PBW 654
Applied Statistics - I
Urban Operations Research

Unit 3

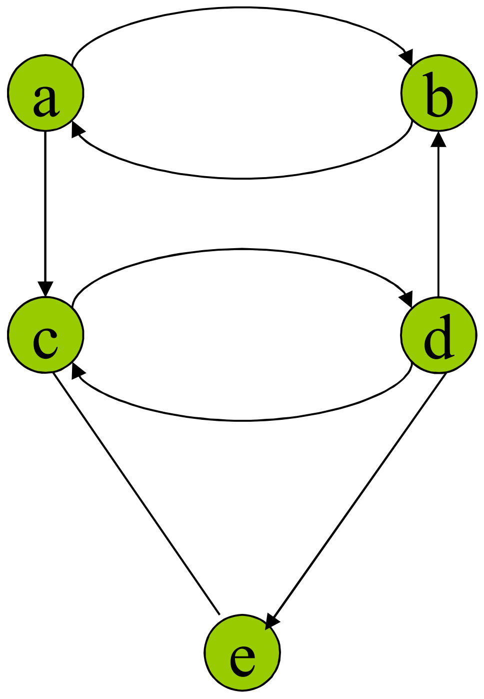
Network Modelling

Background

- So far, we treated urban space as a “continuum”, where an entity could travel from any point to any other point
- In reality, urban space is a “discretized space”, where travel occurs through discrete facilities such as highways, streets, bus routes, subway lines, walkways, etc.
- Such discretized space is a “transportation network”
- Network Theory (or Graph Theory) provides useful methods to analyze networks

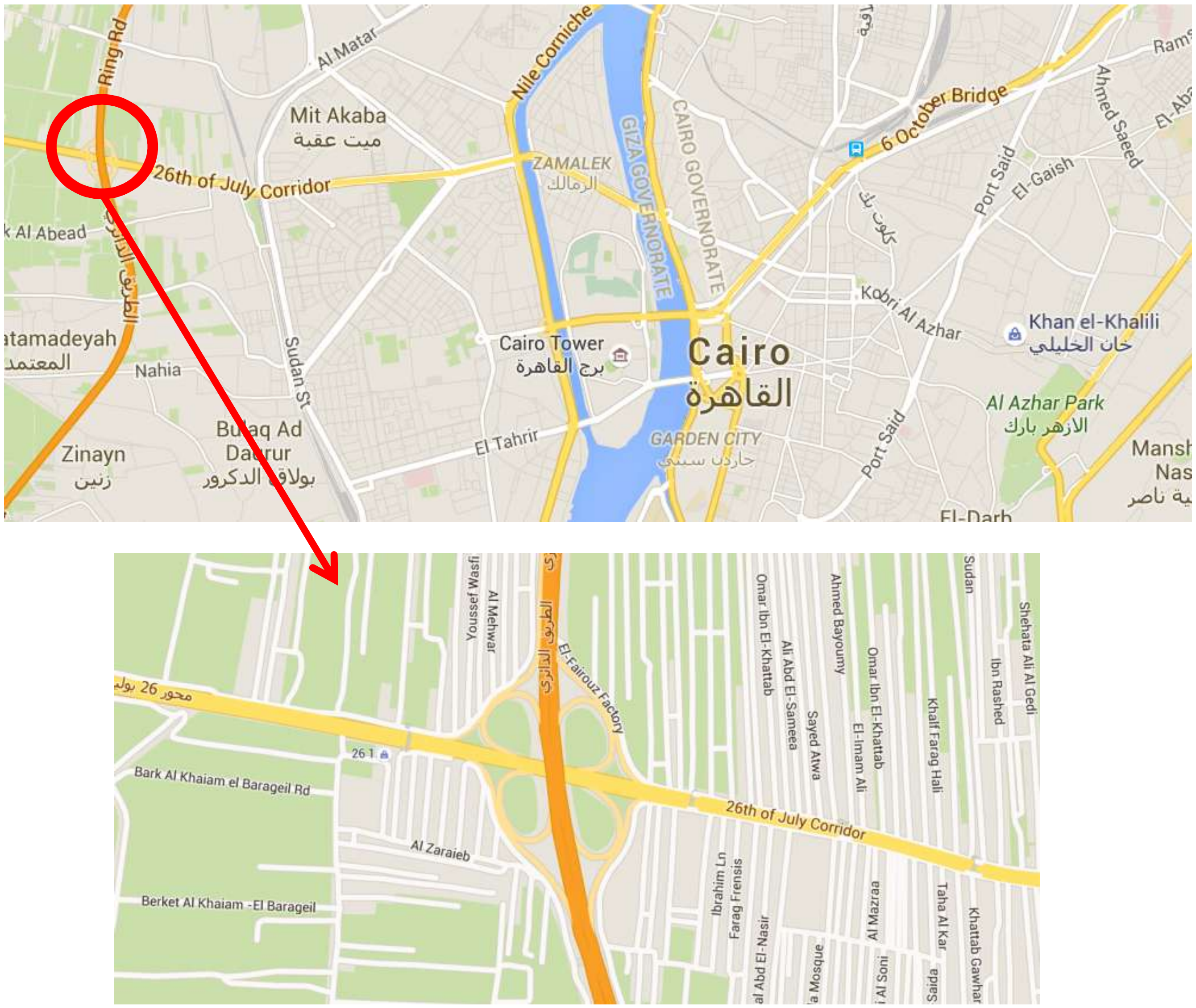
Terms and Notations

- Consider the shown network (aka graph)
  - *a*,*b*,*c*,*d* and *e* are called nodes, and the set *N* includes all nodes
  - (*a*,*b*),(*a*,*c*),(*b*,*a*),(*b*,*d*),….. are called arcs (or links, edges), and the set *A* consists of all arcs
  - Any network/graph is defined as *G*(*N*,*A*)

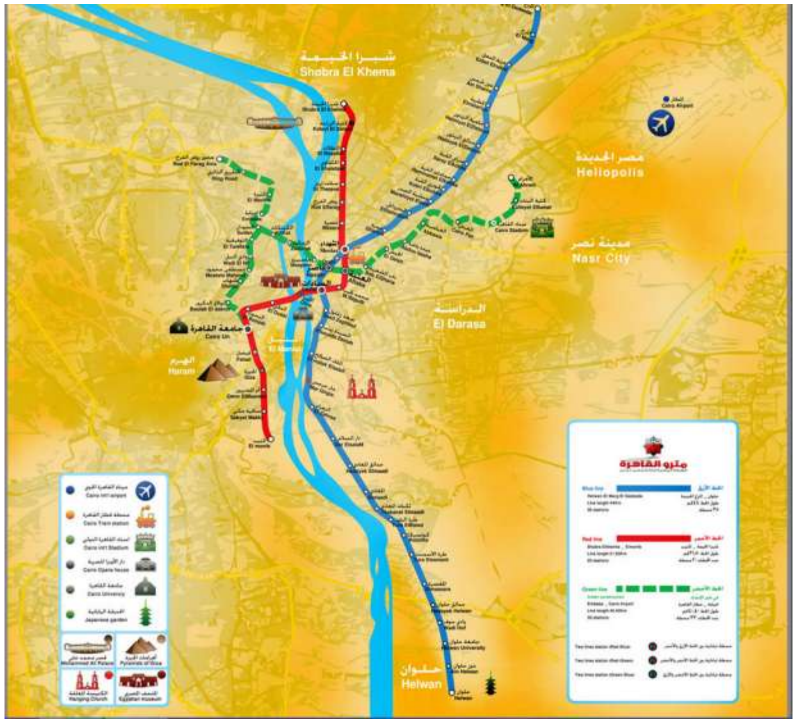


- Example networks
  - Road network
    - Nodes: intersections, freeway interchanges
    - Links: road sections, freeway sections
  - Transit network
    - Nodes: transit stops, stations
    - Links: route/line segments
  - Rail network, telecommunication network,etc.

