Lecture 16: Shortest Paths in a Weighted Graph

Path of Least Action

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.

Science of Consciousness: Natural law always chooses the path of least action, the shortest path to the goal with no wasted effort.

Another Algorithm Design Method

Greedy Strategy

- Examples:
  - Fractional Knapsack Problem
  - Task Scheduling
  - Shortest Path
  - Minimum Spanning Tree

Requires the Greedy-Choice Property

Another Important Technique for Design of Efficient Algorithms

- Useful for effectively attacking many computational problems
- Greedy Algorithms
  - Apply to optimization problems
  - Key technique is to make each choice in a locally optimal manner
  - Many times provides an optimal solution much more quickly than does a dynamic-programming solution

The Greedy Method: Outline and Reading

- The Greedy Design Technique
- Fractional Knapsack Problem
- Today: Shortest Path
- Tomorrow: Minimum Spanning Trees

Greedy Algorithms

- **The feasible solution**: A subset of given inputs that satisfies all specified constraints of a problem is known as a “feasible solution”.
- **An optimal solution** is a feasible solution that results in the largest possible objective function value when maximizing (or smallest when minimizing).
- Used for optimizations
  - some quantity is to be minimized or maximized
- Always make the choice that looks best at each step
  - the hope is that these locally optimal choices will produce the globally optimal solution
- Works for many problems but NOT for others

The Greedy Design Technique

- A general algorithm design strategy,
- Built on the following elements:
  - **configurations**: represent the different choices (collections or values) that are possible at each step
  - **objective function**: a score is assigned to configurations (based on what we want to either maximize or minimize)
- Works when applied to problems with the **greedy-choice** property:
  - A globally-optimal solution can always be found by
    - Beginning from a starting configuration
    - Then making a series of local choices or improvements

Making Change

- Coin Denomination in US – Quarter (25 cents), Dime (10 cents), Nickel (5 cents) and Penny (1 cent).
- Objective: Find the minimum number of coins for a change
- Strategy: Choose the coin with the largest denomination that is less than or equal to the unaccounted portion of the change.
- For example, to find a change for 48. Coin set {25, 10, 5, 1 }
  - Choose 25. Remaining 48-25 = 23. Selected 1 Quarter(25)
  - Can choose 10 cents. 23-10 = 13. Selected 1 Dime(10)
  - Can choose 10 cents. 13-10 = 3. Selected 1 Dime(10)
  - Can choose 1 cent. 3-1=2. Selected 1 Penny
  - Can choose 1 cent. 2-1=1. Selected 1 Penny
  - Can choose 1 cent. 1-1=0. Selected 1 Penny.
  - We reached zero and stop.
  - Minimum number of coins for 48 change is 6 coins.
- we would choose 1 quarter, 2 dimes and 3 pennies.
- The optimal solution is thus 6 coins and there cannot be anything less than 6 coins for US coin denominations.

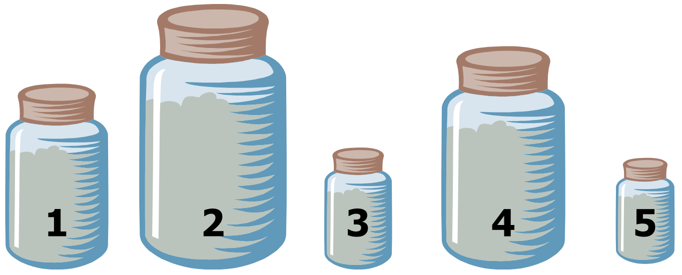
The Fractional Knapsack Problem

- Given: A set S of n items, with each item i having
  - b_i - a positive benefit/profit
  - w_i - a positive weight
- Goal: Choose items with maximum total benefit but with weight at most W.
- If we are allowed to take fractional amounts, then this is the **fractional knapsack problem**.
  - In this case, we let x_i denote the amount we take of item i
  - Objective: maximize $$\sum_{i \in S} b_i (x_i / w_i)$$
  - Constraint: $$\sum_{i \in S} x_i \leq W$$

Example

- Given: A set S of n items, with each item i having
  - b_i - a positive benefit
  - w_i - a positive weight
- Goal: Choose items with maximum total benefit but with weight at most W.

"knapsack W = 10"



Items:	1	2	3	4	5
Weight:	4 ml	8 ml	2 ml	6 ml	1 ml
Benefit:	\$12	\$32	\$40	\$30	\$50
Value: (\$ per ml $X_i = b_i/w_i$)	3	4	20	5	50

10 ml

Solution:
- 1 ml of 5
- 2 ml of 3
- 6 ml of 4
- 1 ml of 2

Maximum Profit = \$124

The Fractional Knapsack Algorithm

- Greedy choice:
- Keep taking item with highest **value** (benefit to weight ratio)
  - Since $\sum_{i \in S} b_i (x_i / w_i) = \sum_{i \in S} (b_i / w_i) x_i$
  - Run time: O(n log n). Why?

```
Algorithm fractionalKnapsack(S, W)
    Input: set S of items w/ benefit $b_i$
        and weight $w_i$; max. weight $W$
    Output: amount $x_i$ of each item $i$
        to maximize benefit with
        weight at most $W$
    for each item i in S do
        $x_i \leftarrow 0$
        $v_i \leftarrow b_i / w_i$            {value}
    $w \leftarrow 0$                       {total weight}
    while $w < W$ do
        remove item $i$ with highest $v_i$
        $x_i \leftarrow \min\{w_i, W - w\}$
        $w \leftarrow w + \min\{w_i, W - w\}$
```

The Fractional Knapsack Algorithm

Abstract Algorithm:

```
Algorithm fractionalKnapsack(S, W)
    Input: set S of items w/ benefit $b_i$
        and weight $w_i$; max. weight $W$
    Output: amount $x_i$ of each item $i$
        to maximize benefit with
        weight at most $W$
    for each item i in S do
        $x_i \leftarrow 0$
        $v_i \leftarrow b_i / w_i$            {value}
    $w \leftarrow 0$                    {total weight}
    while $w < W$ do
        remove item $i$ with highest $v_i$
        $x_i \leftarrow \min\{w_i , W - w\}$
        $w \leftarrow w + x$
```

