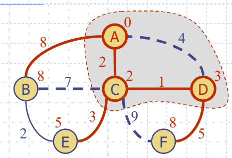
Lecture 14a: Shortest Paths in a Weighted Graph

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

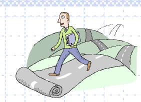
2

Outline and Reading

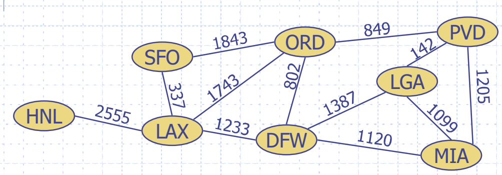
- Weighted graphs (§7.1)
  - Shortest path problem
  - Shortest path properties
- Dijkstra's algorithm (§7.1.1)
  - Algorithm
  - Edge relaxation

3

Weighted Graphs

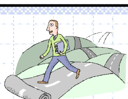


- In a weighted graph, each edge has an associated numerical value, called the weight of the edge
- Edge weights may represent, distances, costs, etc.
- Example:
  - In a flight route graph, the weight of an edge represents the distance in miles between the endpoint airports

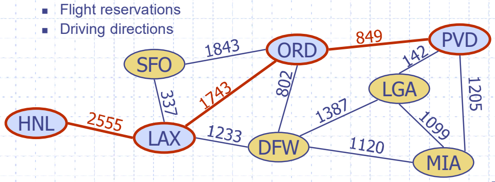


4

Shortest Path Problem

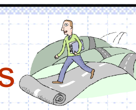


- Given a weighted graph and two vertices *u* and *v*, we want to find a path of minimum total weight between *u* and *v*.
  - Length of a path is the sum of the weights of its edges.
- Example:
  - Shortest path between Providence and Honolulu
- Applications
  - Internet packet routing
  - Flight reservations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



5

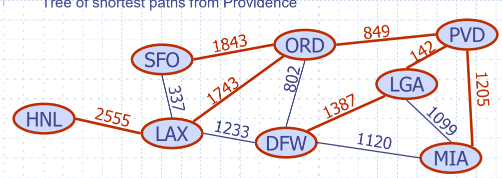
Shortest Path Properties



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



6

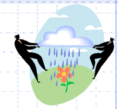
Dijkstra's Algorithm

- 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**

7

7

Dijkstra's Algorithm (Informal)



- We grow a "**cloud**" of vertices, beginning with s and eventually covering all the vertices
- We could also grow a tree of shortest paths from s to all other vertices in the graph (we can do this with a small change to our algorithm)
- 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, $d(u)$
    - $d(u)$ is the shortest distance s from u we will explain why we can't do better
  - Then we update the labels of the vertices adjacent to u

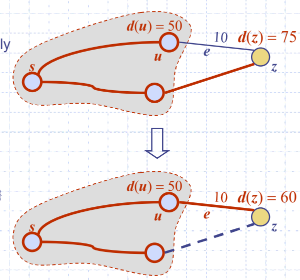
8

8

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:

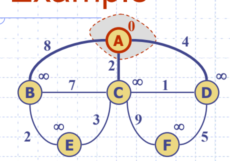
$d(z) \leftarrow \min\{d(z),\ d(u)+weight(e)\}$