Example – Roadway Networks



Example – Transit Networks



El Monira
Embaba
Sudan
KitKat
El Tawfikia
Zamalek
Wadi El Nil
Maspero
Shodaa
Orabi
Nasser
Sadat
Attaba
Bab ElSharia
El Geish
Abdou basha
Ghamra
ElDamrdash
Masara
Rod Elfarag
Shehab
El Dokki
Opra
El Bohoth
M Naguib
Saad Zaghloul

Terms and Notations (cont'd)

- Arcs could be directed (e.g. arc *a,b*) or undirected (e.g. arc *c,e*)
- If *A* consists of directed arcs only, then *G* is a directed graph
- If *A* consists of undirected arcs only, then *G* is an undirected graph
- If *A* consists of some directed and some undirected arcs, then *G* is a mixed graph (our example network is a mixed graph)
- Arc (*a,b*) is incident on nodes *a* and *b*
- Nodes *a* and *b* are called adjacent since they are connected by an arc (i.e. *a,b*)
- Arcs (*a,b*) and (*a,c*) are adjacent since they are connected by node *a*
- In an undirected graph, the degree of a node is the number of arcs incident on it

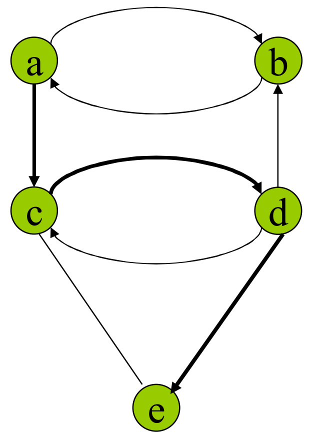
Terms and Notations (cont'd)

- In a directed graph
  - the <u>in-degree of a node</u> is the number of arcs leading into the node
  - the <u>out-degree of a node</u> is the number of arcs leading away from the node

- A <u>path</u> between 2 nodes is a sequence of adjacent arcs and nodes
  - A possible path between *a* and *e* is:

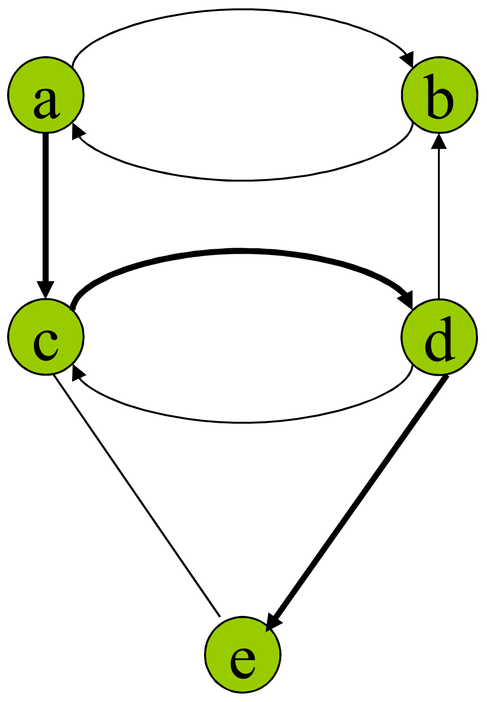
a-->c-->d-->e

  - We usually write such a path as
    - S = {*a,c,d,e*}
  - Could also be written as
    - S = {(*a,c*),(*c,d*),(*d,e*)}



Terms and Notations (cont'd)

- A path is
  - simple if each arc appears only once in *S*
  - elementary if each node appears only once in *S*



Terms and Notations (cont'd)

- A special path is a cycle (e.g. *c,d,e,c*)
- Any node, *i*, is connected to node *j* if there is a path from *i* to *j*
  - Note that in our example graph, *e* is not connected to *b*
- A connected undirected graph is one where a path exists between each pair of nodes $i \in N$ and $j \in N$
- A strongly connected directed graph is one where a path exists from each node $i \in N$ to each other node $j \in N$, and *vice versa*
- If $N' \subset N$ and $A' \subset A$, then $G'(N',A')$ is a subgraph of graph $G(N,A)$
- A tree of an undirected graph is a connected subgraph having no cycles
- A tree of *t* nodes contains *t-1* arcs, and there exists a single path between any 2 nodes on the tree
- A spanning tree of $G(N,A)$ is a tree containing all nodes of N

Terms and Notations (cont'd)

- In almost all transportation networks, each link (i,j) has a length $l(i,j)$, which can be distance, time, cost, etc.
- The length of a path S is $L(S)$ and it is equal to the summation of all $l(i,j)$ where $(i,j) \in S$
- The shortest path between 2 nodes i and j will be denoted as $d(i,j)$

Terms and Notations (cont'd)

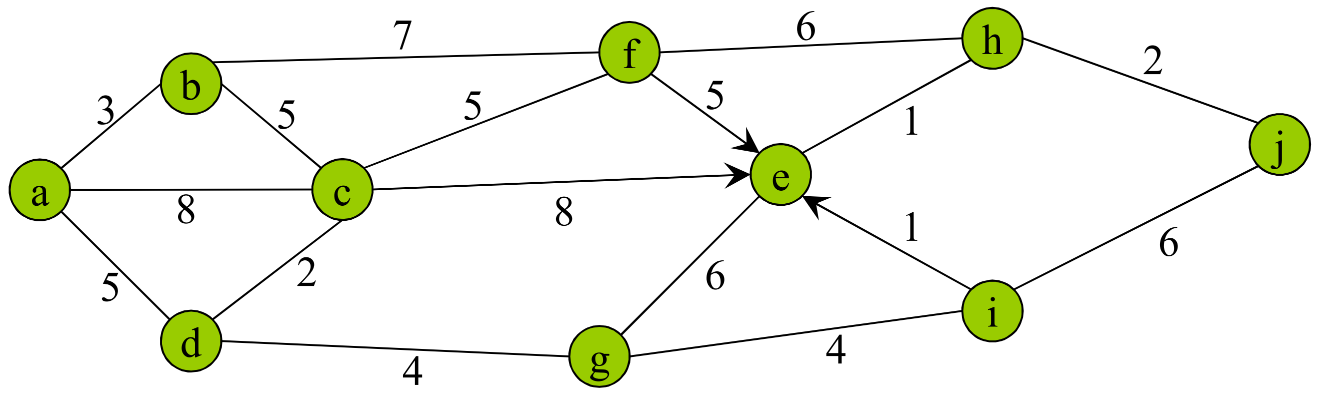
- Famous network problems
  - Shortest Path Problem
    - Find the shortest path from an origin node to a destination node
    - Direct application to dispatching problems
    - Applied indirectly to many other network problems
  - Travelling Salesman Problem (TSP) – aka Node Covering Problem
    - Find the shortest route starting from a given node, visiting all members of a specified set of nodes at least once, then returning to the initial node
    - Applications include delivery services, demand-responsive bus services,etc.

Terms and Notations (cont'd)

- Famous network problems (cont'd)
  - Chinese Postman Problem (CPP) – aka Edge Covering Problem (or Route Inspection Problem)
    - Find the shortest route, starting from a given node, travelling members of a specified set of arcs at least once, then returning to the initial node
    - Applications include street sweeping, snow plowing, garbage collection, etc.