Algorithm Details:

```
Algorithm fractionalKnapsack(S, W)
    $Q \leftarrow$ new Max Priority Queue
    $x \leftarrow$ new Array of size n
    for $i \leftarrow 0$ to S.size() - 1 do
        $(bn, wt) \leftarrow$ S.elemAtRank(i)
        $x[i] \leftarrow 0$
        $v \leftarrow bn / wt$    {benefit per unit}
        Q.insertItem(v, (bn, wt, i) )
    $w \leftarrow 0$                   {total weight}
    while $w < W$ do
        $(bn, wt, i) \leftarrow$ Q.removeMax()
        $x[i] \leftarrow \min\{wt , W - w\}$
        $w \leftarrow w + x[i]$
```

The 0/1 Knapsack Problem

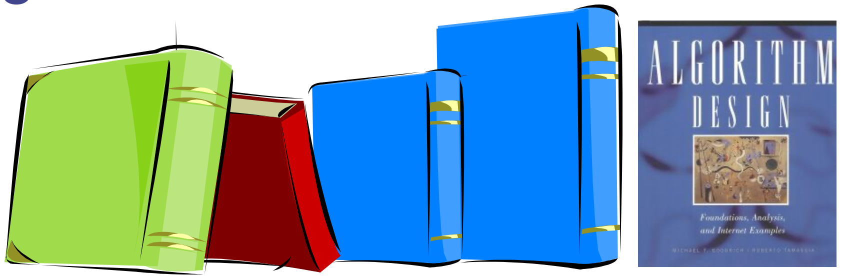
The 0/1 Knapsack Problem

- Given: A set S of n items, with each item i having
  - w_i - a positive weight
  - b_i - a positive benefit
- Goal: Choose items with maximum total benefit but with weight at most W.
- If we are **not** allowed to take fractional amounts, then this is the **0/1 knapsack problem**.
  - In this case, we let T denote the set of items we take
  - Objective: maximize $\sum_{i \in T} b_i$
  - Constraint: $\sum_{i \in T} w_i \leq W$

Example

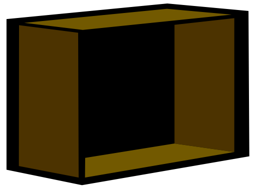
- Given: A set S of n items, with each item i having
  - b_i - a positive "benefit"
  - w_i - a positive "weight"
- Goal: Choose items with maximum total benefit but with weight at most W.

Items:



	1	2	3	4	5
Weight:	5 in	2 in	2 in	6 in	2 in
Benefit:	\$20	\$3	\$7	\$26	\$80
Xi:	1	0	1	0	1

"knapsack", W = 9

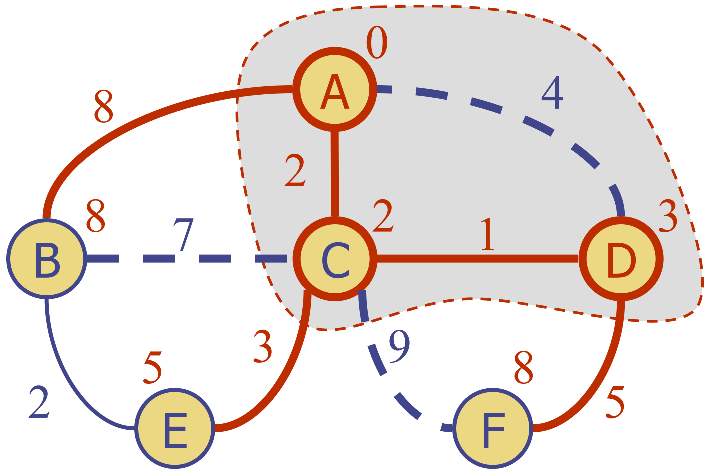


box of width 9 in

Solution:
- item 5 (\$80, 2 in)
- item 1 (\$20, 5in)
- item 3 (\$7, 2in)
- Total Max benefit = \$ 107

Shortest Paths

Path of Least Action

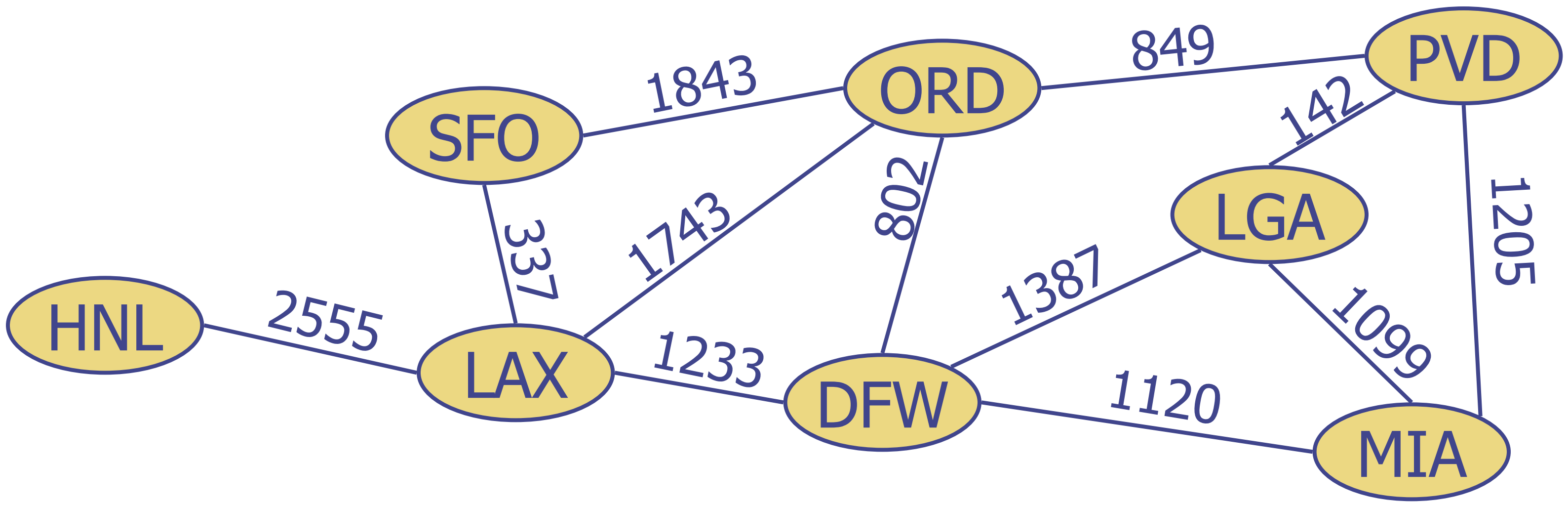


Outline and Reading

- Weighted graphs
  - Shortest path problem
  - Shortest path properties
- Dijkstra's algorithm is a well-known algorithm used to find the shortest paths from a single source vertex to all other vertices in a graph with non-negative edge weights.

