

## Lecture 13: Template Methods and Shortest Paths in a Weighted Graph

Path of Least Action

1

## Wholeness Statement

In a weighted graph, the shortest path algorithm finds the path between a given pair of vertices such that the sum of the weights of that path's edges is the minimum. Natural law always chooses the path of least action, the shortest path to the goal with no wasted effort.

2

## Template Method Pattern

Depth-first search is to graphs  
what the Euler tour is to binary  
trees

3

## Recall Our Earlier Example of the Template Method Pattern

- ◆ Generic algorithm that can be specialized by redefining certain steps
- ◆ Implemented by means of an abstract Java class
- ◆ Visit methods that can be redefined by subclasses
- ◆ Template method `eulerTour`
  - Recursively called on the left and right children
  - A `Result` object with fields `leftResult`, `rightResult` and `finalResult` keeps track of the output of the recursive calls to `eulerTour`

```
public abstract class EulerTour {
    protected BinaryTree tree;
    protected void visitExternal(Position p, Result r) {}
    protected void visitPreOrder(Position p, Result r) {}
    protected void visitInOrder(Position p, Result r) {}
    protected void visitPostOrder(Position p, Result r) {}
    protected Object eulerTour(Position p) {
        Result r = new Result();
        if (tree.isExternal(p)) { visitExternal(p, r); }
        else {
            visitPreOrder(p, r);
            r.leftResult = eulerTour(tree.leftChild(p));
            visitInOrder(p, r);
            r.rightResult = eulerTour(tree.rightChild(p));
            visitPostOrder(p, r);
            return r.finalResult;
        }
    }
}
```

4

## Specializations of EulerTour

- ◆ We show how to specialize class `EulerTour` to evaluate an arithmetic expression
- ◆ Assumptions
  - External nodes store `Integer` objects
  - Internal nodes store `Operator` objects supporting method `operation(Integer, Integer)`

```
public class EvaluateExpression
    extends EulerTour {
    protected void visitExternal(Position p, Result r) {
        r.finalResult = (Integer) p.element();
    }
    protected void visitPostOrder(Position p, Result r) {
        Operator op = (Operator) p.element();
        r.finalResult = op.operation(
            (Integer) r.leftResult,
            (Integer) r.rightResult
        );
    }
    ...
}
```

5

## DFS Template Example

6

## Depth-First Search (review)

- ◆ Depth-first search (DFS) is a general technique for traversing a graph
- ◆ A DFS traversal of a graph  $G$ 
  - Visits all the vertices and edges of  $G$
  - Determines whether  $G$  is connected
  - Computes the connected components of  $G$
  - Computes a spanning forest of  $G$

7

## Template Version of DFS

<b>Algorithm <math>DFS(G)</math></b> <b>Input</b> graph $G$ <b>Output</b> the edges of $G$ are labeled as discovery edges and back edges  <b>initResult(<math>G</math>)</b> <b>for all</b> $u \in G.vertices()$ $setLabel(u, UNEXPLORED)$ <b>for all</b> $e \in G.edges()$ $setLabel(e, UNEXPLORED)$ <b>for all</b> $v \in G.vertices()$ <b>if</b> $getLabel(v) = UNEXPLORED$ $preComponentVisit(G, v)$ $DFS(G, v)$ $postComponentVisit(G, v)$ <b>result(<math>G</math>)</b>	<b>Algorithm <math>DFS(G, v)</math></b> $setLabel(v, VISITED)$ $startVertexVisit(v)$ <b>for all</b> $e \in G.incidentEdges(v)$ <b>if</b> $getLabel(e) = UNEXPLORED$ $w \leftarrow opposite(v, e)$ <b>if</b> $getLabel(w) = UNEXPLORED$ $setLabel(e, DISCOVERY)$ $preDiscoveryTraversal(G, v, e, w)$ $DFS(G, w)$ $postDiscoveryTraversal(G, v, e, w)$ <b>else</b> $setLabel(e, BACK)$ $backTraversal(G, v, e, w)$ $finishVertexVisit(G, v)$
--	--

