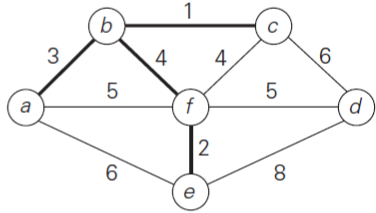
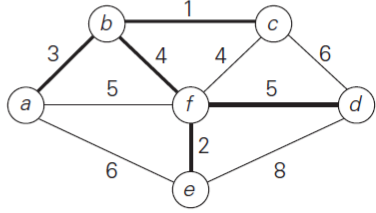
ef 2	bc 1   ef 2 <b>ab</b> 3   bf 4   cf 4   af 5   df 5   ae 6   cd 6   de 8	
ab 3	bc 1   ef 2   ab 3 <b>bf</b> 4   cf 4   af 5   df 5   ae 6   cd 6   de 8	
bf 4	bc 1   ef 2   ab 3   bf 4   cf 4   af 5 <b>df</b> 5   ae 6   cd 6   de 8	
df 5		

FIGURE 9.5 Application of Kruskal’s algorithm. Selected edges are shown in bold.

# KRUSKAL'S ALGORITHM FOR MST (3)

## Checking for Cycles Executing Kruskal's Algorithm

Kruskal's algorithm has to check whether the addition of the next edge to the current subgraph would create a cycle.

This happens if and only if the new edge connects 2 vertices already selected and belonging to the same connected component (in this case a tree).

There are efficient algorithms to check if 2 vertices belong to the same connected component (or tree in this case): these algorithms are called union-find algorithms.

## Efficiency of Kruskal's Algorithm

Using an efficient union-find algorithm, Kruskal's algorithm efficiency is dominated by the time needed to sort the graph edges, so the running time of Kruskal's algorithm is in:

$$O(|E| \log |E|)$$

# SINGLE-SOURCE SHORTEST-PATHS

The single-source shortest paths (SSSP) problem is the problem of: finding all the shortest paths to reach all vertices of a weighted connected graph, given a vertex (called the source).

**Note:** A variety of practical applications of the SSSP problem have made it very popular.

The most widely used applications are: transportation planning, network packet routing, social network connection-distance, speech recognition, document formatting, robotics, compilers, worker scheduling, videogame pathfinding.

**Note:** There is a similar problem, more general, that is called the all-pairs shortest-paths (APSP) that consists in finding the shortest paths to reach all pair of different vertices in a weighted connected graph.

# DIJKSTRA'S ALGORITHM FOR SSSP

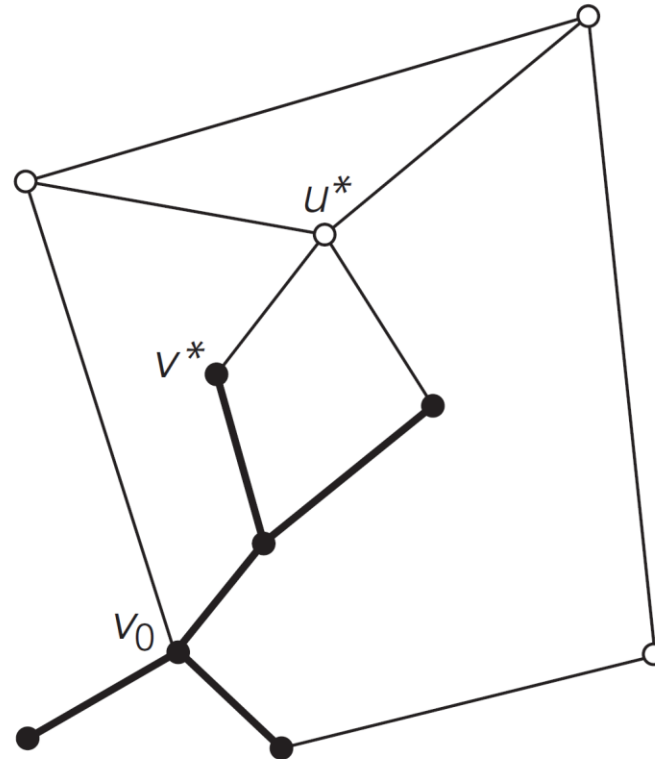
The best-known algorithm to solve the SSSP problem is called the Dijkstra Algorithm.