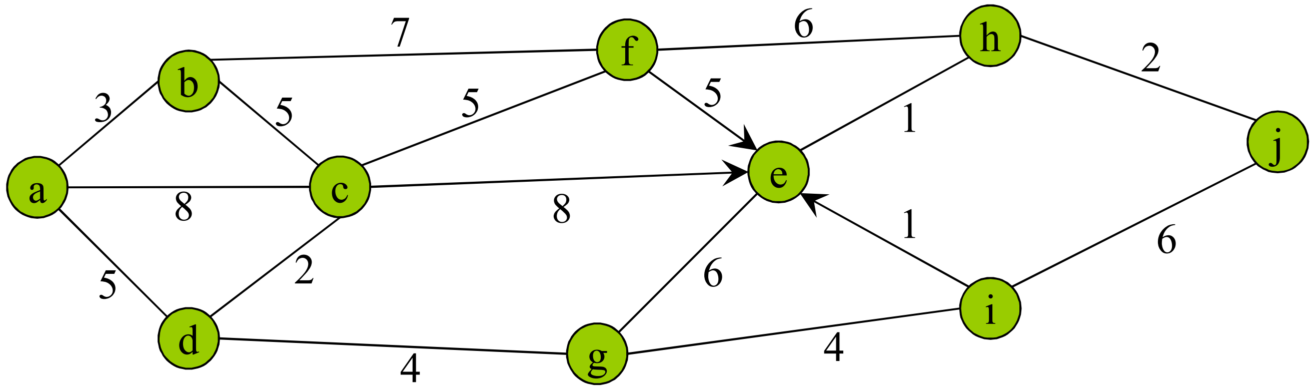
Shortest Path Problem

- 2 types
  - Shortest paths from a given node (called "source" node) to all other nodes
    - Solved using Dijkstra's Node-Labelling Algorithm
  - Shortest paths between all pairs of nodes
    - Solved using the Floyd Algorithm



Shortest Path Problem

- For the mixed graph below, find the shortest paths from "a"
  - Use Dijkstra's Node-Labelling Algorithm
  - The algorithm consists of beginning at the specified node s (the "source" node) and then successively finding its closest, second closest, third closest, and so on, node, one at a time, until all nodes in the network have been exhausted.



Shortest Path Problem

- Dijkstra’s Node-Labelling Algorithm
  - Attach to each node **x** a “label” (d,p)

    length of the shortest path from s to x

    number of preceding node on the shortest path from s to x
- In the evolution of the algorithm, each node can be in one of two states:
  - In the open state, when its label is still tentative; or
  - In the closed state, when its label is permanent

Shortest Path Problem

- Dijkstra's Node-Labelling Algorithm

STEP 1 - To initialize the process set d(s) = 0. p(s) = *;

- set d(j) = ∞, p(j) = - for all other nodes j ≠s;
- consider node ***s*** as closed and all other nodes as open;
- set k = s (i.e., s is the last closed node)

STEP 2 - To update the labels, examine all edges (k, j) out of the last closed node;

- if node **j** is closed, go to the next edge;
- if node **j** is open, set the first entry of its label to

d(j) = Min [d(j), d(k) + (k,i)]

STEP 3 - To choose the next node to close, compare the d(j) parts of the labels for all nodes that are in the open state.

- Choose the node with the smallest d(j) as the next node to be closed. Suppose that this is node **i**

Shortest Path Problem

- Dijkstra's Node-Labelling Algorithm (cont'd)

STEP 4 - To find the predecessor node of the next node to be closed, i, consider, one at a time, the edges (j, i) leading from **closed** nodes to **i** until one is found such that

$$d(i) - (a, i) = d(j)$$

- Let this predecessor node be j*. Then set p(i) = j*

STEP 5 - Now consider node **i** as a closed node.

- If all nodes in the graph are closed, then stop; the procedure is finished.

- If there are still some open nodes in the graph, set **k = i** and return to Step 2.

Shortest Path Problem

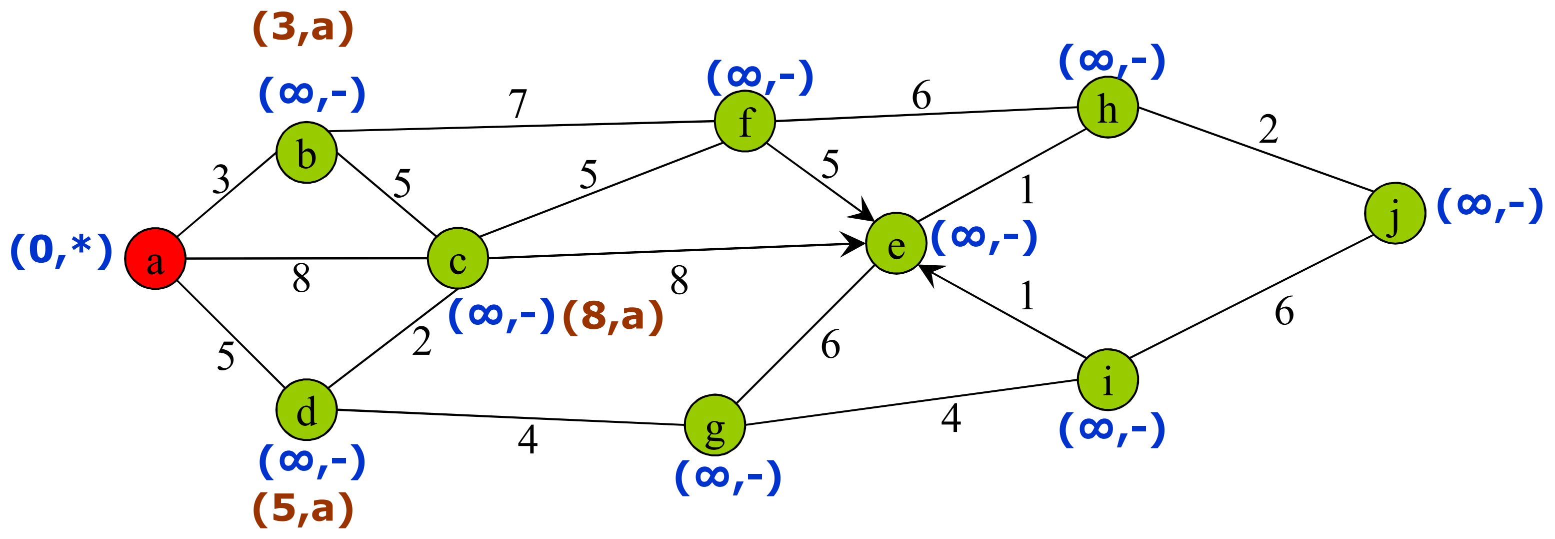
Dijkstra’s Node-Labelling Algorithm

k=a -- update node j for all (k,j)