8

## Path Finding Override hook operations

<b>Algorithm <math>DFS(G, v)</math></b> $setLabel(v, VISITED)$ $startVertexVisit(v)$ <b>for all</b> $e \in G.incidentEdges(v)$ <b>if</b> $getLabel(e) = UNEXPLORED$ $w \leftarrow opposite(v, e)$ <b>if</b> $getLabel(w) = UNEXPLORED$ $setLabel(e, DISCOVERY)$ $preDiscoveryTraversal(G, v, e, w)$ $DFS(G, w)$ $postDiscoveryTraversal(G, v, e, w)$ <b>else</b> $setLabel(e, BACK)$ $backEdgeVisit(G, v, e, w)$ $finishVertexVisit(G, v)$	<b>Algorithm <math>pathDFS(G, v, z)</math></b> $setLabel(v, VISITED)$ $S.push(v)$ <b>if</b> $v = z$ <b>then</b> $path \leftarrow S.elements()$ <b>for all</b> $e \in G.incidentEdges(v)$ <b>do</b> <b>if</b> $getLabel(e) = UNEXPLORED$ <b>then</b> $w \leftarrow opposite(v, e)$ <b>if</b> $getLabel(w) = UNEXPLORED$ <b>then</b> $setLabel(e, DISCOVERY)$ $S.push(e)$ $pathDFS(G, w, z)$ $S.pop()$ { $e$ gets popped } <b>else</b> $setLabel(e, BACK)$ $S.pop()$ { $v$ gets popped }
--	---

9

## Overriding hook methods in a subclass FindSimplePath

**Algorithm  $findPath(G, u, v)$**   
 $S \leftarrow$  new empty stack { $S$  is a subclass field}  
 $z \leftarrow v$  { $z$  is a subclass field & is the target vertex}  
**for all**  $u \in G.vertices()$   
      $setLabel(u, UNEXPLORED)$   
**for all**  $e \in G.edges()$   
      $setLabel(e, UNEXPLORED)$   
 $DFS(G, u)$   
 $return(path)$

**Algorithm  $startVertexVisit(G, v)$**   
 $S.push(v)$   
**if**  $v = z$  **then** { $z$  is a subclass field & is the target}  
      $path \leftarrow S.elements()$  { $path$  is a subclass field & is the result}

**Algorithm  $preDiscoveryTraversal(G, v, e, w)$**   
 $S.push(e)$

**Algorithm  $postDiscoveryTraversal(G, v, e, w)$**   
 $S.pop()$  {pop  $e$  off the stack}

**Algorithm  $finishVertexVisit(G, v)$**   
 $S.pop()$  {pop  $v$  off the stack}

10

## Exercise: Cycle Finding Override hook operations

<b>Algorithm <math>DFS(G, v)</math></b> $setLabel(v, VISITED)$ $startVertexVisit(v)$ <b>for all</b> $e \in G.incidentEdges(v)$ <b>if</b> $getLabel(e) = UNEXPLORED$ $w \leftarrow opposite(v, e)$ <b>if</b> $getLabel(w) = UNEXPLORED$ $setLabel(e, DISCOVERY)$ $preDiscoveryTraversal(G, v, e, w)$ $DFS(G, w)$ $postDiscoveryTraversal(G, v, e, w)$ <b>else</b> $setLabel(e, BACK)$ $backEdgeVisit(G, v, e, w)$ $finishVertexVisit(G, v)$	<b>Algorithm <math>cycleDFS(G, v)</math></b> $setLabel(v, VISITED)$ <b>if</b> $cycle == null$ <b>then return</b> $S.push(v)$ <b>for all</b> $e \in G.incidentEdges(v)$ <b>if</b> $getLabel(e) = UNEXPLORED$ $w \leftarrow opposite(v, e)$ $S.push(e)$ <b>if</b> $getLabel(w) = UNEXPLORED$ $setLabel(e, DISCOVERY)$ $cycleDFS(G, w)$ $S.pop()$ <b>else</b> $setLabel(e, BACK)$ $cycle \leftarrow$ new empty sequence $o \leftarrow w$ <b>repeat</b> $cycle.insertLast(o)$ $o \leftarrow S.pop()$ <b>until</b> $o = w$ $S.pop()$
--	---

