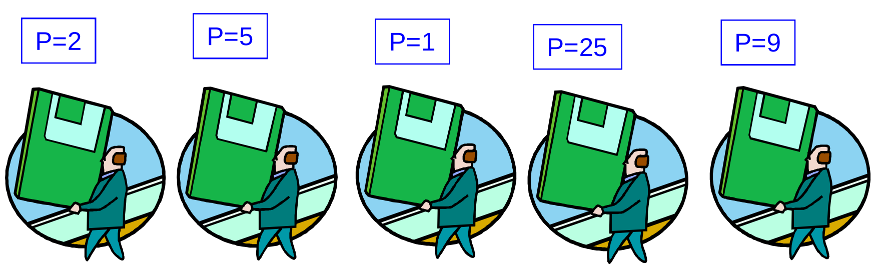
Bonus Topic:

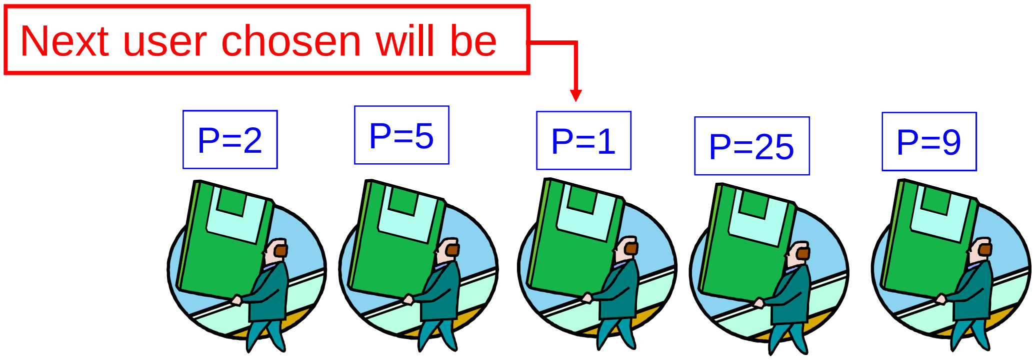
Priority Queue

The Priority Queue



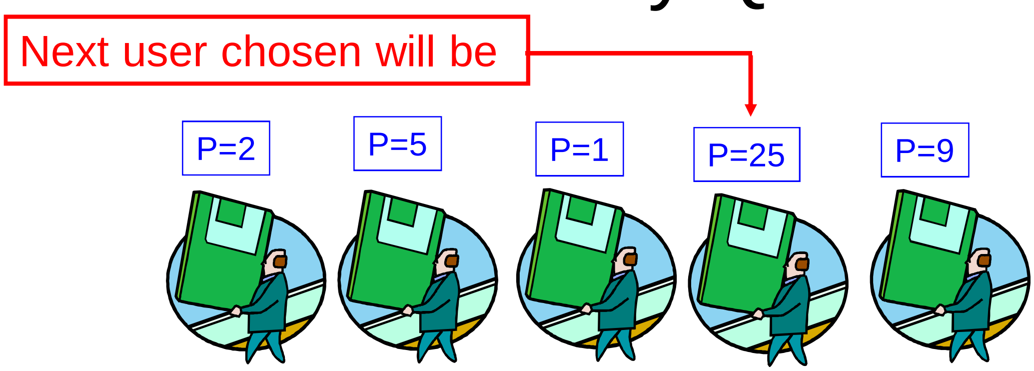
- We call it a priority *queue* - but its not FIFO
- Items in queue have PRIORITY
- Elements are removed from priority queue in either increasing or decreasing priority
  - Min Priority Queue
  - Max Priority Queue

The Priority Queue



- Consider situation where we have a computer whose services we are selling
- Users need different amounts of time
- Maximise earnings by min priority queue of users
  - i.e. when machine becomes free, the user who needs least time gets the machine; the average delay is minimised

The Priority Queue



- Consider situation where users are willing to pay more to secure access - they are in effect bidding against each other
- Maximise earnings by max priority queue of users
  - i.e. when machine becomes free, the user who is willing to pay most gets the machine

The Priority Queue

- Common data structure in computer science
- Responsible for scheduling jobs
  - Unix (linux) can allocate processes a priority
  - Time allocated to process is based on priority of job
- Priority of jobs in print scheduler

Priority Queue

Priority Queue

- The elements in a stack or a FIFO queue are ordered based on the sequence in which they have been inserted.
- In a priority queue, the sequence in which elements are removed is based on the priority of the elements.

Ordered Priority Queue

A Priority=1	B Priority=2	C Priority=3	D Priority=3

(highest priority) (lowest priority)

The first element to be removed.

Unordered Priority Queue

B Priority=2	C Priority=3	A Priority=1	D Priority=3

Priority Queue

Priority Queue - Array Implementation

- To implement a priority queue using an array such that the elements are **ordered** based on the priority.

Time complexity of the operations :

(assume the sorting order is from highest priority to lowest)

Insertion: Find the location of insertion. O(__)

Shift the elements after the location O(__)

where n = number of elements in the queue

Insert the element to the found location O(__)

Altogether: O(__)

The efficiency of insertion is important

Deletion: The highest priority element is at the front, ie. Remove the front element (Shift the remaining) takes O(__) time

Priority Queue

Priority Queue - Array Implementation

- To implement a priority queue using an array such that elements are **unordered**.

Time complexity of the operations :

Insertion: Insert the element at the rear position. *O(1)*

Deletion: Find the highest priority element to be removed. *O(n)*

Copy the value of the element to return it later. *O(1)*

Shift the following elements so as to fill the hole. *O(n)*

or replace the hole with the rear element *O(1)*

Altogether: *O(n)*

The efficiency of deletion is important

- Consider that, on the average,

Ordered Priority Queue: since it is sorted, every insertion needs to search half the array for the insertion position, and half elements are to be shifted.

Unordered Priority Queue: every deletion needs to search all n elements to find the highest priority element to delete.

Priority Queue

Priority Queue - List Implementation

- To implement a priority queue as an **ordered** list.

  Time complexity of the operations :
  (assume the sorting order is from highest priority to lowest)

  Insertion: Find the location of insertion. *O(n)*
  No need to shift elements after the location.
  Link the element at the found location. *O(1)*
  Altogether: *O(n)*

  Deletion: The highest priority element is at the front.
  ie. Remove the front element takes *O(1)* time

The efficiency of insertion is important.

More efficient than array implementation.

Priority Queue

Priority Queue - List Implementation

- To implement a priority queue as an **unordered** list.