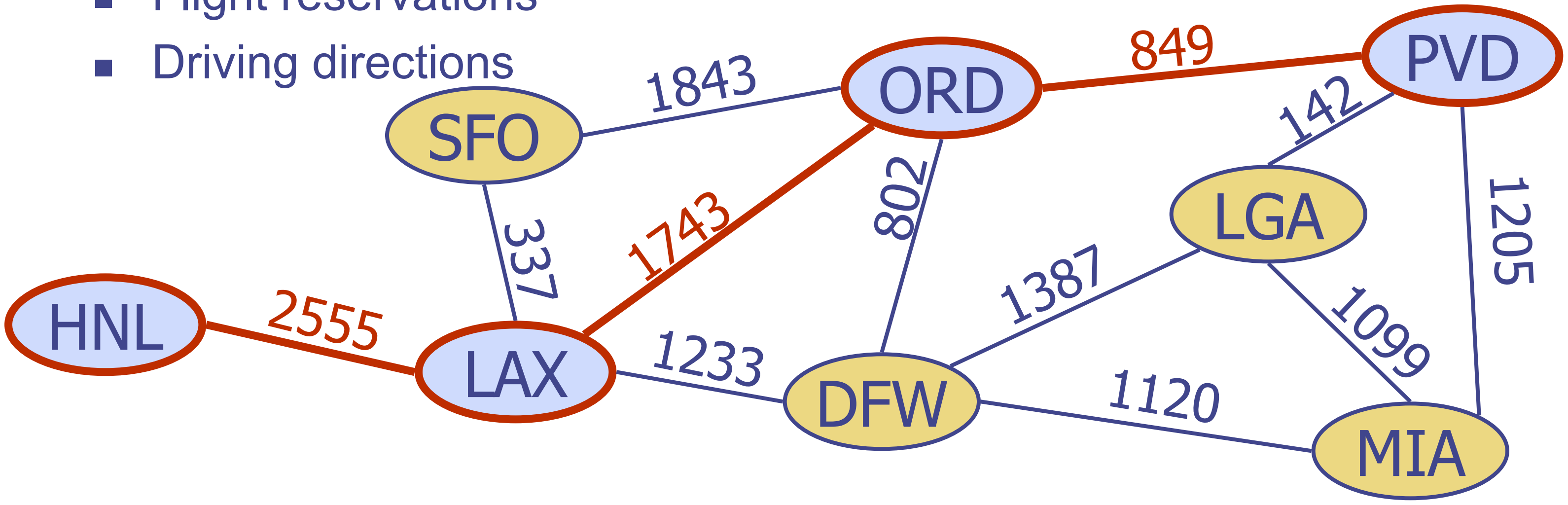
Weighted Graphs

- 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



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



Shortest Path Properties

Property 1:

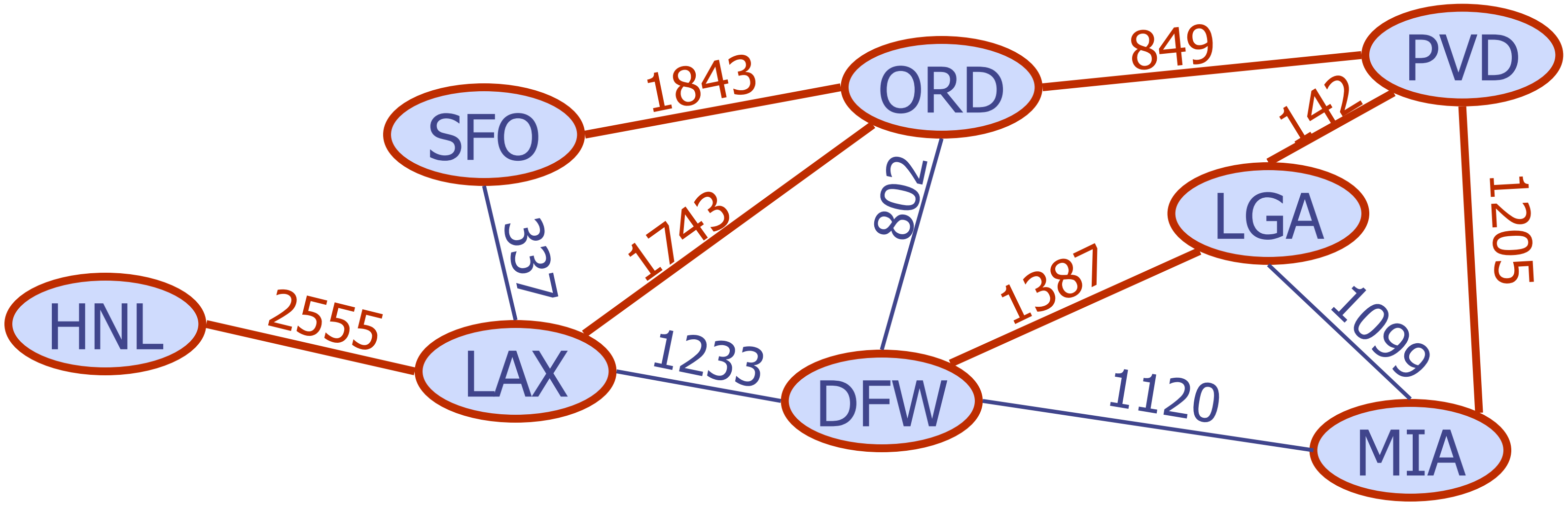
A subpath of a shortest path is itself a shortest path