11

## Overriding template methods in subclass FindCycles

**Algorithm  $startVertexVisit(G, v)$**   
**if**  $\neg cycleFound$  **then**  $S.push(v)$

**Algorithm  $finishVertexVisit(G, v)$**   
**if**  $\neg cycleFound$  **then**  $S.pop()$

**Algorithm  $preDiscoveryTraversal(G, v, e, w)$**   
**if**  $\neg cycleFound$  **then**  $S.push(e)$

**Algorithm  $postDiscoveryTraversal(G, v, e, w)$**   
**if**  $\neg cycleFound$  **then**  $S.pop()$

**Algorithm  $backEdgeVisit(G, v, e, w)$**   
**if**  $\neg cycleFound$  **then**  
      $S.push(e)$   
      $cycle \leftarrow$  new empty sequence  
      $o \leftarrow w$   
     **repeat**  
          $cycle.insertLast(o)$   
          $o \leftarrow S.pop()$   
     **until**  $o = w$   
      $cycleFound \leftarrow true$  { $cycleFound$  is a subclass field, initially false}

12

- ◆ What additional method(s) do we need to override?

13

## BFS Template Method

14

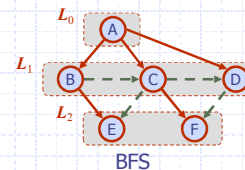
## Applications of BFS (review)

- ◆ Using the template method pattern, we can specialize the BFS traversal of a graph  $G$  to solve the following problems in  $O(n + m)$  time
  - Compute the connected components of  $G$
  - Compute a spanning forest of  $G$
  - Find a simple cycle in  $G$ , or report that  $G$  is a forest
  - Given two vertices of  $G$ , find a path in  $G$  between them with the minimum number of edges, or report that no such path exists

15

## BFS Levels

When actually implemented, the levels are normally merged into a single sequence/queue



16

## BFS Algorithm (revised)

- ◆ The BFS algorithm using a single list/sequence  $L$

**Algorithm  $BFS(G)$**   
**Input** graph  $G$   
**Output** labeling of the edges and partition of the vertices of  $G$   
 for all  $u \in G.vertices()$   
    $setLabel(u, UNEXPLORED)$   
 for all  $e \in G.edges()$   
    $setLabel(e, UNEXPLORED)$   
 for all  $v \in G.vertices()$   
   if  $getLabel(v) = UNEXPLORED$   
    $BFS(G, v)$

**Algorithm  $BFS(G, s)$**   
 $L \leftarrow$  new empty List  
 $L.insertLast(s)$   
 $setLabel(s, VISITED)$   
 while  $\neg L.isEmpty()$   
    $v \leftarrow L.remove(L.first())$   
   for all  $e \in G.incidentEdges(v)$   
   if  $getLabel(e) = UNEXPLORED$  then  
    $w \leftarrow opposite(v, e)$   
   if  $getLabel(w) = UNEXPLORED$  then  
    $setLabel(e, DISCOVERY)$   
    $setLabel(w, VISITED)$   
    $L.insertLast(w)$   
   else  
    $setLabel(e, CROSS)$

17

## Main Point

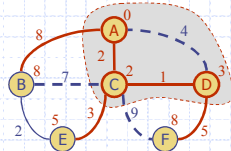
1. The Template Method Pattern implements the changing and non-changing parts of an algorithm in the superclass; it then allows subclasses to override certain (changeable) steps of an algorithm without modifying the basic structure of the original algorithm.

The changing and non-changing aspects of creation are unified in the field pure intelligence that we experience every day during our TM program.