  Time complexity of the operations :

  Insertion: Simply insert the item at the rear. *O(1)*

  Deletion: Traverse the entire list to find the maximum priority element. *O(n)*.
  Copy the value of the element to return it later. *O(1)*
  No need to shift any element.
  Delete the node. *O(1)*
  Altogether: *O(n)*

The efficiency of deletion is important

- Ordered list vs Unordered list
  <Comparison is similar to array implementations.>

Implementation Options

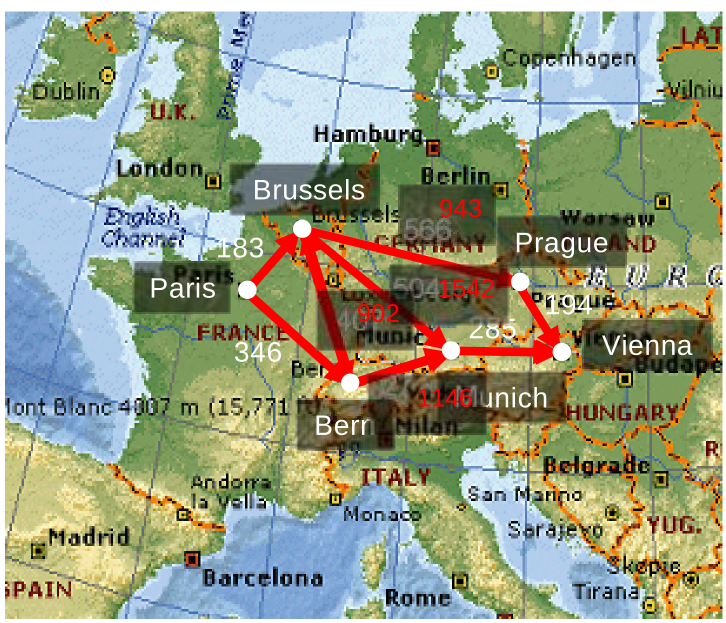
- Priority queue can be regarded as a heap
  - `isEmpty`, `size`, and `get` => O(1) time
  - `put` and `remove` => O(log n) time

    where n is the size of the priority queue

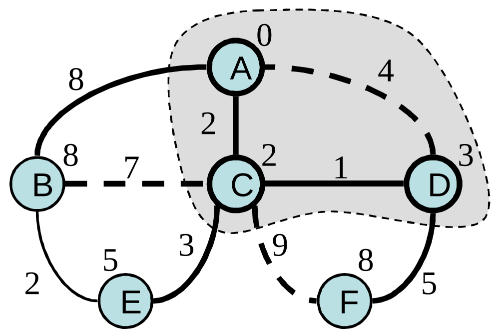
i.e. this is better than linear list option on average

- HEAP
  - A complete binary tree with values at its nodes arranged in a particular way (the priority!)

Shortest Paths Problem

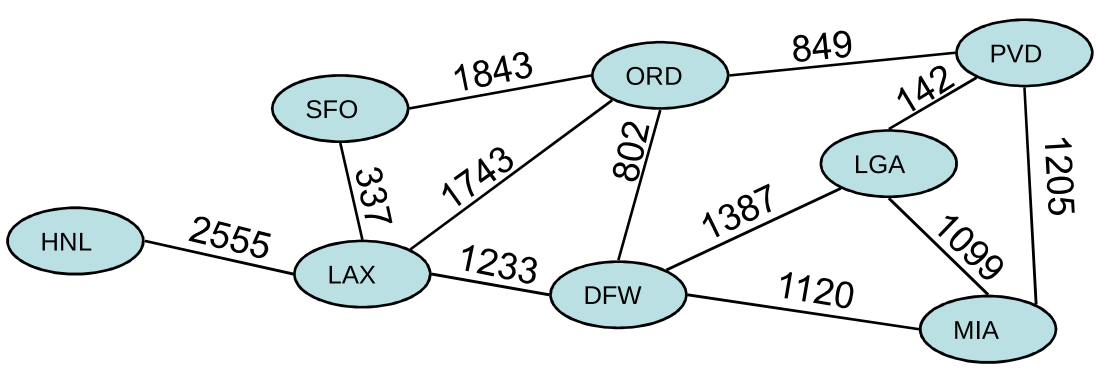


Shortest Paths



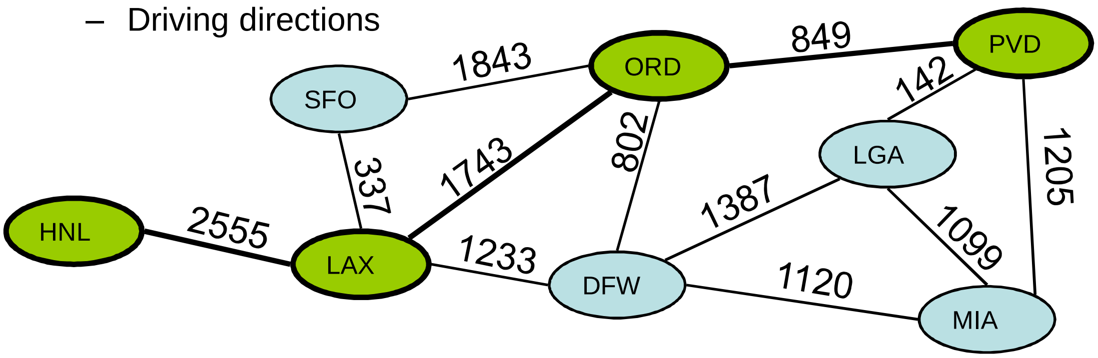
Weighted Graphs

- 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



Definition of Shortest Path

- Generalize distance to weighted setting
- Digraph $G = (V,E)$ with weight function $W: E \rightarrow R$ (assigning real values to edges)
- Weight of path $p = v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k$ is

$$w(p) = \sum_{i=1}^{k-1} w(v_i, v_{i+1})$$

- Shortest path = a path of the minimum weight
- Applications
  - static/dynamic network routing
  - robot motion planning
  - map/route generation in traffic

Shortest Path Properties

Property 1:

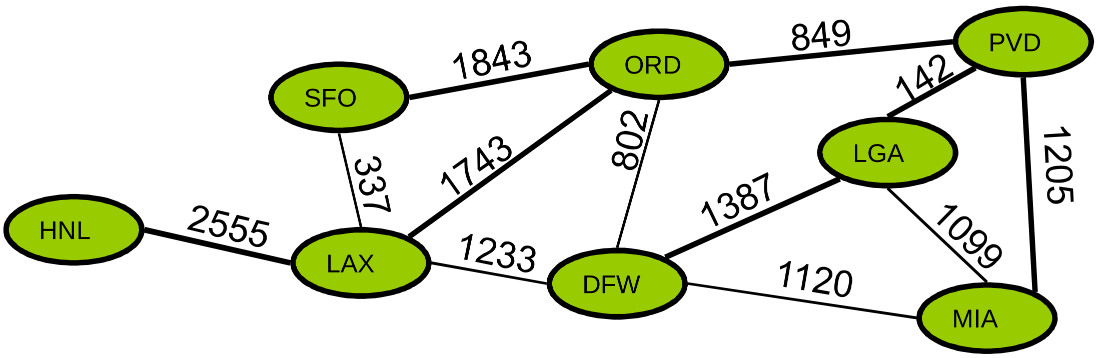
A subpath of a shortest path is itself a shortest path

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



Types of Shortest Path Problems

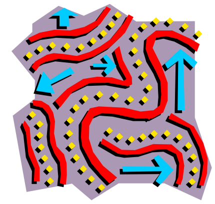
- Shortest-Path problems
  - **Single-source (single-destination).** Find a shortest path from a given source to each of the vertices
  - **Single-pair.** Given two vertices, find a shortest path between them. Solution to single-source problem solves this problem efficiently, too.
  - **All-pairs.** Find shortest-paths for every pair of vertices. Dynamic programming algorithm.
  - Unweighted shortest-paths – BFS.