smallest d(j)=3 -- for node **b**

set i=b --> consider **a** as the predecessor for **b**

set **b** as a **closed** node



Shortest Path Problem

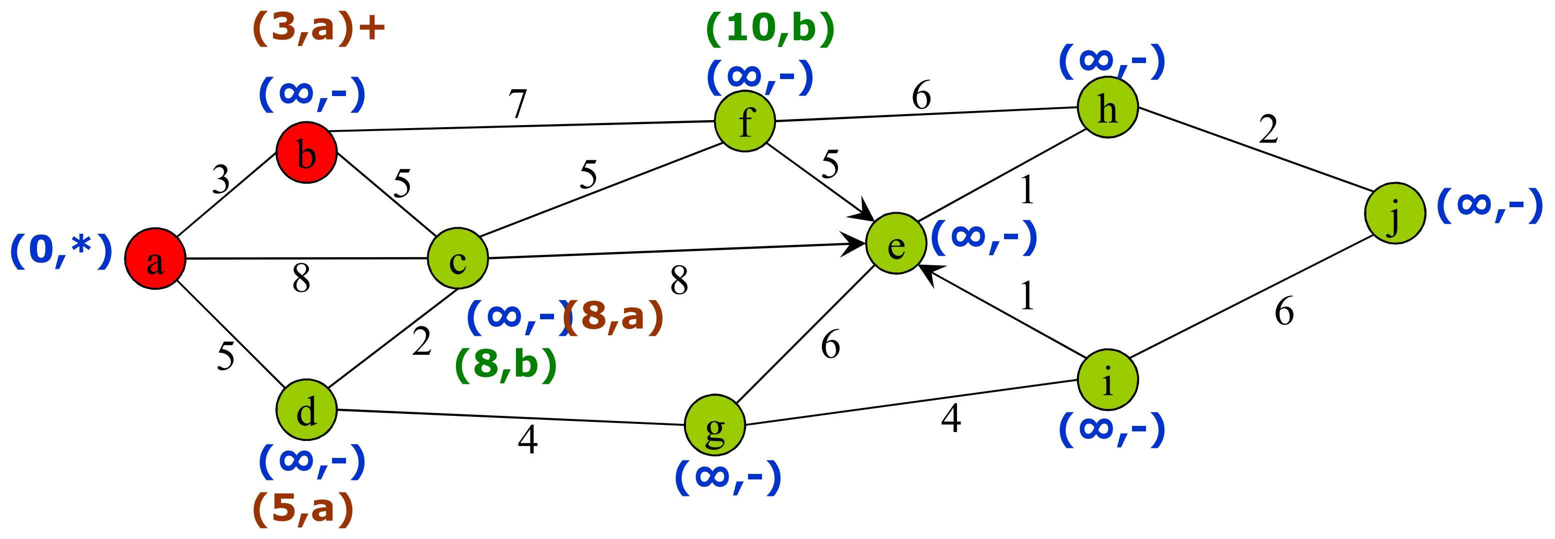
Dijkstra’s Node-Labelling Algorithm

k=b -- update node j for all (k,j)

smallest d(j)=5 -- for node **d**

set i=d --> consider **a** as the predecessor for **d**

set **d** as a **closed** node



Shortest Path Problem

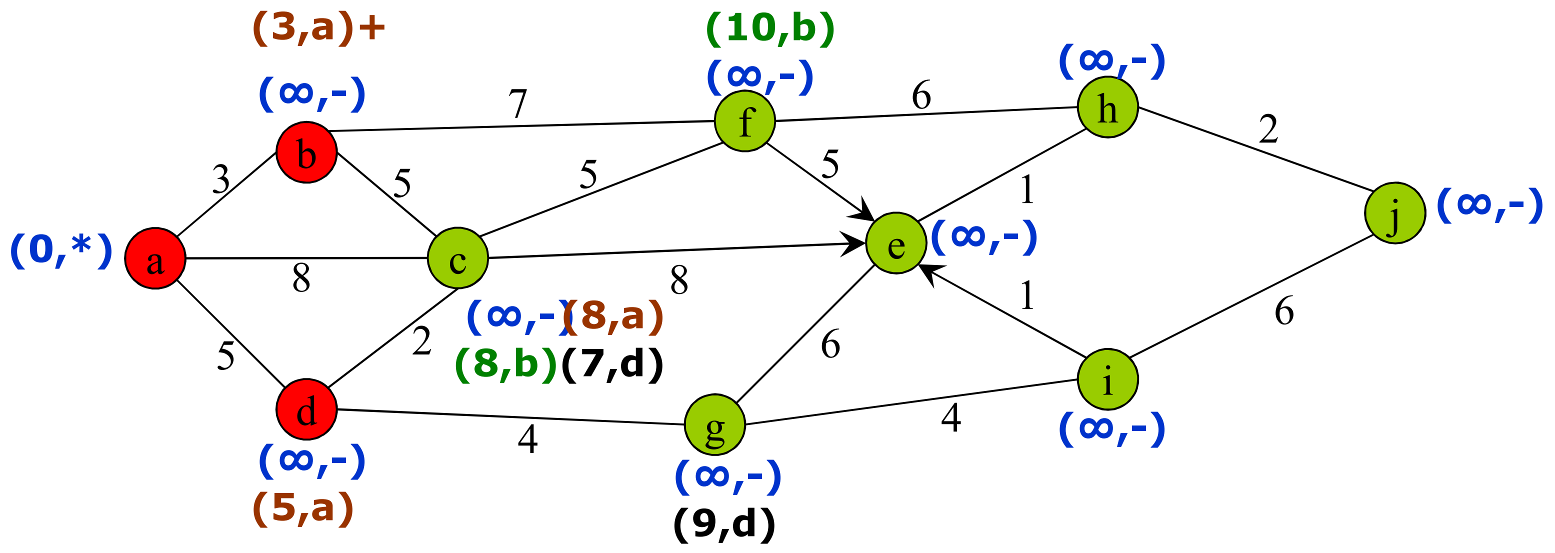
Dijkstra's Node-Labelling Algorithm

k=d --update node j for all (k,j)

smallest d(j)=7 -- for node **c**

set i=c --> **d** is the predecessor for **c**, since *d(c)[7] - (d,c)[2]=d(d)[5]*

set **c** as a **closed** node



Shortest Path Problem

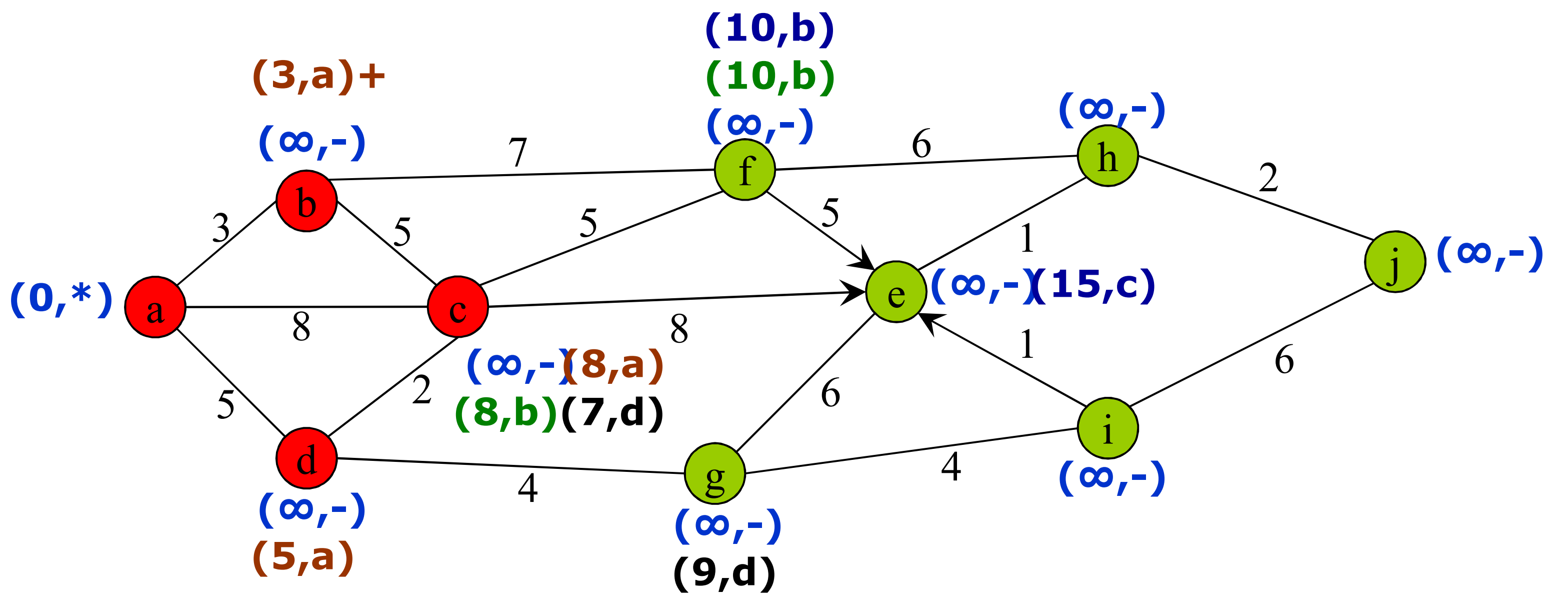
Dijkstra’s Node-Labelling Algorithm

k=c --update node j for all (k,j)