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



Dijkstra’s Algorithm

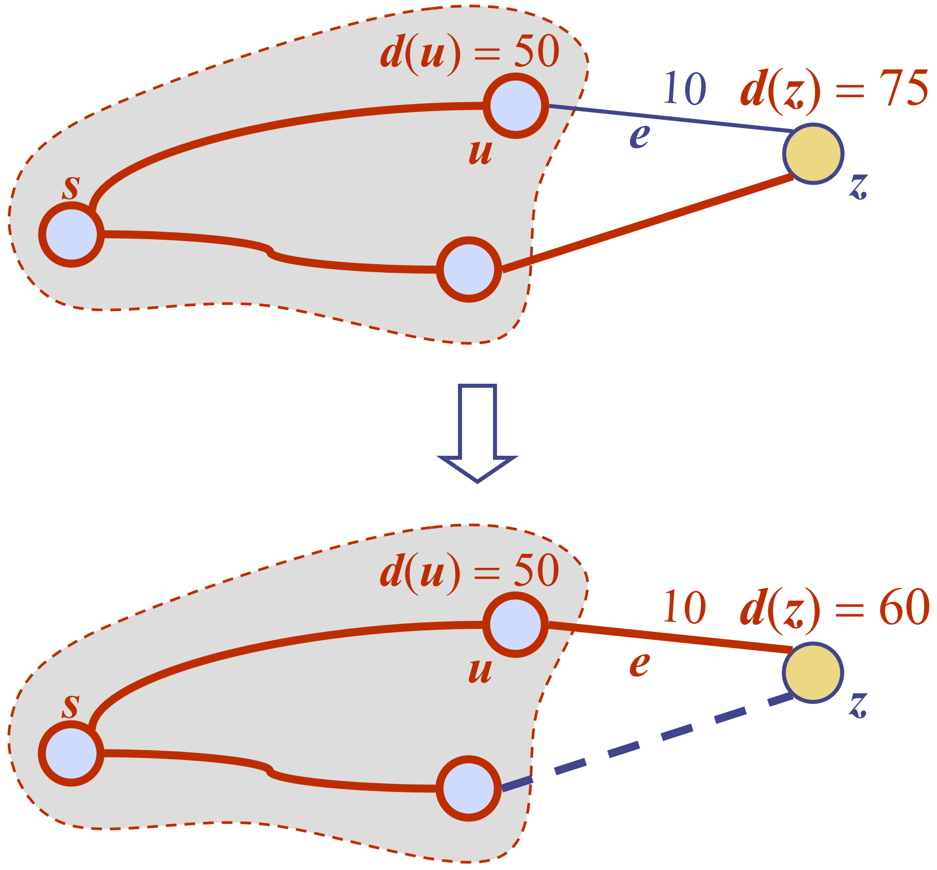
- The distance of a vertex v from a vertex s is the length of a shortest path between s and v
- Dijkstra’s algorithm computes the shortest distances of all the vertices from a given start vertex s
- Assumptions:
  - the graph is connected
  - the edges are undirected
  - the edge weights are **nonnegative**

Dijkstra's Algorithm (Informal)

- We grow a "**tree**" 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 tree and its adjacent vertices
- At each step
  - We add to the tree a vertex u
    - outside the tree
    - with the smallest distance label, $d(u)$
  - Then we update the labels of the vertices adjacent to u