9

9

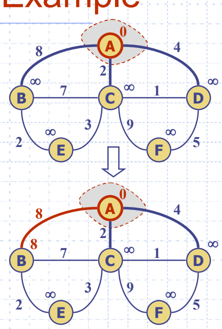
Example



10

10

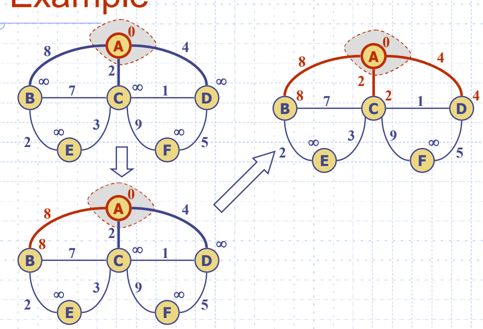
Example



11

11

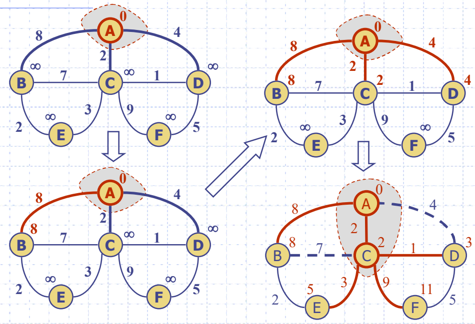
Example



12

12

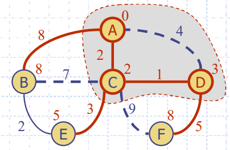
Example



13

13

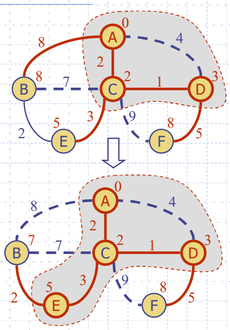
Example



14

14

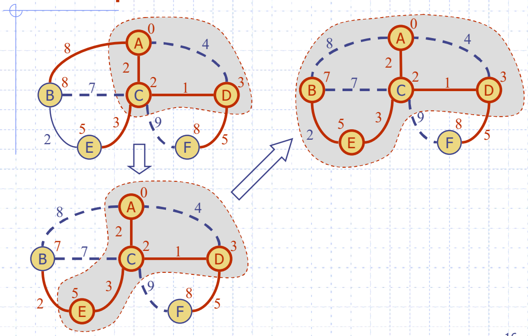
Example



15

15

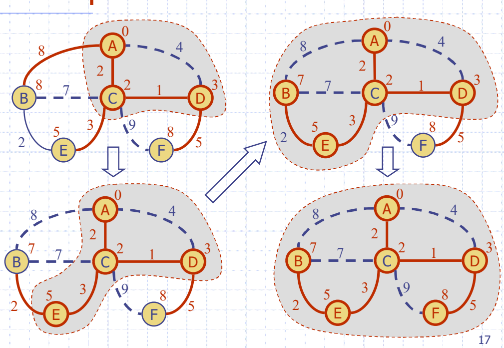
Example



16

16

Example



17

17

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?
  - Right now the last line runs in O(m n) time

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

18

18

Positions Revisited

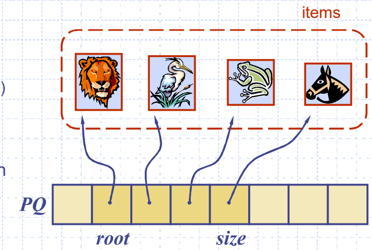
- Position
  - represents a "place" in a data structure
  - related to other positions in the data structure (e.g., previous/next or parent/child)
  - implemented as a node (in a Tree or List) or an array cell (in a Sequence)
- Position-based ADTs are fundamental data storage schemes
  - (e.g., tree and graph)
- In key-based ADTs a Position can be augmented to include both the key and element (i.e., an item)
  - (e.g., priority queue or dictionary)
- Position with locator
  - identifies and tracks a (key, element) item
  - has methods p.key() and p.element()
  - implemented as an object storing the key, element, and its location in the underlying structure (e.g., index into an array in a Heap or reference to a node in a tree)

19

19

Array-based Implementation

- We could use an array storing items in our Priority Queue
- An item object stores:
  - Item (key,elem)
- How can we implement a Position that can track the item location in the Priority Queue?



Priority Queue 20

20

How can we implement the position locators?

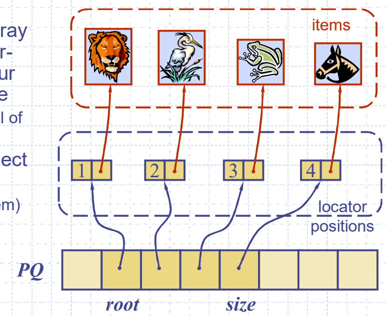
- "*We can solve any problem by introducing an extra level of indirection (abstraction).*" (David Wheeler; British Computer Scientist)
- Dynamic binding is implemented with another level of indirection
- Same technique for these locator positions

21

21

Array-based Implementation

- We use an array storing locator-positions in our Priority Queue
  - Another level of indirection
- A position object stores:
  - Item (key,elem)
  - Index