This algorithm is applicable to undirected/directed graphs with non-negative weights only!

Dijkstra's algorithm finds the shortest paths to the vertices of a graph in order of their distance from a given vertex/source.

- First, it finds the shortest path from the source to a vertex nearest to it.
- Then, to a second-nearest, and so on.
- In general, before its  $i^{\text{th}}$  iteration starts, the algorithm already found the shortest paths to  $i-1$  other vertices nearest to the source.
- The vertices selected/found so far (including the source) and the edges forming the shortest paths form a subtree  $T_i$  (at iteration  $i$ ).

# DIJKSTRA'S ALGORITHM FOR SSSP (2)



**FIGURE 9.10** Idea of Dijkstra's algorithm. The subtree of the shortest paths already found is shown in bold. The next nearest to the source  $v_0$  vertex,  $u^*$ , is selected by comparing the lengths of the subtree's paths increased by the distances to vertices adjacent to the subtree's vertices.

# DIJKSTRA'S ALGORITHM FOR SSSP (3)

Because all graph edge weights are non-negative, the next vertex nearest to the source can be found among the graph vertices adjacent to the vertices of  $T_i$ . These vertices are called "*fringe vertices*", and they are the candidates to select the next nearest vertex.

- To identify the  $i^{\text{th}}$  nearest vertex, the algorithm computes, for every fringe vertex  $u$ , the sum of the distance to the nearest vertex  $v$  in tree  $T_i$ .
- This sum is given by the weight of the edge  $(v,u)$  and the length  $d_{s-v}$  of the shortest path from the source  $s$  to  $v$  (previously determined by the algorithm).
- Then, the algorithm selects the vertex with the smallest such sum.

**Note:** The fact that it suffices to compare the lengths of such special paths is the central insight of Dijkstra's algorithm.

# DIJKSTRA'S ALGORITHM FOR SSSP (4)

To facilitate Dijkstra's algorithm operations, we label each vertex with 2 labels:

- The length  $d$  of the shortest path from the source to this vertex, as found by the algorithm so far (infinity if vertex has not been processed yet).
- The name of the next-to-last vertex of the shortest path from the source to this vertex, as found by the algorithm so far (null for unprocessed vertices, and for the source).

**Note:** The next-to-last vertex of the shortest path from the source to a vertex, is also the parent of that vertex in the tree being built representing the SSSP.

**Note:** With such vertex labeling, finding the next nearest vertex  $u^*$  becomes a simple task of finding the fringe vertex with the smallest  $d$  value (with ties broken arbitrarily).

# DIJKSTRA'S ALGORITHM FOR SSSP (5)

After we have found the next nearest vertex  $u^*$  (the fringe vertex with the smallest  $d$  value), we need to perform 2 operations:

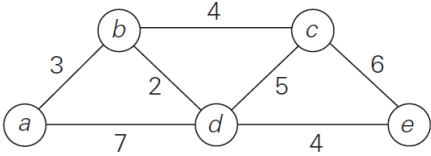
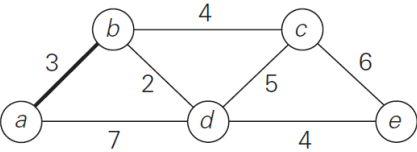
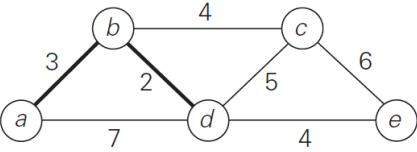
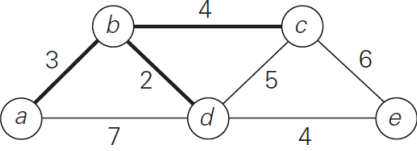
- Move  $u^*$  from the fringe to the set of tree vertices.
- For each other fringe vertex  $u$  connected to  $u^*$  by an edge of weight  $w(u^*, u)$  with:  
$$d_{s-u^*} + w(u^*, u) < d_{s-u}$$
  
update the labels of  $u$  by  $u^*$  and  $d_{s-u^*} + w(u^*, u)$ , respectively.