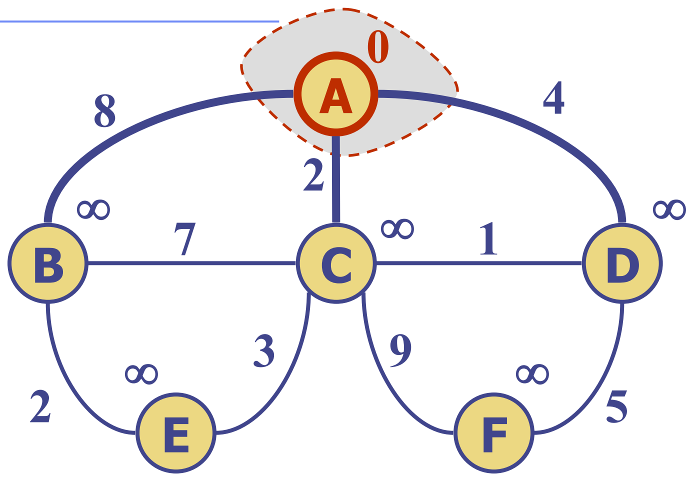
Edge Relaxation

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



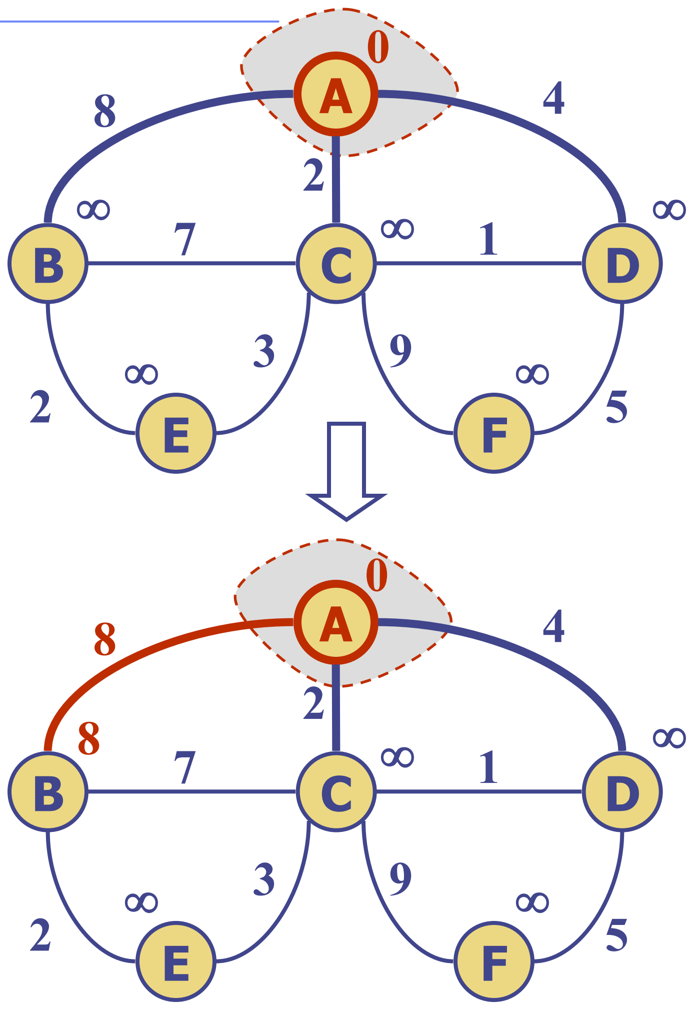
Example

$d(z) \leftarrow \min\{d(z), d(u)+weight(e)\}$



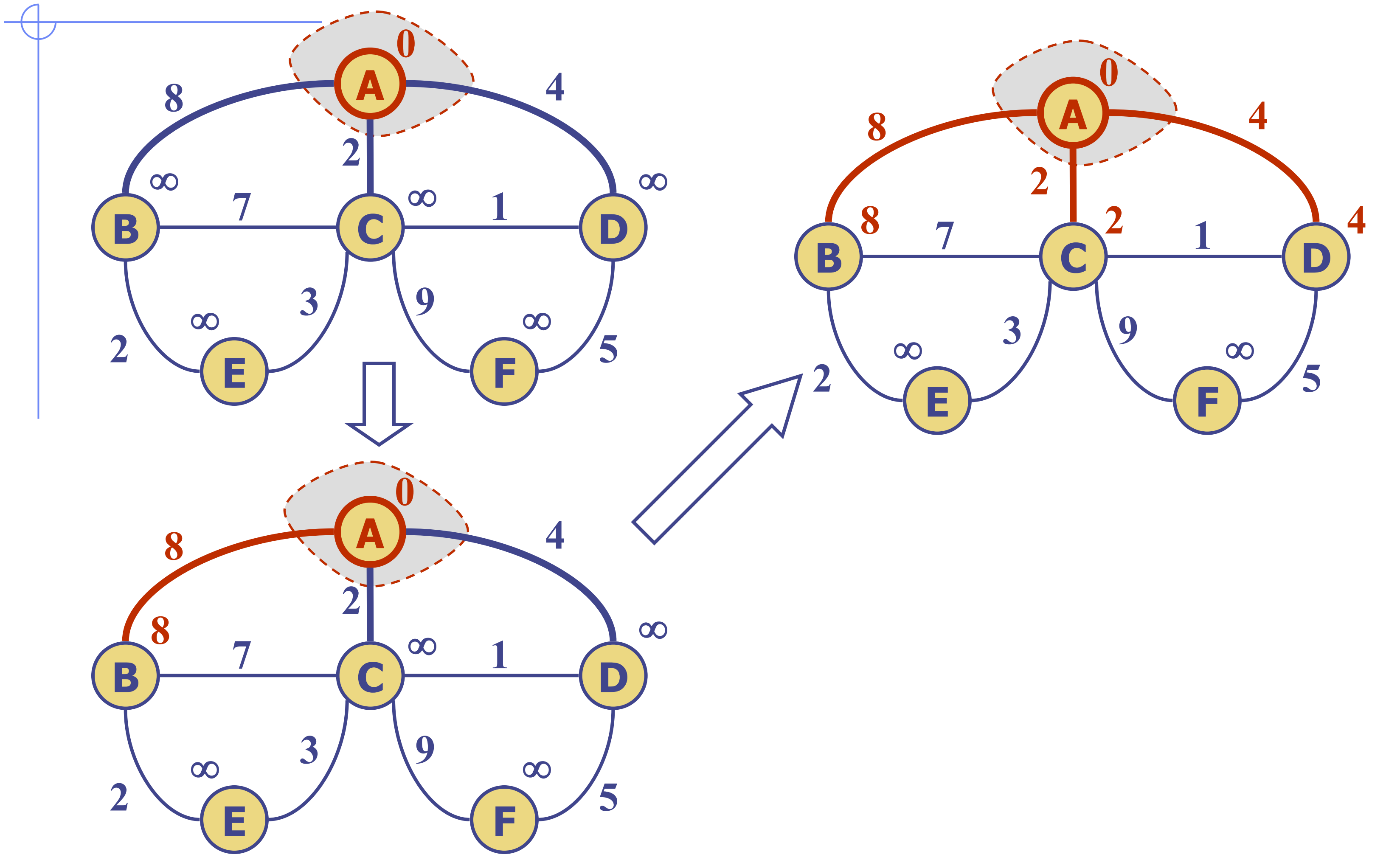
$$d(z) \leftarrow \min\{d(z),\ d(u)+weight(e)\}$$