smallest d(j)=9 -- for node **g**

set i=g --> **d** is the predecessor for **g**, since *d(g)[9] - (d,g)[4]=d(d)[5]*

set **g** as a **closed** node



Shortest Path Problem

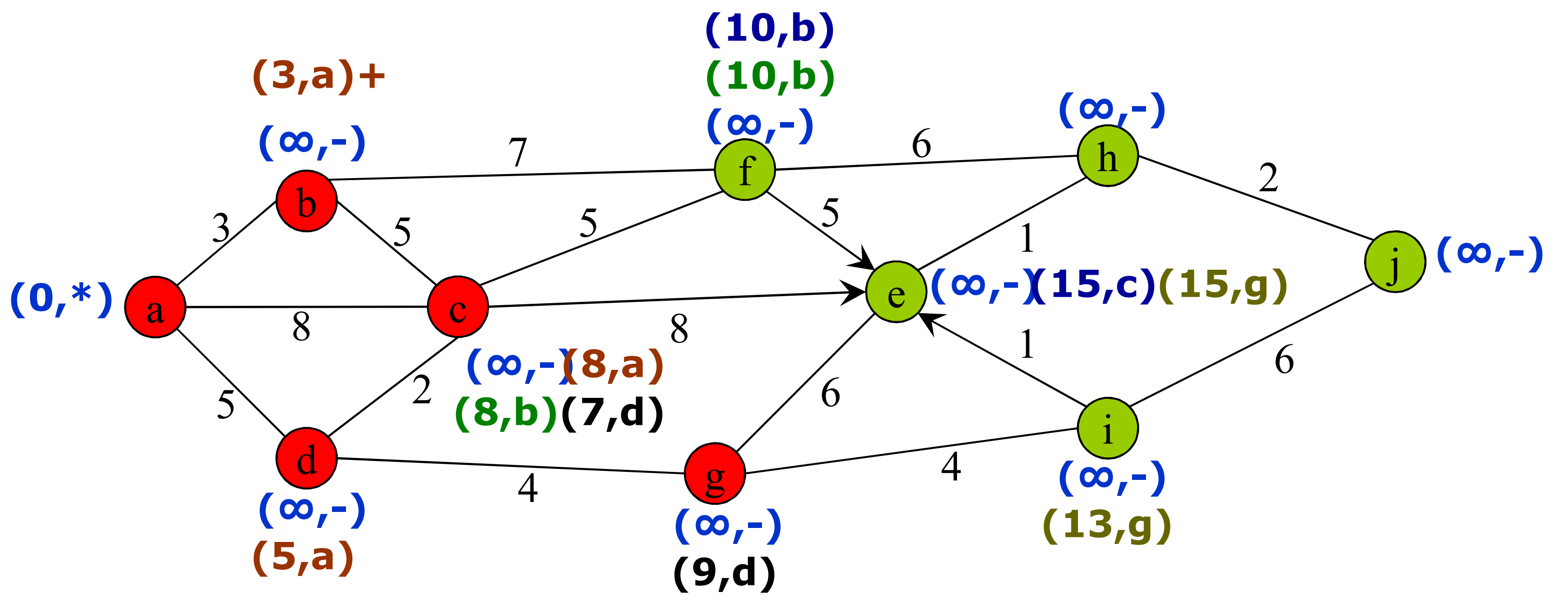
Dijkstra's Node-Labelling Algorithm

k=g --update node j for all (k,j)

smallest d(j)=10 -- for node **f**

set i=f --> **b** is the predecessor for **f**, since *d(f)[10] - (b,f)[7]=d(b)[3]*

set **f** as a **closed** node



Shortest Path Problem

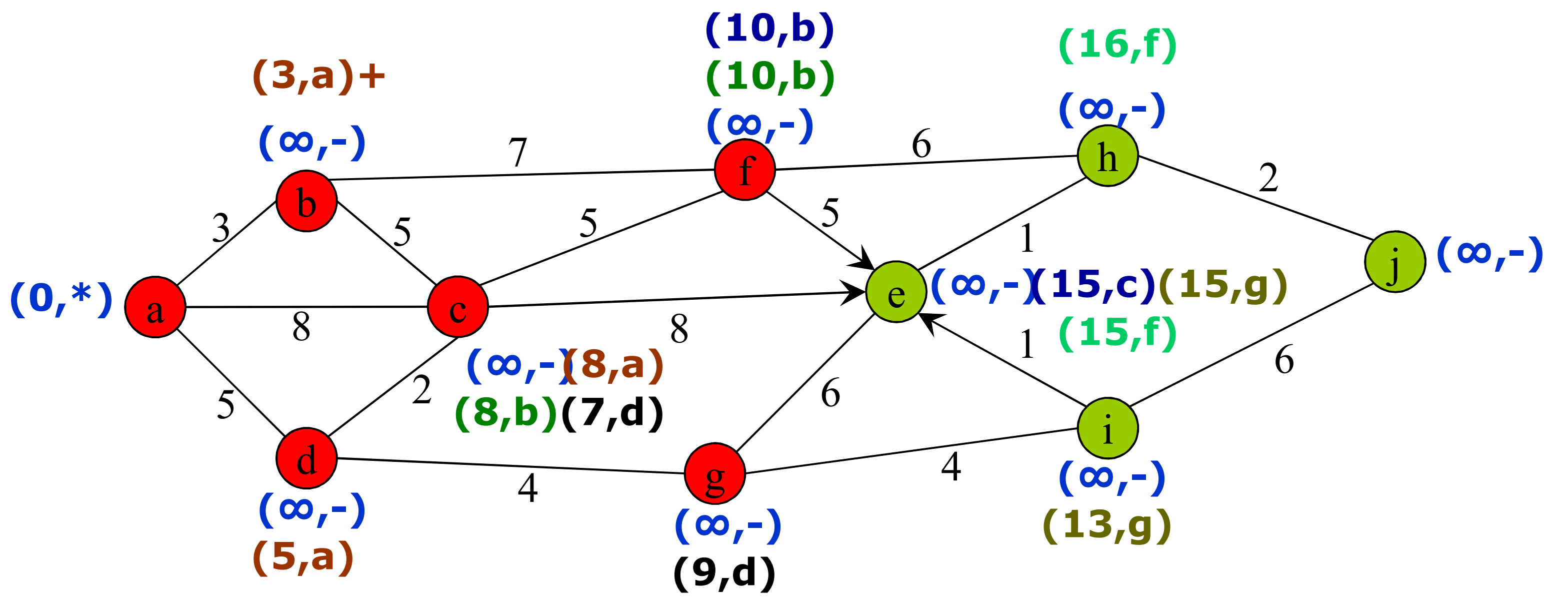
Dijkstra’s Node-Labelling Algorithm

k=f --update node j for all (k,j)

smallest d(j)=13 -- for node **i**

set i=**i** --> **g** is the predecessor for **i**, since *d(i)[13] - (g,i)[4]=d(g)[9]*