Single-Source Shortest Paths

- The *single-source shortest paths* problem is to find the shortest paths from a vertex $v \in V$ to all other vertices in V of a weighted graph.
- Today, we will discuss the Dijkstra's serial algorithm, which is very similar to Prim's algorithm.
- This approach maintains a set of known shortest paths and adds to this set greedily to include other vertices in the graph.

Dijkstra's Shortest Path Algorithm

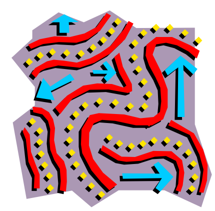
Single-Source Shortest Paths



- We wish to find the shortest route between Binghamton and NYC. Given a NYS road map with all the possible routes how can we determine our shortest route?

- We could try to enumerate all possible routes. It is certainly easy to see we do not need to consider a route that goes through Buffalo.

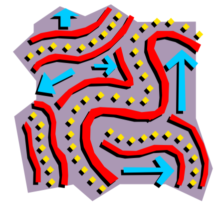
Modeling the "SSSP" Problem



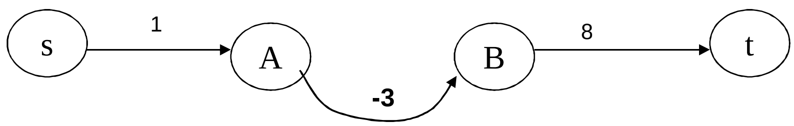
- **We can model this problem with a directed graph. Intersections correspond to vertices, roads between intersections correspond to edges and distance corresponds to weights. One way roads correspond to the direction of the edge.**

- **The problem:**
  - **Given a weighted digraph and a vertex *s* in the graph*:* find a shortest path from *s* to t**

The distance of a shortest path

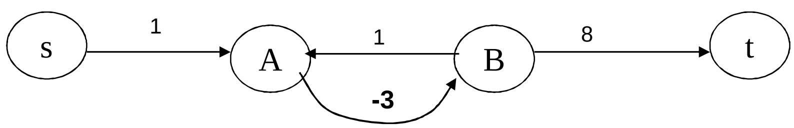


Case 1: The graph may have negative edges but no negative cycles. The shortest distance from s to t can be computed.



d(s,t)=6

Case 2: The graph contains negative weight cycles, and a path from s to t includes an edge on a negative weight cycle. The shortest path distance is -¥.



d(s,t)=- ¥

Dijkstra's Algorithm

- Non-negative edge weights
- Greedy, similar to Prim's algorithm for MST
- Like breadth-first search (if all weights = 1, one can simply use BFS)
- Use *Q*, priority queue keyed by *d*[*v*] (BFS used FIFO queue, here we use a PQ, which is re-ordered whenever *d* decreases)
- Basic idea
  - – maintain a set *S* of solved vertices
  - – at each step select "closest" vertex *u*, add it to *S*, and relax all edges from *u*

Dijkstra’s Algorithm

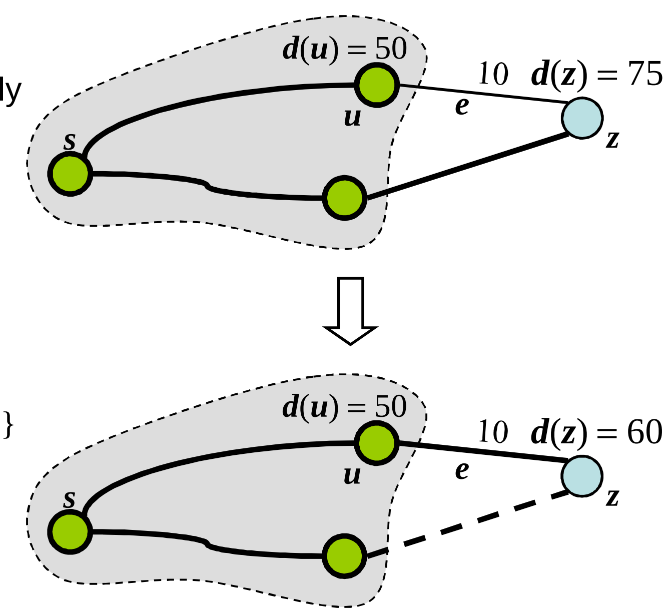
- The distance of a vertex $\boldsymbol{v}$ from a vertex $\boldsymbol{s}$ is the length of a shortest path between $\boldsymbol{s}$ and $\boldsymbol{v}$
- Dijkstra’s algorithm computes the distances from a given start vertex $\boldsymbol{s}$ to all the other vertices
- Assumptions:
  - the graph is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