18

## Shortest Paths

Path of Least Action



19

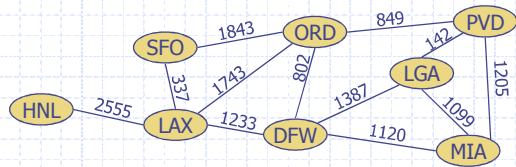
## Outline and Reading

- ◆ Weighted graphs (§7.1)
  - Shortest path problem
  - Shortest path properties
- ◆ Dijkstra's algorithm (§7.1.1)
  - Algorithm
  - Edge relaxation

20

## Weighted Graphs

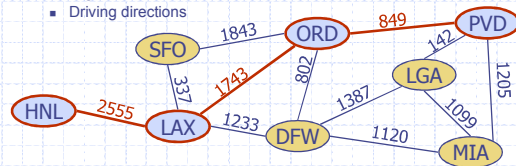
- ◆ In a weighted graph, each edge has an associated numerical value, called the weight of the edge
- ◆ Edge weights may represent, distances, costs, etc.
- ◆ Example:
  - In a flight route graph, the weight of an edge represents the distance in miles between the endpoint airports



21

## Shortest Path Problem

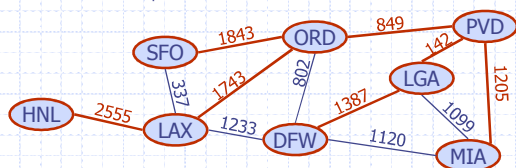
- ◆ Given a weighted graph and two vertices  $u$  and  $v$ , we want to find a path of minimum total weight between  $u$  and  $v$ .
  - Length of a path is the sum of the weights of its edges.
- ◆ Example:
  - Shortest path between Providence and Honolulu
- ◆ Applications
  - Internet packet routing
  - Flight reservations
  - Driving directions



22

## Shortest Path Properties

- Property 1:  
A subpath of a shortest path is itself a shortest path
- Property 2:  
There is a tree of shortest paths from a start vertex to all the other vertices
- Example:  
Tree of shortest paths from Providence



23

## Dijkstra's Algorithm

- ◆ The distance of a vertex  $v$  from a vertex  $s$  is the length of a shortest path between  $s$  and  $v$
- ◆ Dijkstra's algorithm computes the shortest distances of all the vertices from a given start vertex  $s$
- ◆ Assumptions:
  - the graph is connected
  - the edges are undirected
  - the edge weights are **nonnegative**

24

## Dijkstra's Algorithm (Informal)



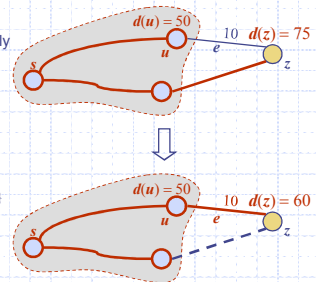
- ◆ We grow a “**cloud**” of vertices, beginning with  $s$  and eventually covering all the vertices
- ◆ We store with each vertex  $v$  a label  $d(v)$ 
  - represents the distance of  $v$  from  $s$  in the subgraph consisting of the cloud and its adjacent vertices
- ◆ At each step
  - We add to the cloud a vertex  $u$ 
    - outside the cloud
    - with the smallest distance label,  $d(u)$
  - Then we update the labels of the vertices adjacent to  $u$

25

## Edge Relaxation

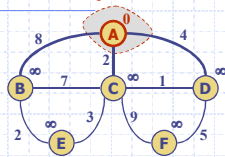


- ◆ Consider an edge  $e = (u, z)$  such that
  - $u$  is the vertex most recently added to the cloud
  - $z$  is not in the cloud
- ◆ The relaxation of edge  $e$  updates distance  $d(z)$  as follows:
 