Example



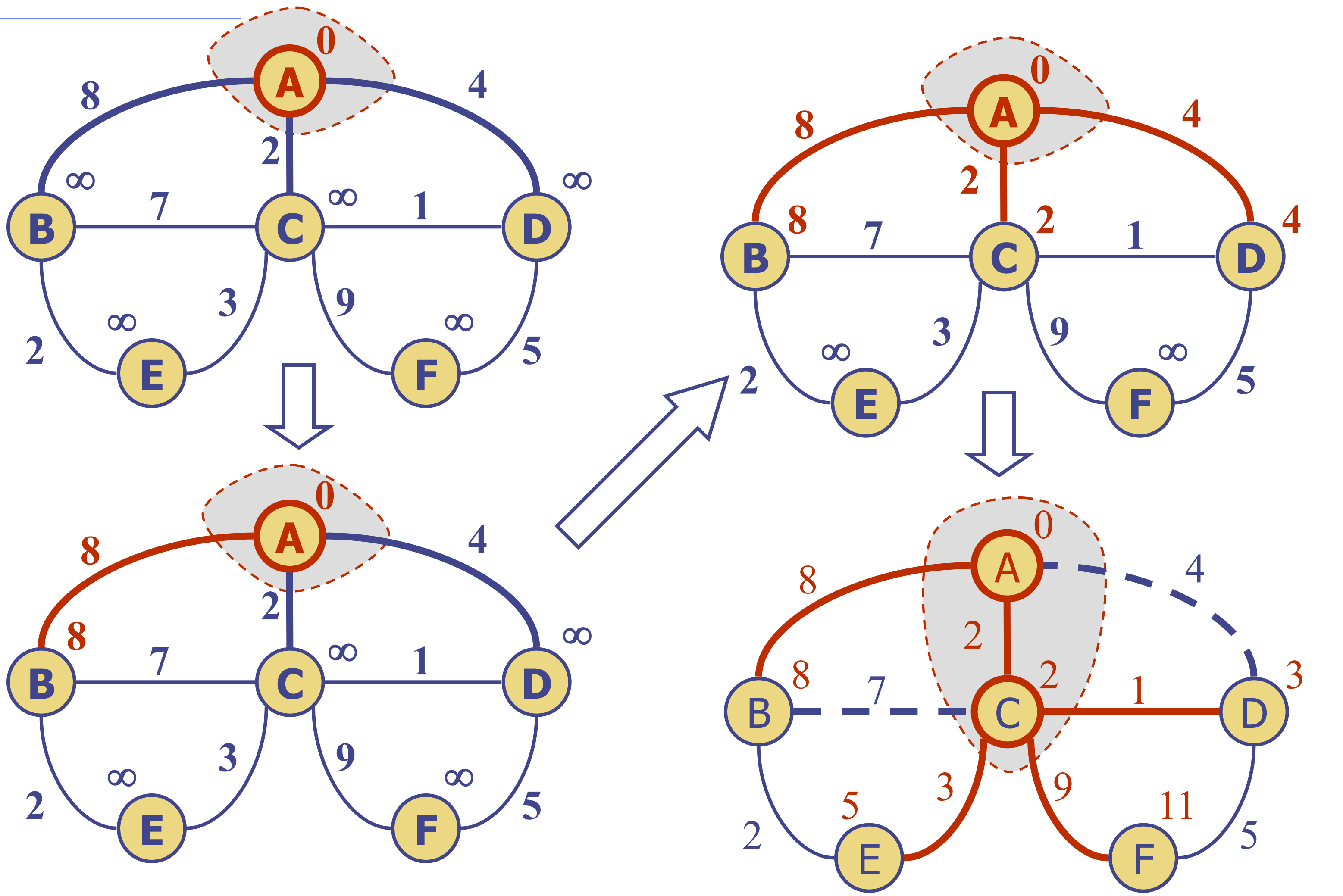
$$d(z) \leftarrow \min\{d(z),\ d(u)+weight(e)\}$$

Example



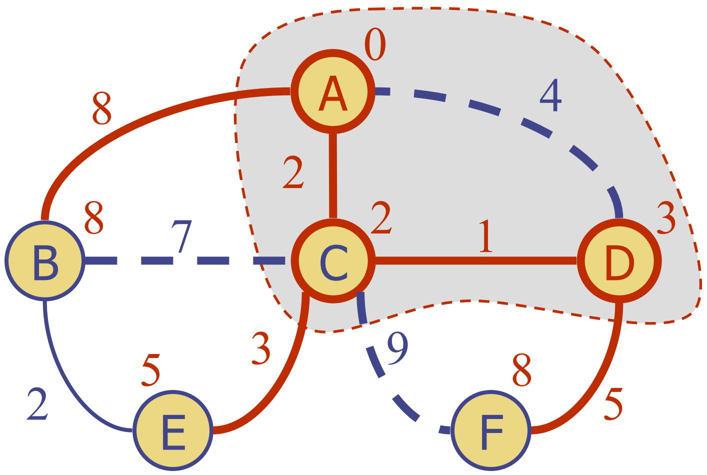
$$d(z) \leftarrow \min\{d(z),\ d(u)+weight(e)\}$$