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

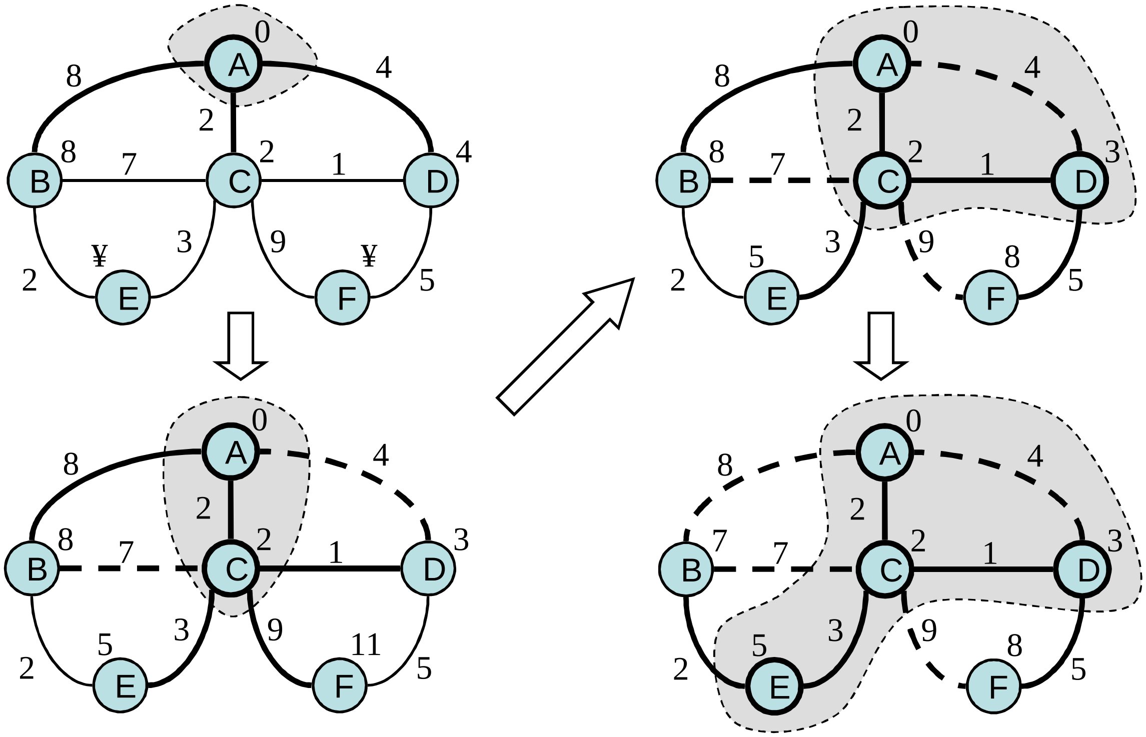
Edge Relaxation

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

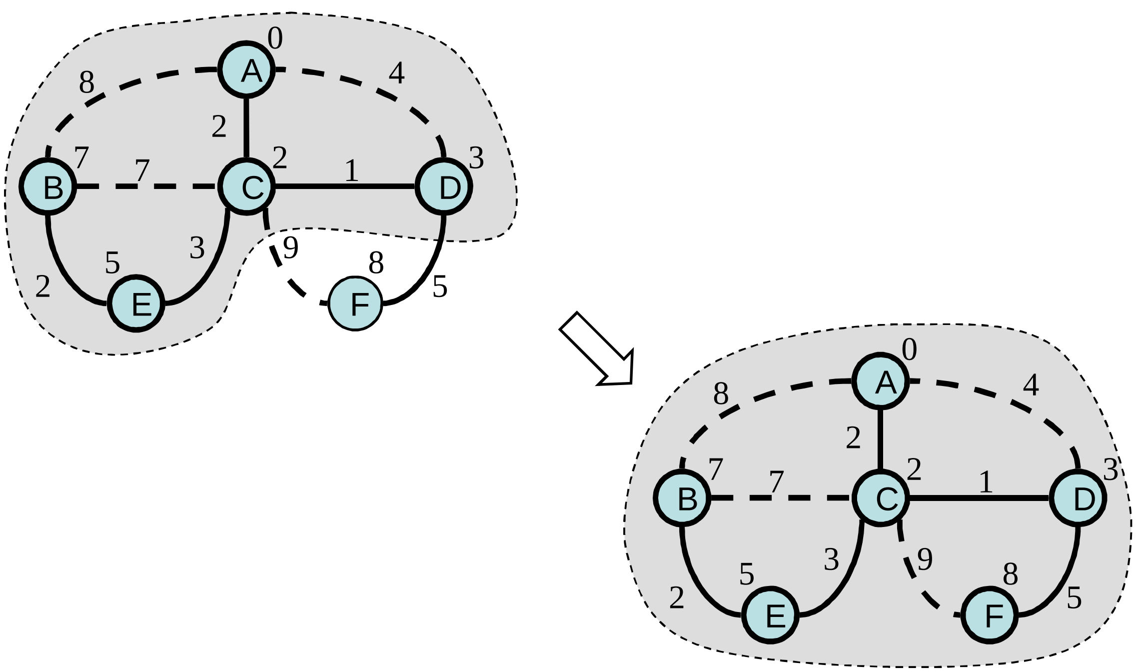
$$\boldsymbol{d}(\boldsymbol{z}) \leftarrow \min\{\boldsymbol{d}(\boldsymbol{z}),\boldsymbol{d}(\boldsymbol{u}) + \boldsymbol{weight}(\boldsymbol{e})\}$$



Example



Example (cont.)



Dijkstra's Pseudo Code

- Graph *G*, weight function *w*, root *s*

```
DIJKSTRA(G, w, s)
for each v ∈ V
        do d[v] ← ∞
d[s] ← 0
S ← ∅    ▷ Set of discovered nodes
Q ← V
while Q ≠ ∅
        do u ← EXTRACT-MIN(Q)
            S ← S ∪ {u}
            for each v ∈ Adj[u]
                  do if d[v] > d[u] + w(u, v)
                        then d[v] ← d[u] + w(u, v)
```

relaxing edges

Example

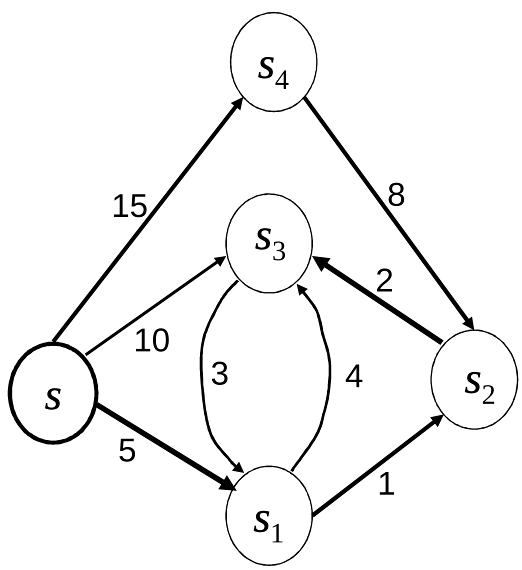
d(s, s) =0 $\leq$

d(s, s_1)=5 $\leq$

d(s, s_2)=6 $\leq$

d(s, s_3)=8 $\leq$

d(s, s_4)=15



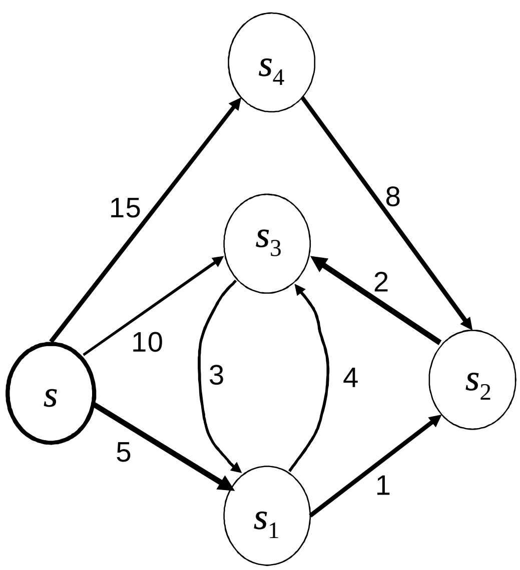
Note: The shortest path from *s* to s_2 includes s_1 as an intermediate node but cannot include s_3 or s_4.