$$d(z) \leftarrow \min\{d(z), d(u) + \text{weight}(e)\}$$



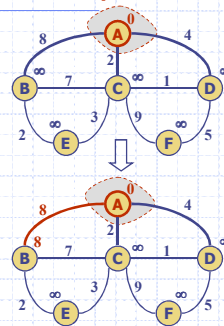
26

## Example



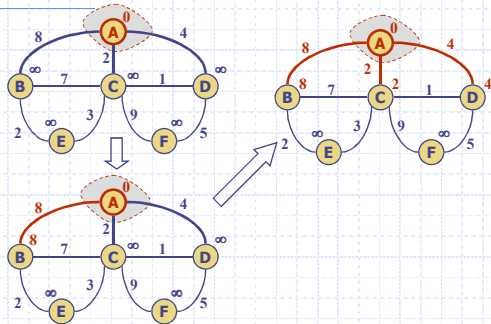
27

## Example



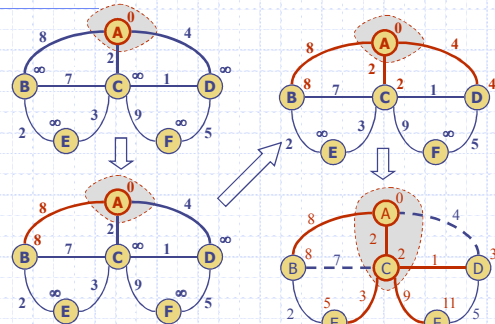
28

## Example



29

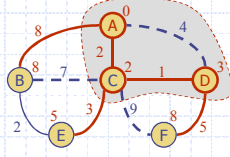
## Example



30

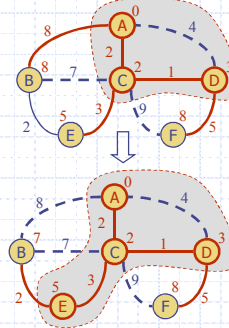


## Example



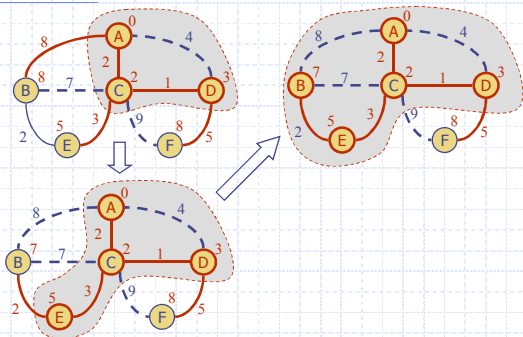
31

## Example



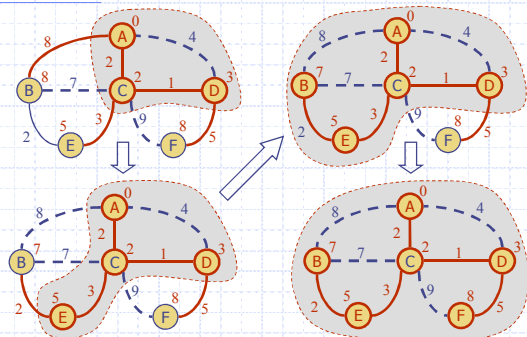
32

## Example



33

## Example



34

## Dijkstra's Algorithm (version 1)

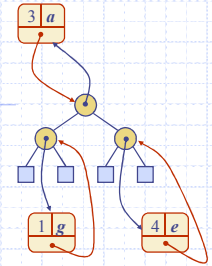


- A priority queue stores the vertices outside the cloud
  - Key: distance
  - Element: vertex
- We store the distance with each vertex:
  - Distance (d(v) label)
- In the Relax step we need to change the location of the vertex z in the priority queue
  - How do we do this efficiently?

**Algorithm *DijkstraDistances*(G, s)**  
 $Q \leftarrow$  new heap-based priority queue  
**for all**  $v \in G.vertices()$   
   **if**  $v = s$   
      $setDistance(v, 0)$   
   **else**  
      $setDistance(v, \infty)$   
    $Q.insertItem(getDistance(v), v)$   
**while**  $\neg Q.isEmpty()$   
    $u \leftarrow Q.removeMin()$   
   **for all**  $e \in G.incidentEdges(u)$   
     { relax edge  $e$  }  
      $z \leftarrow G.opposite(u, e)$   
      $r \leftarrow getDistance(u) + weight(e)$   
     **if**  $r < getDistance(z)$   
        $setDistance(z, r)$   
        $Q.replaceKey(z, r)$  {new method}

35

## Locators



36

## Outline and Reading

- ◆ Locators (§2.4.4)
- ◆ Locator-based methods (§2.4.4)
- ◆ Implementation
- ◆ Positions vs. Locators