set **i** as a **closed** node



Shortest Path Problem

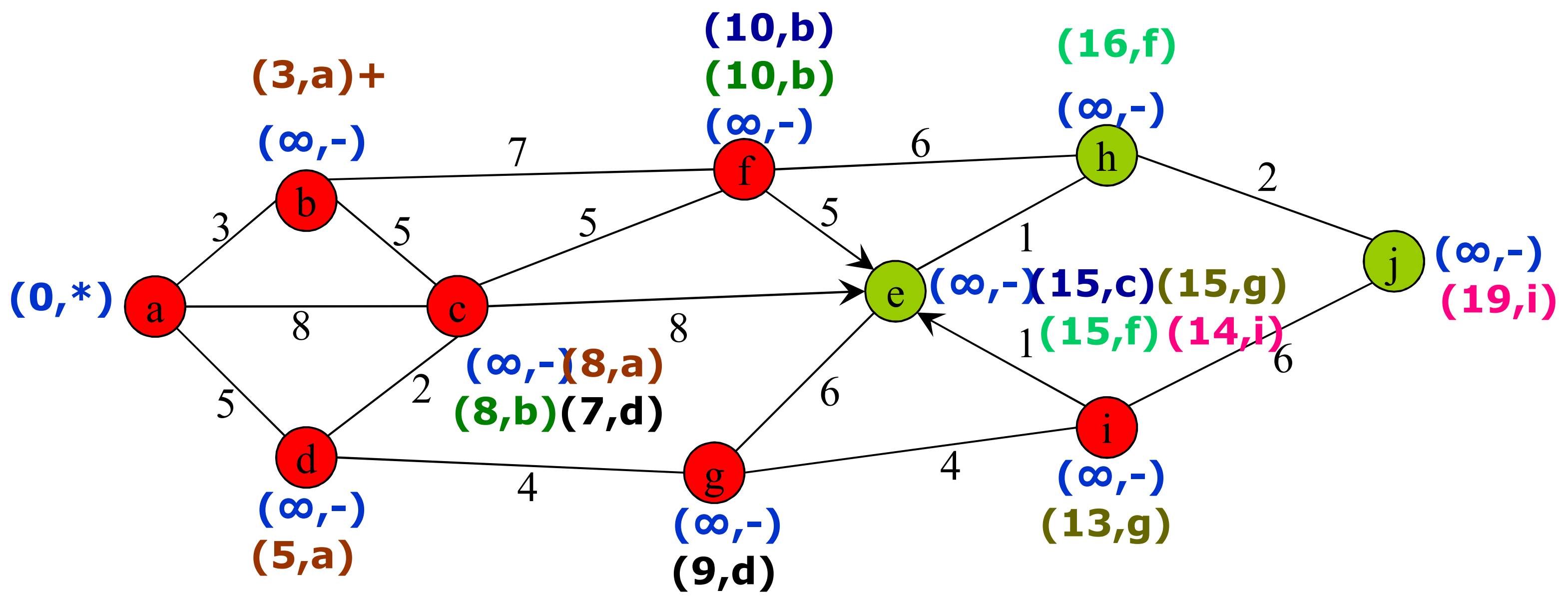
- Dijkstra's Node-Labelling Algorithm

k=i --update node j for all (k,j)

smallest d(j)=14 -- for node **e**

set i=**e** --> **i** is the predecessor for **e**, since *d(e)[14] - (i,e)[1]=d(i)[13]*

set **e** as a **closed** node



Shortest Path Problem

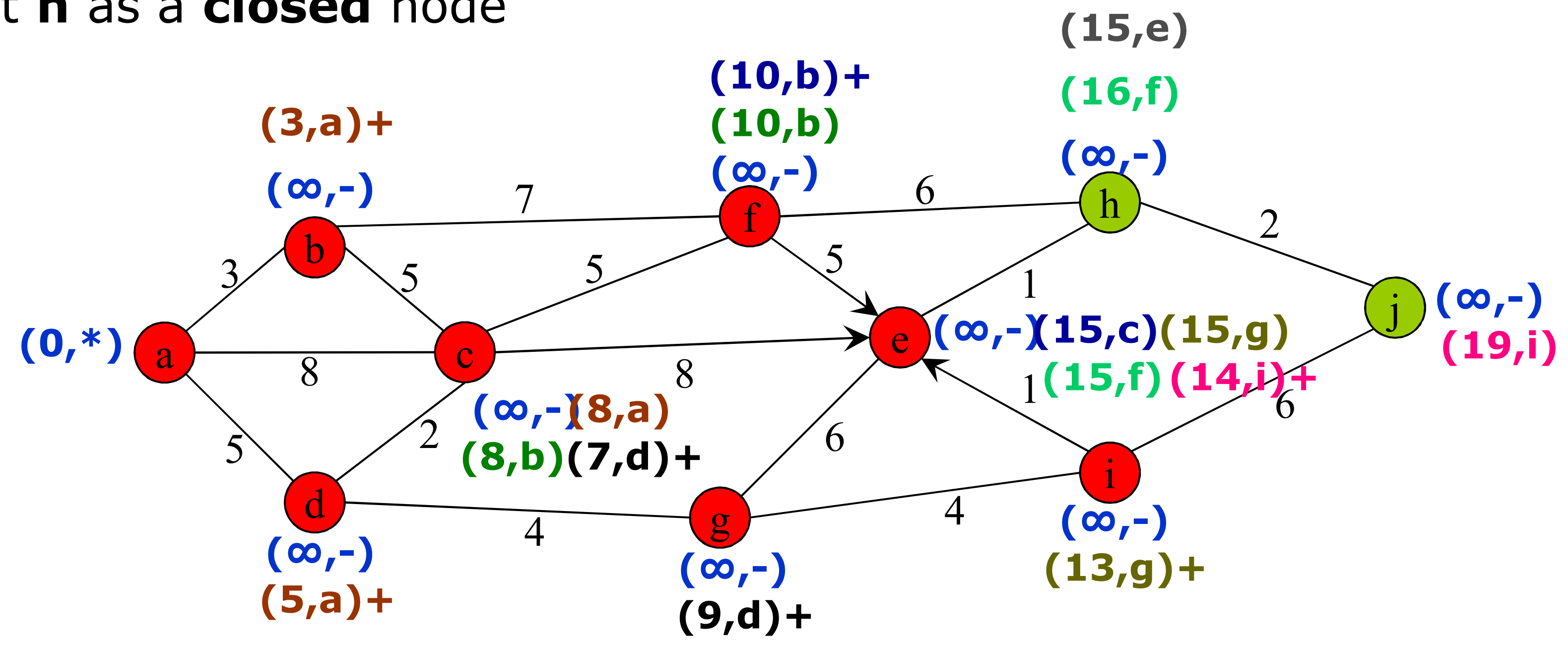
Dijkstra's Node-Labelling Algorithm

k=e --update node j for all (k,j)

smallest d(j)=15 -- for node **h**

set i=**h** --> **e** is the predecessor for **h**, since *d(h)[15] - (e,h)[1]=d(e)[14]*