Dijkstra's greedy selection rule

- Assume s_1, s_2 … s_{i-1} have been selected, and their shortest distances have been stored in Solution
- Select node s_i and save d(s, s_i) if s_i has the shortest distance from s on a path that may include only s_1, s_2 … s_{i-1} as intermediate nodes. We call such paths *special*
- To apply this selection rule efficiently, we need to maintain for each unselected node v the *distance of the shortest special path* from s to v, $D[v]$.

Application Example

Solution = {(s, 0)}
D[s_1]=5 for path [s, s_1]
D[s_2]= ∞ for path [s, s_2]
D[s_3]=10 for path [s, s_3]
D[s_4]=15 for path [s, s_4].



Solution = {(s, 0), (s_1, 5) }
D[s_2]= 6 for path [s, s_1, s_2]
D[s_3]=9 for path [s, s_1, s_3]
D[s_4]=15 for path [s, s_4]

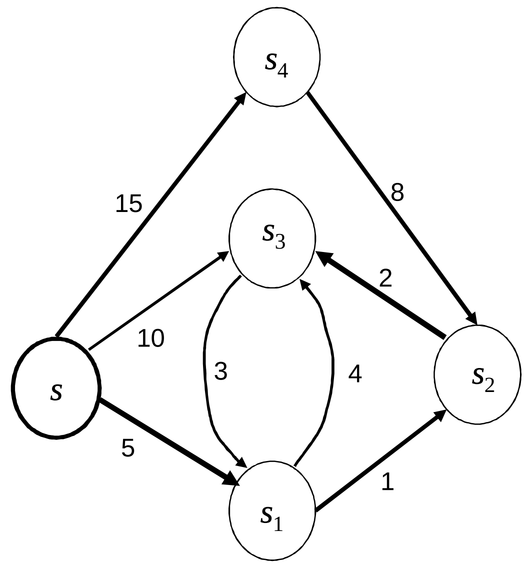
Solution = {(s, 0), (s_1, 5), (s_2, 6) }
D[s_3]=8 for path [s, s_1, s_2, s_3]
D[s_4]=15 for path [s, s_4]

Solution = {(s, 0), (s_1, 5), (s_2, 6),(s_3, 8), (s_4, 15) }

Implementing the selection rule

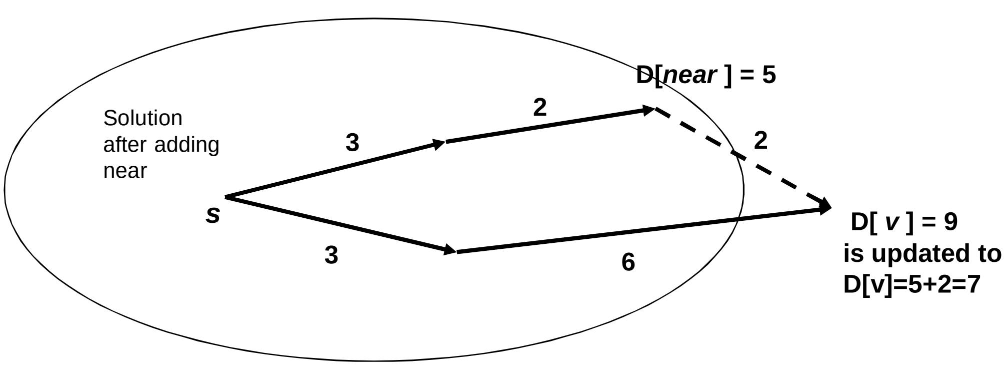
- **Node *near* is selected and added to *Solution* if D(*near*) £ D(*v*) for any *v* Ï *Solution*.**

Solution = {(s, 0)}
D[s_1]=5 £ D[s_2]= ∞
D[s_1]=5 £ D[s_3]=10
D[s_1]=5 £ D[s_4]=15
Node s_1 is selected
Solution = {(s, 0), (s_1, 5) }



Updating D[ ]

- **After adding *near* to *Solution*, D[*v*] of all nodes *v* Ï *Solution are* updated if there is a shorter special path from *s* to *v* that contains node *near*, i.e., if**
  (D[*near*] + w(*near*, *v*) < D[*v*]) then
  D[v]=D[*near*] + w(*near*, *v*)



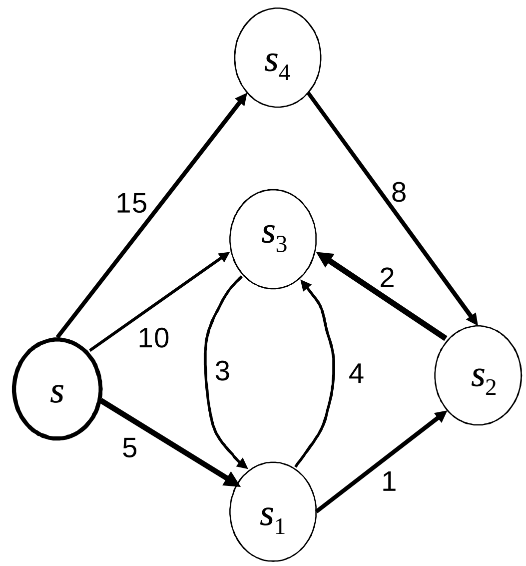
Updating D Example

Solution = {(s, 0)}
D[s_1]=5, D[s_2]= ∞ , D[s_3]=10, D[s_4]=15.