Example

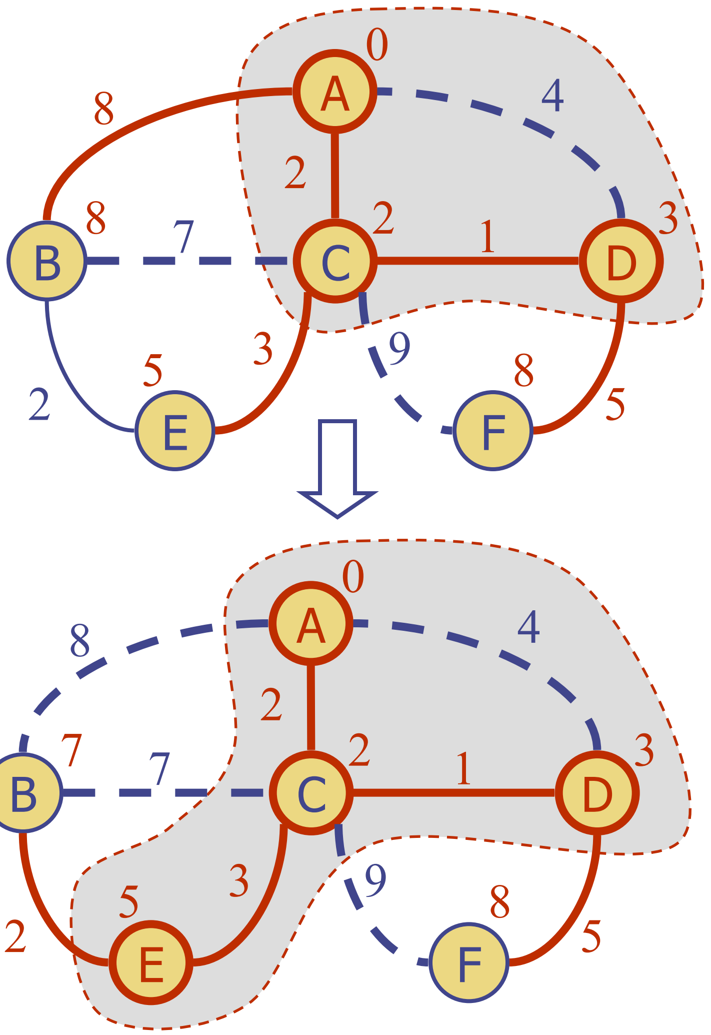


$$d(z) \leftarrow \min\{d(z), d(u)+weight(e)\}$$

Example

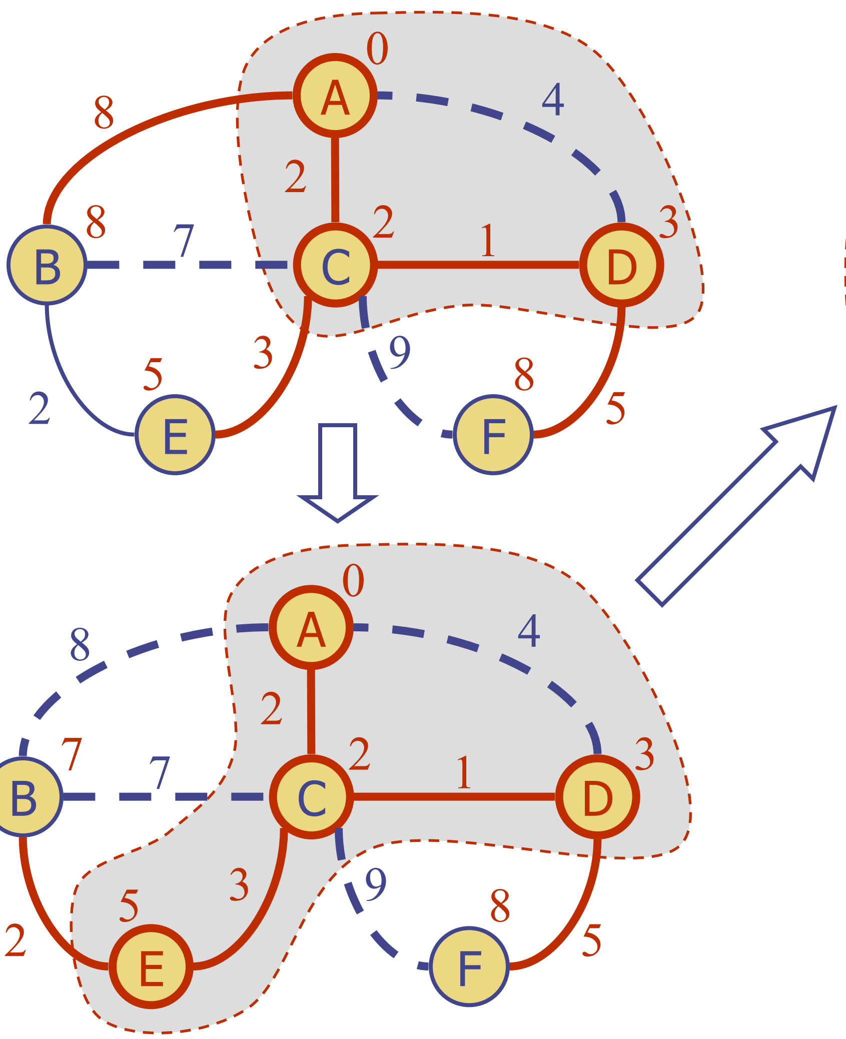


Example

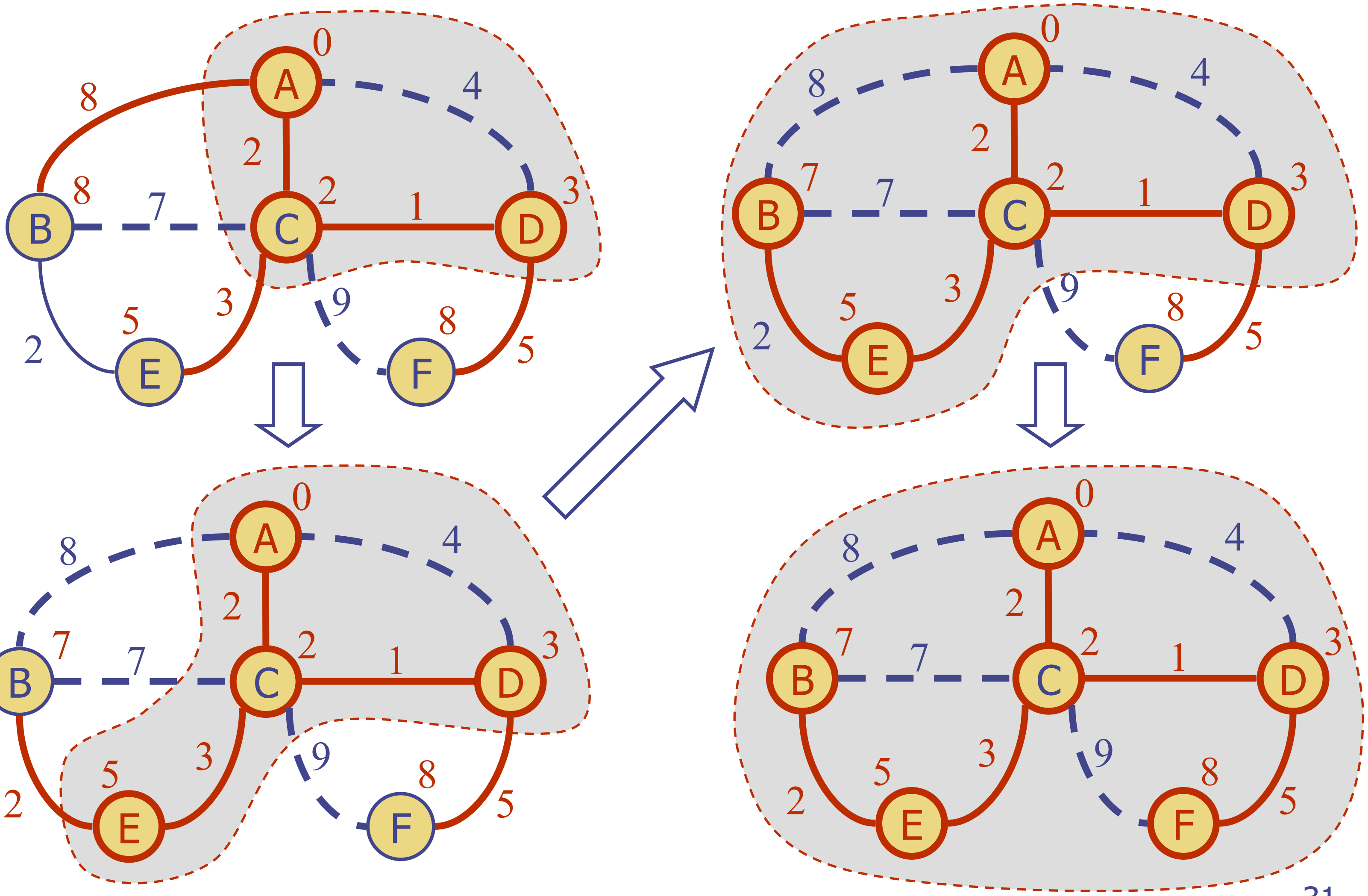


$$d(z) \leftarrow \min\{d(z), d(u)+weight(e)\}$$

Example



Example



Dijkstra’s Algorithm

- A Heap (or priority queue) stores the vertices outside the tree
  - 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.
- Opposite(u,e)-Given a vertex v and an edge e, this method returns the vertex on the opposite end of e. If e is connected to v

```
Algorithm DijkstraDistances(G, s)
    PQ ← new priority queue
    for all  v ∈ G.vertices() do
        if  v = s // Initial vertex
            setDistance(v, 0)
        else
            setDistance(v, ∞)
        PQ.insertItem(getDistance(v), v)
        setLocator(v, p)// Set node and position
    while ! PQ.isEmpty() do
        u ← PQ.removeMin() // get the min vertex
        for all  e ∈ G.incidentEdges(u) do
            { relax edge e }
            z ← G.opposite(u,e)
            d ← getDistance(u) + weight(e)
            if  d < getDistance(z) then
                setDistance(z,d)
                PQ.replaceKey(z,d)  {new
    method}
```

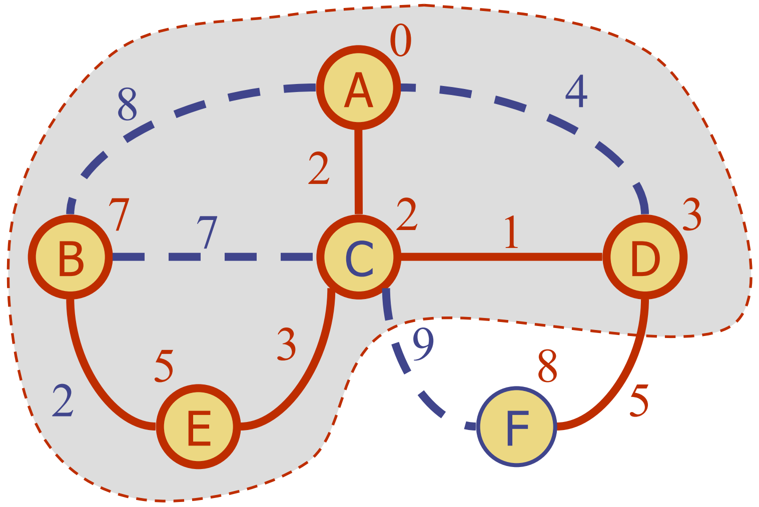
Analysis

- Graph operations
  - Method incidentEdges is called once for each vertex
  - Recall that $\Sigma_v \deg(v) = 2m$
- 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 $\deg(w)$ 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?