37

## Locator Design Pattern

- ◆ A mechanism for maintaining the association between an item and its current position in a container
- ◆ When an item is inserted into a container, we get back a locator for that item
  - this locator can be used later to refer to that same item in the container even if its position has changed
  - an item's position may change when items are inserted or an item's key is changed

38

## Locators

- ◆ identifies and tracks a (key, element) item within a data structure
- ◆ sticks with a specific item, even if that element changes its position in the data structure
- ◆ Intuitive notion:
  - claim check
  - reservation number
- ◆ Methods of the Locator ADT:
  - **key()**: returns the key of the item associated with the locator
  - **element()**: returns the element of the item associated with the locator
- ◆ Application example:
  - Orders to purchase and sell a given stock are stored in two priority queues (sell orders and buy orders)
    - the key of an order is the price
    - the element is the number of shares
  - When an order is placed, a Locator to it is returned
  - Given a Locator, an order can be canceled or modified

39

## Positions vs. Locators

- ◆ Position
  - represents a "place" in a data structure
  - related to other positions in the data structure (e.g., previous/next or parent/child)
  - implemented as a node or an array cell
- ◆ Position-based ADTs are fundamental data storage schemes (e.g., list, sequence, and tree)
- ◆ Locator
  - identifies and tracks a (key, element) item
  - unrelated to other locators in the data structure
  - implemented as an object storing the item and its position in the underlying structure
- ◆ Key-based ADTs can be augmented with locator-based methods (e.g., list, sequence and dictionary)

40

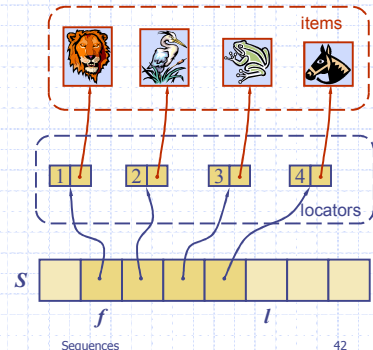
## Locator-based Methods added to the Priority Queue

- ◆ Locator-based methods:
  - **insert(k, o)**: inserts the item (k, o) and returns a locator for it
  - **min()**: returns the locator of an item with smallest key
  - **remove(l)**: remove the item with locator l
  - **replaceKey(l, k)**: replaces the key of the item with locator l
  - **replaceElement(l, o)**: replaces with o the element of the item with locator l
  - **locators()**: returns an iterator over the locators of the items in the priority queue (or dictionary)

41

## Array-based Implementation

- ◆ We use an array storing locators
- ◆ A position object stores:
  - Item (key, element)
  - Rank/Index
- ◆ Indices *f* and *l* keep track of first and last locators



42

## Dijkstra's Algorithm with Locators



- ◆ A priority queue stores the vertices outside the cloud
  - Key: distance
  - Element: vertex
- ◆ Locator-based methods
  - `insert(k,e)` returns a locator
  - `replaceKey(l,k)` changes the key of an item
- ◆ We store two labels with each vertex:
  - Distance ( $d(v)$  label)
  - locator in priority queue

**Algorithm *DijkstraDistances*( $G, s$ )**

```

Q ← new heap-based priority queue
for all v ∈ G.vertices()
    if v = s
        setDistance(v, 0)
    else
        setDistance(v, ∞)
l ← Q.insert(getDistance(v), v)
setLocator(v, l)
while ¬Q.isEmpty()
    u ← Q.removeMin()
    for all e ∈ G.incidentEdges(u)
        { relax edge e }
        z ← G.opposite(u, e)
        r ← getDistance(u) + weight(e)
        if r < getDistance(z)
            setDistance(z, r)
            Q.replaceKey(getLocator(z), r)
    
```

43

## Analysis



- ◆ Graph operations
  - Method `incidentEdges` is called once for each vertex
  - Recall that  $\sum \deg(v) = 2m$
- ◆ Label operations of vertices
  - We set/get the distance and locator labels of vertex  $z$   $O(\deg(z))$  times
  - Setting/getting a label takes  $O(1)$  time
- ◆ Priority queue operations
  - Each vertex is inserted once into and removed once from the priority queue, where each insertion or removal takes  $O(\log n)$  time
  - The key of a vertex in the priority queue is modified at most  $\deg(v)$  times, where each key change takes  $O(\log n)$  time
- ◆ Dijkstra's algorithm runs in  $O((n + m) \log n)$  time provided the graph is represented by the adjacency list structure
- ◆ The running time can also be expressed as  $O(m \log n)$  since the graph is connected. Why?