Sequences 22

22

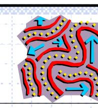
Locator-based Methods added to the Priority Queue

- Locator-Position 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 associated with locator l
  - replaceKey(l, k): replaces with k 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)

23

23

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 position
  - *replaceKey*(*l*,*k*) changes the key of the item in locator position l
- We store two labels with each vertex:
  - Distance (d(v) label)
  - Locator position in the priority queue

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

24

24

Dijkstra's Algorithm with Locators

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

```
Algorithm DijkstraDistances(G, s)
  D ← new Dictionary(BST)
  for all v ∈ G.vertices() do
    if v = s then
      setDistance(v, 0)
    else
      setDistance(v, ∞)
    p ← D.insert(getDistance(v), v)
    setLocator(v, p)
  while ¬D.isEmpty() do
    d ← D.minKey()
    u ← D.removeElement(d)
    for all e ∈ G.incidentEdges(u) do
      z ← G.opposite(u,e) { relax edge e }
      r ← getDistance(u) + weight(e)
      p ← getLocator (z)
      if r < getDistance(z) then
        setDistance(z,r)
        D.replaceKey(p,r)
```

25

25

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?

26

26

- If the graph is connected, then m $\geq$ n-1
- Therefore, O(m + n) is …
  - O(m) because m $\geq$ n-1 and we discard the low-order term n

27

27

Shortest Paths instead of Distances

- We can extend Dijkstra's algorithm to return a tree of shortest paths from the start vertex to all other vertices
- To do this, we store with each vertex a third attribute labeled parent:
  - the parent edges form a shortest path tree
  - the added code is shown in red
- In the edge relaxation step, we update the parent label

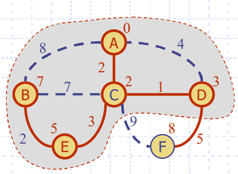
```
Algorithm DijkstraShortestPathsTree(G, s)
  Q ← new heap-based priority queue
  for all v ∈ G.vertices()
    if v = s then
      setDistance(v, 0)
    else
      setDistance(v, ∞)
    l ← Q.insert(getDistance(v), v)
    setLocator(v, l)
    setParent(v, ∅)
  while ¬Q.isEmpty() do
    u ← Q.removeMin()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

28

28

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.

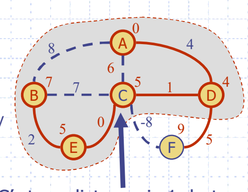


29

29

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!

30

30

Bellman-Ford Algorithm (later)

Works even with negative-weight edges

Must assume directed edges

31

31

Main Point

2. By using the adjacency list data structure to represent the graph and a priority queue enhanced with locator positions 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.

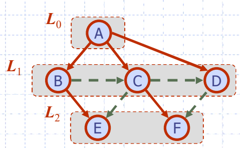
32

32

Exercise: BFS Levels

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

How could we keep track of the level of a vertex?



BFS

33

33

BFS Algorithm

- The BFS algorithm using a single sequence/list/queue L

```
Algorithm BFS(G) {top level}
    Input graph G
    Output labeling of the edges
        and partition of the
        vertices of G
  for all u ∈ G.vertices()
    setLabel(u, UNEXPLORED)
  for all e ∈ G.edges()
    setLabel(e, UNEXPLORED)
  for all v ∈ G.vertices()
    if isNextComponent(G, v)
      BFScomponent(G, v)

Algorithm isNextComponent(G, v)
  return getLabel(v) = UNEXPLORED
```

```
Algorithm BFScomponent(G, s)
  setLabel(s, VISITED)
  L ← new empty List
  L.insertLast(s)
  while ¬L.isEmpty() do
      v ← L.remove(L.first())
      for all e ∈ G.incidentEdges(v) do
        if getLabel(e) = UNEXPLORED
          w ← opposite(v,e)
          if getLabel(w) = UNEXPLORED
            setLabel(e, DISCOVERY)
            setLabel(w, VISITED)
            L.insertLast(w)
          else
            setLabel(e, CROSS)
```

34

34

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 $O(m \log n)$.

35

35

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.

36

36