Solution = {(s, 0), (s_1, 5) }
D[s_2]= D[s_1]+w(s_1, s_2)=5+1=6,
D[s_3]= D[s_1]+w(s_1, s_3)=5+4=9,
D[s_4]=15

Solution = {(s, 0), (s_1, 5), (s_2, 6) }
D[s_3]=D[s_2]+w(s_2, s_3)=6+2=8,
D[s_4]=15

Solution = {(s, 0), (s_1, 5), (s_2, 6), (s_3, 8), (s_4, 15) }



Dijkstra's Algorithm for Finding the Shortest Distance from a Single Source

Dijkstra(G,s)

1. **for each** $v \in V$
2.   **do** $D[v] \leftarrow \infty$
3. $D[s] \leftarrow 0$
4. $PQ \leftarrow$make-PQ(D,V)
5. **while** $PQ \neq \emptyset$
6.   **do** *near* $\leftarrow$ *PQ*.extractMin ()
7.   **for each** $v \in$ Adj(*near*)
8   **if** $D[v] > D[near] + w(near, v)$
9.   **then** $D[v] \leftarrow D[near] + w(near, v)$
10.   PQ.decreasePriorityValue (D[v], v)
11. **return** the label $D[u]$ of each vertex u

Time Analysis

Using Heap implementation

1. **for each** $v \in V$
2. **do** $D[v] \leftarrow \infty$
3. $D[s] \leftarrow 0$
4. $PQ \leftarrow$ make-PQ(D,V)
5. **while** $PQ \neq \varnothing$
6. **do** $near \leftarrow PQ$.extractMin ()
7. **for each** $v \in$ Adj($near$)
8. **if** $D[v] > D[near] + w(near, v)$
9. **then** $D[v] \leftarrow D[near] + w(near,v)$
10. PQ.decreasePriorityValue ($D[v]$, v)
11. **return** the label $D[u]$ of each vertex u

Assume a node in *PQ* can be accessed in $O(1)$

** Decrease key for v requires $O(\lg V)$ provided the node in heap with v's data can be accessed in O(1)

Lines 1 -4 run in $O(V)$

Max Size of *PQ* is | V |

(5) Loop = $O(V)$ - Only decreases

(6+(5)) $O(V) * O(\lg V)$

(7+(5)) Loop = $O(\Sigma \deg(near)) = O(E)$

(8+(7+(5))) $O(1) * O(E)$

(9) $O(1)$

(10+(7+(5))) Decrease- Key operation on the heap can be implemented in $O(\lg V) * O(E)$.

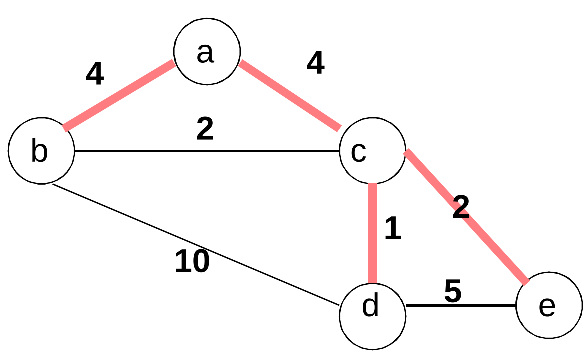
So total time for Dijkstra's Algorithm is $O(V \lg V + E \lg V)$

What is $O(E)$?

For Sparse Graph = $O(V \lg V)$

For Dense Graph = $O(V^2 \lg V)$

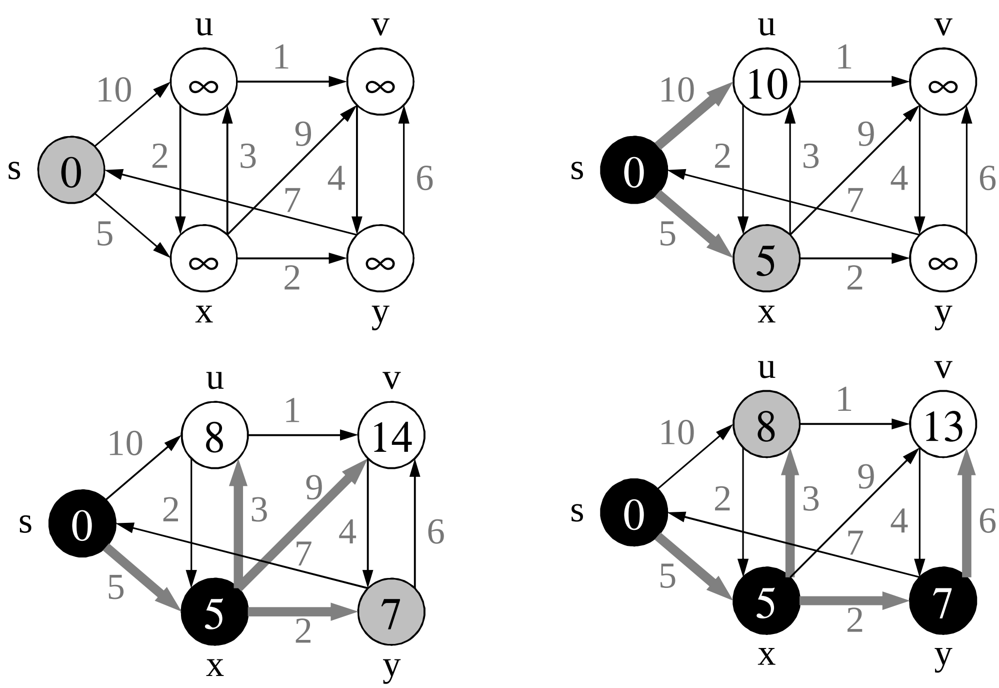
Example



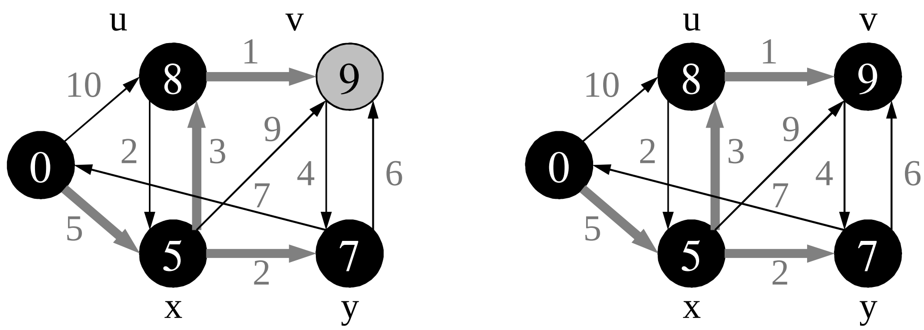
Solution for example

S	D(a)	D(b)	D(c)	D(d)	D(e)
a	0 ( )	∞ ( )	∞ ( )	∞ ( )	∞ ( )
b		4 (a, b)	4 (a, c)	∞ ( )	∞ ( )
c			4 (a, c)	14(b, d)	∞ ( )
d				5 (c, d)	6(c, e)
e					6(c, e)

Dijkstra's Example

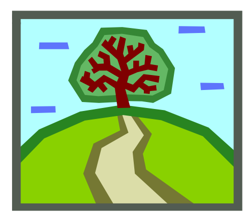


Dijkstra’s Example (2)



- Observe
  - relaxation step (lines 10-11)
  - setting *d*[*v*] updates *Q* (needs Decrease-Key)
  - similar to Prim's MST algorithm

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 z a trace-back label P[z]:
  - The parent edge in the shortest path tree
- In the edge relaxation step, we update P[Z]