set **h** as a **closed** node



Shortest Path Problem

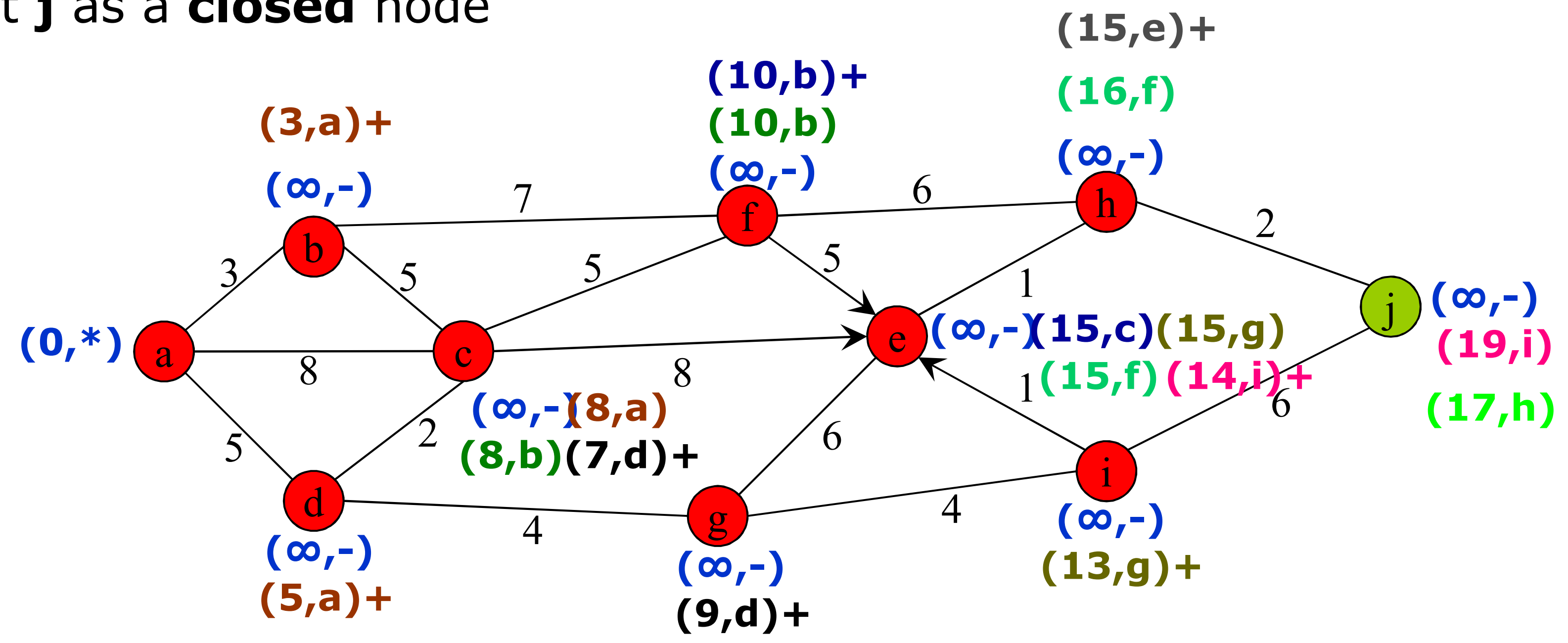
Dijkstra's Node-Labelling Algorithm

k=h --update node j for all (k,j)

smallest d(j)=17 -- for node **j**

set i=**j** --> **h** is the predecessor for **j**, since *d(j)[17] - (h,j)[2]=d(h)[15]*

set **j** as a **closed** node



Shortest Path Problem

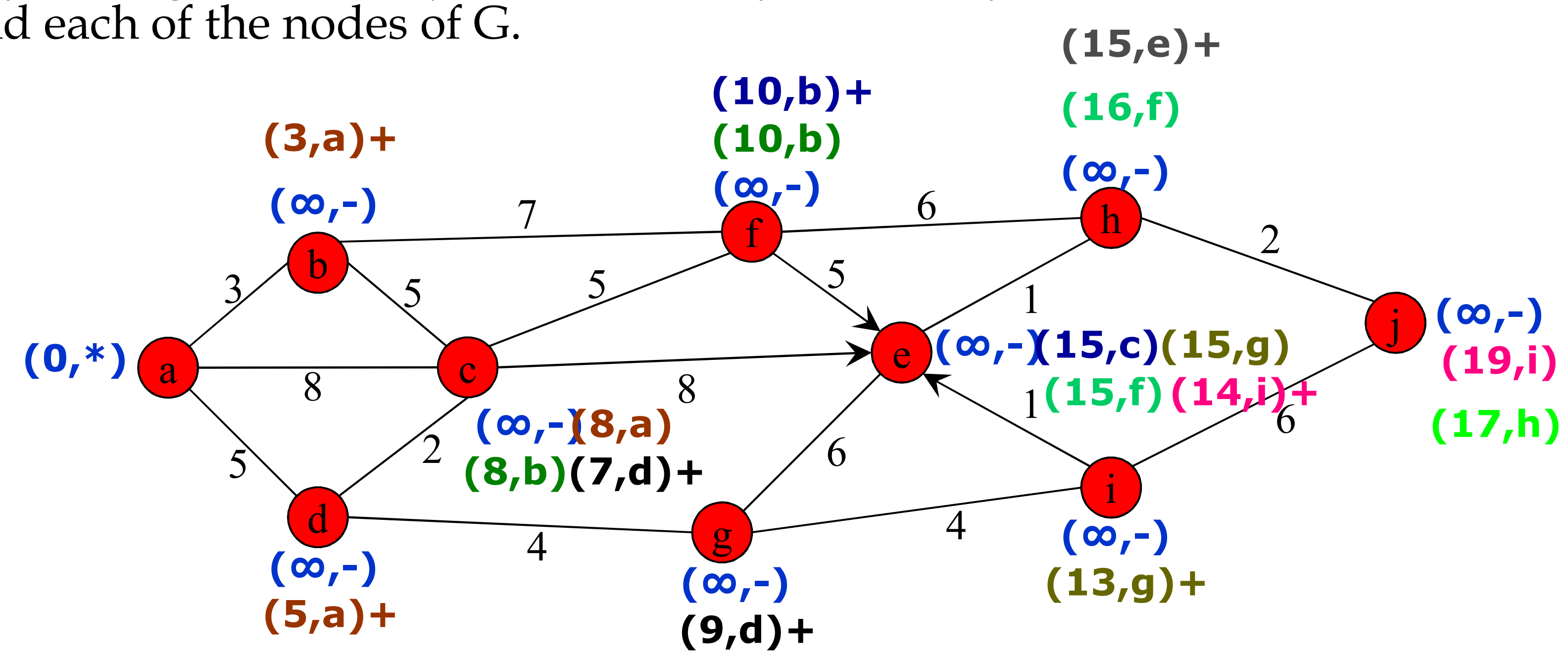
Dijkstra's Node-Labelling Algorithm

<u>Terminate</u> -- all nodes are **closed**

-The d(j) part of the label of node j indicates the length of the shortest path from s to j,

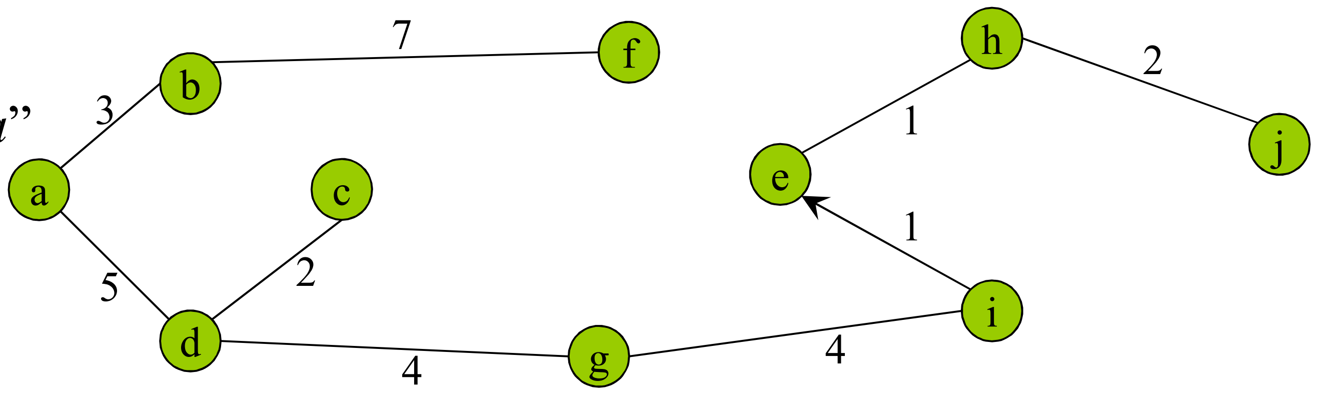
- The p(j) part indicates the predecessor node to **j** on this shortest path.