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

Why Dijkstra's Algorithm Works

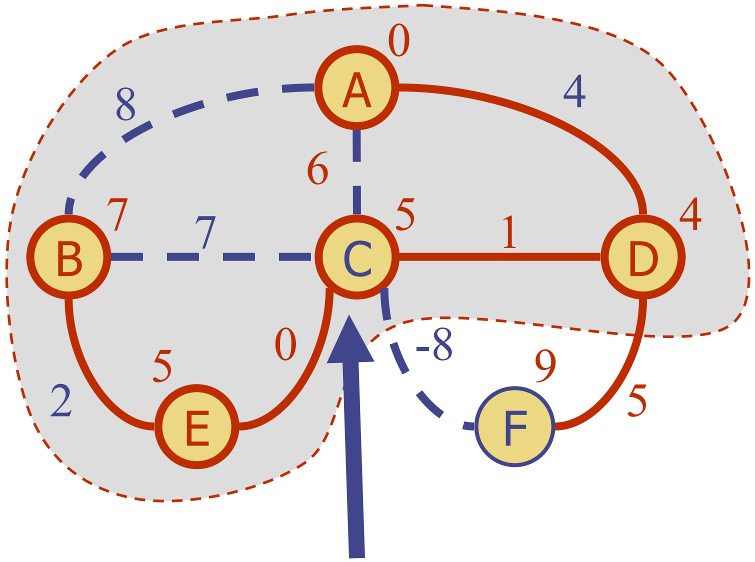
- Dijkstra's algorithm is based on the greedy strategy. It adds vertices by increasing distance.
  - Each vertex outside the tree has distance greater or equal to vertices in tree because there are no negative weight edges.
  - Suppose that when the last node, D, was considered and added to the tree, its minimum distance (3) was correct.
  - But the edge (D,F) was **relaxed** at that time!
  - Thus, as long as d(F)≥d(D), F's distance cannot be wrong. That is, there is no wrong vertex distance.



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.



Main Point

1. By using the adjacency list data structure to represent the graph and a heap or priority queue enhanced with positions to store the vertices not yet in the tree, the shortest path algorithm achieves a running time ***O***(***m*** log ***n***).

Science of Consciousness: The algorithms of nature in the unified field are always most efficient for maximum growth and progress (e.g., laws of physics, biochemistry, etc.). Contact with this field brings growth and progress to individual life as demonstrated by hundreds of scientific studies showing the benefits of the TM technique.

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 (heap based Priority Queue), the shortest path algorithm runs in time $\boldsymbol{O}(\boldsymbol{m}\log \boldsymbol{n})$.

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