44

- ◆ If the graph is connected, then  $m \geq n - 1$
- ◆ Therefore,  $O(m + n)$  is ...

45

## Template Pattern Extension



- ◆ Using the template method pattern, we can extend Dijkstra's algorithm to return a tree of shortest paths from the start vertex to all other vertices
- ◆ We store with each vertex a third label:
  - parent edge in the shortest path tree
- ◆ In the edge relaxation step, we update the parent label

**Algorithm *DijkstraShortestPathsTree*( $G, s$ )**

```

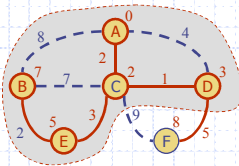
...
for all v ∈ G.vertices()
    ...
    setParent(v, ∅)
    ...
for all e ∈ G.incidentEdges(u)
    { relax edge e }
    z ← G.opposite(u, e)
    r ← getDistance(u) + weight(e)
    if r < getDistance(z)
        setDistance(z, r)
        setParent(z, e)
        Q.replaceKey(getLocator(z), r)
    
```

46

## Why Dijkstra's Algorithm Works



- ◆ Dijkstra's algorithm is based on the greedy method. It adds vertices by increasing distance.
- Suppose it didn't find all shortest distances. Let  $F$  be the first wrong vertex the algorithm processed.
- When the previous node,  $D$ , on the true shortest path was considered, its distance was correct.
- But the edge  $(D, F)$  was **relaxed** at that time!
- Thus, as long as  $d(F) \geq d(D)$ ,  $F$ 's distance cannot be wrong. That is, there is no wrong vertex distance.



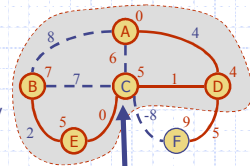
47

## Why It Doesn't Work for Negative-Weight Edges



- ◆ Dijkstra's algorithm is based on the greedy method. It adds vertices by increasing distance.

- If a node with a negative incident edge were to be added late to the cloud, it could mess up distances for vertices already in the cloud.



C's true distance is 1, but it is already in the cloud with  $d(C)=5$ !

48



## Bellman-Ford Algorithm (later)

Works even with negative-weight edges

Must assume directed edges

49

## Main Point

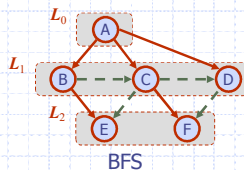
2. By using the adjacency list data structure to represent the graph and a priority queue enhanced with locators to store the vertices not yet in the tree, the shortest path algorithm achieves a running time  $O(m \log n)$ .  
The algorithms of nature are always most efficient for maximum growth and progress.

50

## BFS Levels

When actually implemented, the levels are normally merged into a single sequence/queue

How could we keep track of the level of a vertex? (HW)



51

## Connecting the Parts of Knowledge with the Wholeness of Knowledge

1. Finding the shortest path to some desired goal is a common application problem in systems represented by weighted graphs, such as airline or highway routes.
2. By systematically extending short paths using data structures **especially suited** to this process, the shortest path algorithm operates in time  $O(m \log n)$ .

52

3. **Transcendental Consciousness** is the silent field of infinite correlation where everything is eternally connected by the shortest path.
4. **Impulses within Transcendental Consciousness:** Because the natural laws within this unbounded field are infinitely correlated (no distance), they can govern all the activities of the universe simultaneously.
5. **Wholeness moving within itself:** In Unity Consciousness, the individual experiences the shortest path between one's Self and everything in the universe, a path of zero length.

53