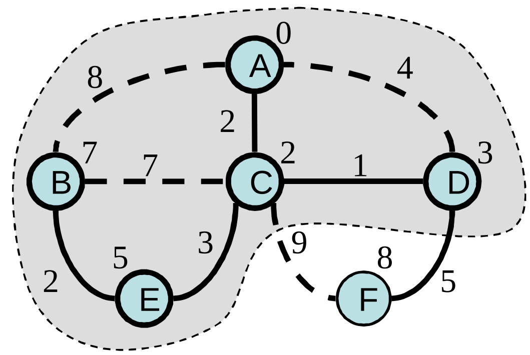
```
Algorithm DijkstraShortestPathsTree(G, s)

    …

  for all  v ∈ G.vertices()
    …
    P[v]= Æ
    …
  while ¬Q.isEmpty()
    u ← Q.removeMin()
     for each vertex z adjacent to u such that z
  is in Q
          if  D[z] < D[u]+weight(u,z) then
       D[z]  ß D[u]+weight(u,z)
             Change to D[z] the key of z in Q
          P[z]=edge(u,z)
  …
```

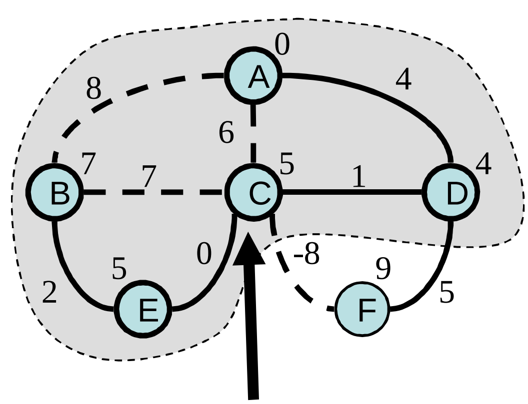
Why Dijkstra’s Algorithm Works

- 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, so long as d(F)≥d(D), F’s distance cannot be wrong. That is, there is no wrong vertex.



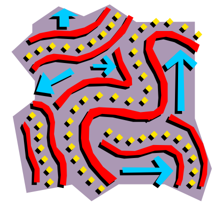
Why It Doesn’t Work for Negative-Weight Edges

- 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.



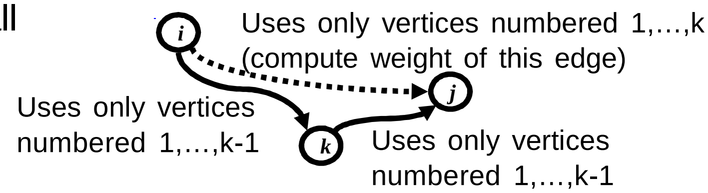
C’s true distance is 1, but it is already in the cloud with d(C)=5!

All-Pairs Shortest Paths



- Find the shortest distance between every pair of vertices in a weighted directed graph G.
- We can make n calls to Dijkstra's algorithm (if no negative edges), which takes O(nmlog n) time.
- Likewise, n calls to Bellman-Ford would take $O(n^2m)$ time.
- We can achieve $O(n^3)$ time using dynamic programming (similar to the Floyd-Warshall algorithm).

```
Algorithm AllPair(G) {assumes vertices 1,…,n}
 for all vertex pairs (i,j)
   if i = j
      D_0[i,i] ← 0
   else if (i,j) is an edge in G
      D_0[i,j] ← weight of edge (i,j)
   else
      D_0[i,j] ← + ¥
 for k ← 1 to n do
   for i ← 1 to n do
     for j ← 1 to n do
       D_k[i,j] ← min{D_{k-1}[i,j], D_{k-1}[i,k]+D_{k-1}[k,j]}
 return D_n
```



All pair shortest Path Problem

- The easiest way!
  - Iterate Dijkstra's and Bellman-Ford |V| times!
- Dijkstra:
  - O(VlgV + E) -> $O(V^2lgV + VE)$
- Bellman-Ford:
  - O(VE) -> $O(V^2E)$

On dense graph → $O(V^3)$, $O(V^4)$

- Faster-All-Pairs-Shortest-Paths
  - $O(V^3lgV)$ -> better than Dijkstra and Bellman-Ford
- Any other faster algorithms?
  - Floyd-Warshall Algorithm

Floyd-Warshall Algorithm

- Negative edges is allowed
- Assume that no negative-weight cycle
- Dynamic Programming
  - The structure of a shortest path
  - A recursive solution
  - Computing from bottom-up
  - Constructing a shortest path

The structure of a shortest path

- Intermediate vertex
  - In simple path $p = \langle v_1,\ldots,v_l \rangle$, any vertex of p other than v_1 and v_l
  - Any vertex in the set $\{v_2,\ldots,v_{l-1}\}$
- Key Observation
  - For any pair of vertices i, j in V
  - Let p be a minimum-weight path of all paths from i to j whose intermediate vertices are all from $\{1,2,\ldots,k\}$
  - Assume that we have all shortest paths from every i to every j whose intermediate vertices are from $\{1,2,\ldots,k-1\}$
  - Observe relationship between path p and above shortest paths

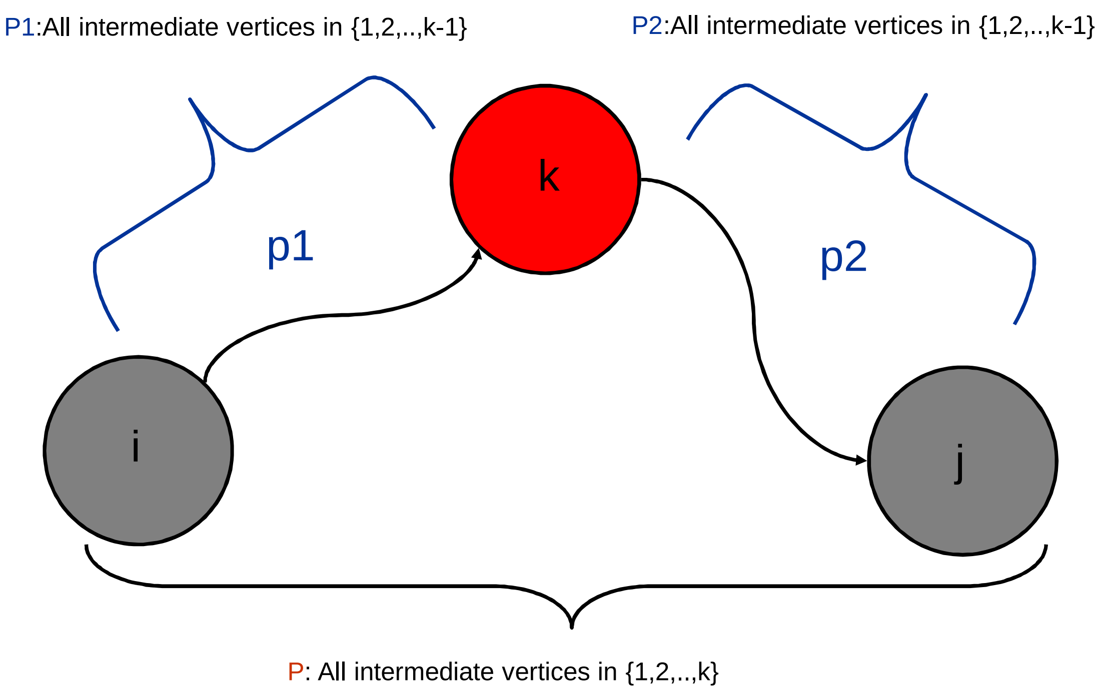
Key Observation (1)

- A shortest path does not contain the same vertex twice
  - Proof: A path containing the same vertex twice contains a cycle. Removing cycle give a shorter path.