**Note:** The labeling and mechanics of Dijkstra's algorithm are quite similar to those used by Prim's algorithm (building expanding subtree, and selecting fringe vertex).

However, they solve different problems and therefore they operate with priorities/values computer differently.



# DIJKSTRA'S ALGORITHM FOR SSSP (6)

		
Tree vertices	Remaining vertices	Illustration
$a(-, 0)$	<b><math>b(a, 3)</math></b> $c(-, \infty)$ $d(a, 7)$ $e(-, \infty)$	
$b(a, 3)$	$c(b, 3 + 4)$ <b><math>d(b, 3 + 2)</math></b> $e(-, \infty)$	
$d(b, 5)$	<b><math>c(b, 7)</math></b> $e(d, 5 + 4)$	

$d(b, 5)$

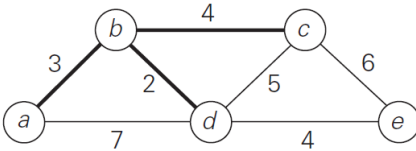
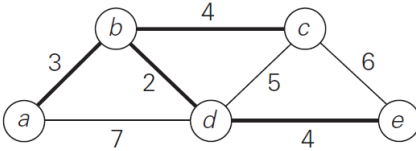
$c(b, 7)$

$e(d, 9)$

$c(b, 7)$

$e(d, 9)$

$e(d, 9)$

---

The shortest paths (identified by following nonnumeric labels backward from a destination vertex in the left column to the source) and their lengths (given by numeric labels of the tree vertices) are as follows:

from  $a$  to  $b$  :  $a - b$

of length 3

from  $a$  to  $d$  :  $a - b - d$

of length 5

from  $a$  to  $c$  :  $a - b - c$

of length 7

from  $a$  to  $e$  :  $a - b - d - e$

of length 9

**FIGURE 9.11** Application of Dijkstra's algorithm. The next closest vertex is shown in bold.

# DIJKSTRA'S ALGORITHM FOR SSSP (7)

## **ALGORITHM** *Dijkstra*( $G, s$ )

//Dijkstra's algorithm for single-source shortest paths

//Input: A weighted connected graph  $G = \langle V, E \rangle$  with nonnegative weights

// and its vertex  $s$

//Output: The length  $d_v$  of a shortest path from  $s$  to  $v$

// and its penultimate vertex  $p_v$  for every vertex  $v$  in  $V$

*Initialize*( $Q$ ) //initialize priority queue to empty

**for** every vertex  $v$  in  $V$

$d_v \leftarrow \infty$ ;  $p_v \leftarrow \mathbf{null}$

*Insert*( $Q, v, d_v$ ) //initialize vertex priority in the priority queue

$d_s \leftarrow 0$ ; *Decrease*( $Q, s, d_s$ ) //update priority of  $s$  with  $d_s$

$V_T \leftarrow \emptyset$

**for**  $i \leftarrow 0$  **to**  $|V| - 1$  **do**

$u^* \leftarrow \text{DeleteMin}(Q)$  //delete the minimum priority element

$V_T \leftarrow V_T \cup \{u^*\}$

**for** every vertex  $u$  in  $V - V_T$  that is adjacent to  $u^*$  **do**

**if**  $d_{u^*} + w(u^*, u) < d_u$

$d_u \leftarrow d_{u^*} + w(u^*, u)$ ;  $p_u \leftarrow u^*$

*Decrease*( $Q, u, d_u$ )

# DIJKSTRA'S ALGORITHM FOR SSSP (8)

## Efficiency of Dijkstra's Algorithm

The time efficiency of Dijkstra's algorithm depends on the data structures used to implement the priority queue and the weighted graph.

**Example:** For a graph using adjacency lists and the priority queue is a min-heap:

- The algorithm performs  $|V| - 1$  selections of the nearest fringe vertex.
- The algorithm makes maximum  $|E|$  updates to graph vertices labels.
- The priority queue (min-heap) size will never exceeds  $|V|$ .
- Each priority queue (min-heap) operation (delete, update) is in  $O(\log |V|)$ .

So, in this case, the running time of Dijkstra's algorithm is in:

$$(|V| - 1 + |E|) O(\log |V|) = O(|E| \log |V|)$$

because in a connected graph  $|V| - 1 \leq |E|$