Key Observation (2)

- P is determined by the shortest paths whose intermediate from {1,…,k-1}
- Case1: If k is not an intermediate vertex of P
  - Path P is a shortest path from i to j with intermediates from {1,…k-1}
- Case2: If k is an intermediate vertex of path P
  - Path P can be broke down into i -- ^{p1} à k - ^{p2} à j
  - P1 is the shortest path from i to k with all intermediate in the set {1,2,…,k}
  - P2 is the shortest path from k to j with {1,2,…,k}

Key Observation(2) – case2



A recursive solution

- Let $d_{ij}^{(k)}$ be the length of the shortest path from i to j such that all intermediate vertices on the path are in set {1,2,…,k}
- Let $D^{(k)}$ be the n X n matrix $[d_{ij}^{(k)}]$
- $d_{ij}^{(0)}$ is set to be w_{ij} (no intermediate vertex).
- $d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$ $(k \geq 1)$
- $D^{(n)} = (d_{ij}^{(n)})$ gives the final answer, for all intermediate are in the set {1,2,…,n}

A recursive solution

- $d_{ij}(k) = \begin{cases} w_{ij} & \text{(if } k=0) \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{(if } k\geq 1) \end{cases}$

- The Matrix $D^{(n)} = (d_{ij}^{(n)})$ gives the final answer: $d_{ij}^{(n)} = \delta(i,j)$ for all $i,j \in V$.

Extracting the Shortest Paths

- The predecessor pointers pred[i,j] can be used.

- Initially all pred[i,j] = nil

- Whenever the shortest path from i to j passing through an intermediate vertex k is discovered, we set pred[i,j] = k

Extracting the Shortest Paths (2)

- Observation:
  - If the shortest path does not pass through any intermediate vertex, then pred[i,j] = nil.
- How to find?
  - If pred[i,j] = nil, the shortest path is edge (i,j)
  - Otherwise, recursively compute (i,pred[i,j]) and (pred[i,j],j)

Computing the weights bottom up

The Floyd-Warshall Algorithm: Version 1

Floyd-Warshall(w, n)

{ for $i = 1$ to n do　　　initialize
　　for $j = 1$ to n do
　　{ $D^0[i, j] = w[i, j]$;
　　　$pred[i, j] = nil$;
　　}

　for $k = 1$ to n do　　　dynamic programming
　　for $i = 1$ to n do
　　　for $j = 1$ to n do
　　　　if $\left(d^{(k-1)}[i, k] + d^{(k-1)}[k, j] < d^{(k-1)}[i, j]\right)$
　　　　　$\{d^{(k)}[i, j] = d^{(k-1)}[i, k] + d^{(k-1)}[k, j]$;
　　　　　$pred[i, j] = k;\}$　　　case2
　　　　else $d^{(k)}[i, j] = d^{(k-1)}[i, j]$;　　　case1
　return $d^{(n)}[1..n, 1..n]$;
}

Analysis

- Running time is clearly Θ(?)
- $\Theta(n^3)$ -> $\Theta(|V|^3)$
- Faster than previous algorithms. $O(|V|^4)$,$O(|V|^3 \lg|V|)$
- Problem: Space Complexity $\Theta(|V|^3)$. It is possible to reduce this down to $\Theta(|V|^2)$by keeping only one matrix instead of n.

Modified Version

The Floyd-Warshall Algorithm: Version 2

```
Floyd-Warshall(w, n)
{ for i = 1 to n do                        initialize
    for j = 1 to n do
    { d[i, j] = w[i, j];
      pred[i, j] = nil;
    }

  for k = 1 to n do                        dynamic programming
    for i = 1 to n do
      for j = 1 to n do
        if (d[i, k] + d[k, j] < d[i, j])
            {d[i, j] = d[i, k] + d[k, j];
            pred[i, j] = k;}
  return d[1..n, 1..n];
}
```

WHY??

Transitive Closure

- Given directed graph G = (V, E)
- Compute G^* = (V, E^*)
- E^* = {(i,j) : there is path from i to j in G}
- Could assign weight of 1 to each edge, then run FLOYD-WARSHALL
- If $d_{ij} < n$, then there is a path from i to j.
- Otherwise, $d_{ij} = \infty$ and there is no path.

Transitive Closure – Solution1

- Using Floyd-Warhshall Algorithm
- Assign weight of 1 to each edge, then run FLOYD-WARSHALL with this weights.
- Finally,
  - If $d_{ij}^{(n)} < n$, then there is a path from i to j.
  - Otherwise, $d_{ij}^{(n)} = \infty$ and there is no path.

Transitive Closure – Solution2

- Using logical operations $\vee$ (OR), $\wedge$ (AND)
- Assign weight of 1 to each edge, then run FLOYD-WARSHALL with this weights.
- Instead of $D^{(k)}$, we have $T^{(k)} = (t_{ij}^{(k)})$
  - $t_{ij}^{(0)}$ = 0 (if $i \neq j$ and $(i, j) \notin E$)
    1 (if $i = j$ or $(i, j) \in E$)
  - $t_{ij}^{(k)}$ = 1 (if there is a path from i to j with all intermediate vertices in {1, 2,…, k})

    ($t_{ij}^{(k-1)}$ is 1) or ($t_{ik}^{(k-1)}$ is 1 and $t_{kj}^{(k-1)}$ is 1)

    0 (otherwise)

Transitive Closure – Solution2

```
TRANSITIVE-CLOSURE(E, n)
for i = 1 to n
    do for j = 1 to n
        do if   i=j  or  (i, j) ∈ E
                then  t_ij^(0) = 1
                else   t_ij^(0) = 0

for k = 1 to n
    do for  i = 1 to n
            do for  j = 1 to n
                    do t_ij^(k) = t_ij^(k-1) ∨ ( t_ik^(k-1) ∧ t_kj^(k-1) )
return T^(n)
```