- By tracing back the p(j) parts, it is easy to identify the shortest paths between s and each of the nodes of G.



Shortest Path Problem

Shortest path tree for node "*a*"



Shortest Path Problem

- Notes on Dijkstra’s Algorithm
  - If ties among node labels occur, choose one of the them arbitrarily
    - If 2 nodes have the smallest label values at some point and qualify to be closed, then choose one of the two arbitrarily
  - The algorithm can be used to find the shortest path between an origin and one destination
    - Terminate when you close the destination node
  - Could apply the algorithm repeatedly for each node to find the shortest paths between all pairs of nodes
    - That is, consider each node at a time as a source and solve
    - A more efficient algorithm is the Floyd Algorithm

Shortest Path Problem

- Floyd Algorithm
  - Floyd's algorithm is simple to describe, but its logic is not particularly easy to grasp.
  - The algorithm, in parallel with a convenient procedure for maintaining a record of the shortest paths

Shortest Path Problem

- Floyd Algorithm
  - To begin, we number the **n** nodes of the graph G(N, A) with the positive integers 1, 2, . . ., n.
  - Then, two matrices, a distance matrix, $D^{(0)}$, and a predecessor matrix, $P^{(0)}$, are set up with elements

$$d_0(i,j) = \begin{cases} \ell(i,j) & \text{if arc } (i,j) \text{ exists}^3 \\ 0 & \text{if } i = j \\ \infty & \text{if arc } (i,j) \text{ does not exist} \end{cases}$$

and

$$p_0(i,j) = \begin{cases} i & \text{for } i \neq j \\ -\text{(blank)} & \text{for } i = j \end{cases}$$

Shortest Path Problem

- Floyd Algorithm

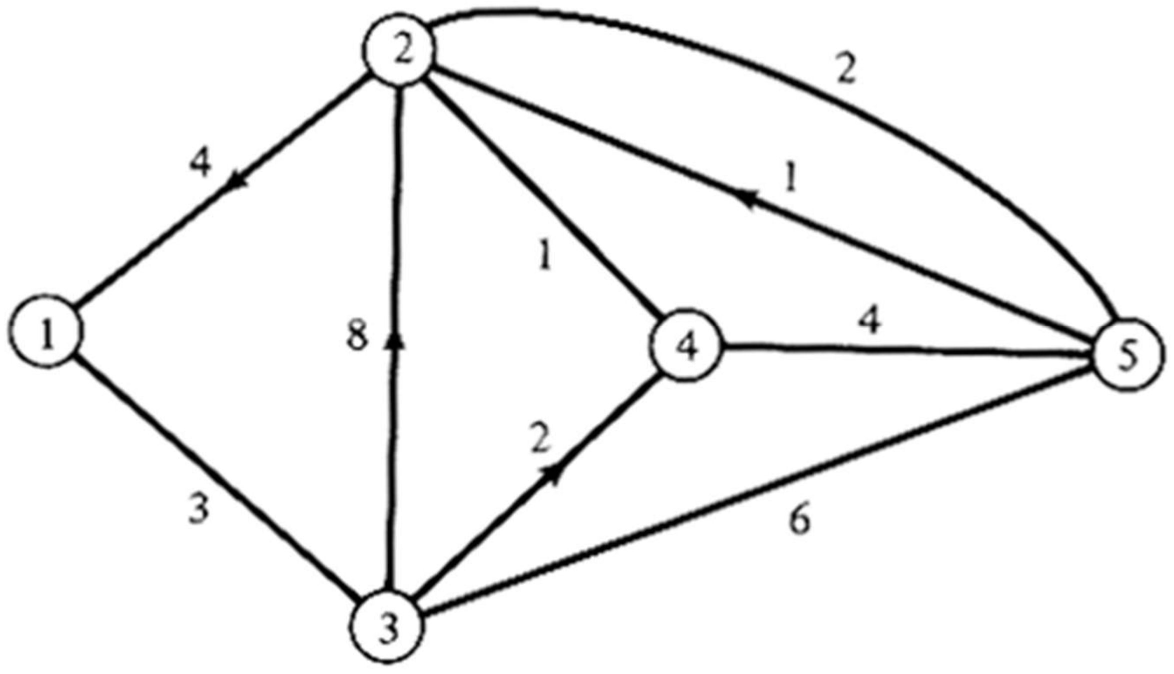
STEP 1	Set k = 1.
STEP 2	Obtain all the elements of the updated distance matrix D(k) from the relation below
STEP 3	Obtain all the elements of the updated predecessor matrix p(k) by using equation below
STEP 4	If k = n, stop; if k < n, set k = k + 1 and return to step 2

$$d_k(i,j) = \text{Min}\,[d_{k-1}(i,j),\, d_{k-1}(i,k) + d_{k-1}(k,j)] \qquad (6.2)$$

$$p_k(i,j) = \begin{cases} p_{k-1}(k,j) & \text{if } d_k(i,j) \neq d_{k-1}(i,j) \\ p_{k-1}(i,j) & \text{otherwise} \end{cases} \qquad (6.3)$$

Shortest Path Problem (continued)

- Floyd Algorithm - Example (cont'd)

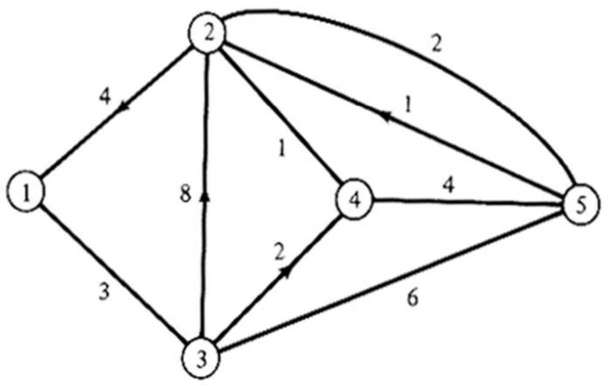


Shortest Path Problem (continued)

- Floyd Algorithm - Example (cont'd)

Initialization:

$$D^{(0)} = \begin{array}{c|ccccc|} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & \infty & 3 & \infty & \infty \\ 2 & 4 & 0 & \infty & 1 & 2 \\ 3 & 3 & 8 & 0 & 2 & 6 \\ 4 & \infty & 1 & \infty & 0 & 4 \\ 5 & \infty & 1 & 6 & 4 & 0 \\ \hline \end{array} \qquad P^{(0)} = \begin{array}{c|ccccc|} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & - & 1 & 1 & 1 & 1 \\ 2 & 2 & - & 2 & 2 & 2 \\ 3 & 3 & 3 & - & 3 & 3 \\ 4 & 4 & 4 & 4 & - & 4 \\ 5 & 5 & 5 & 5 & 5 & - \\ \hline \end{array}$$



Through Node 1:

$$D^{(1)} = \begin{array}{c|ccccc|} & 1^* & 2 & 3 & 4 & 5 \\ \hline {}^*1 & 0 & \infty & 3 & \infty & \infty \\ 2 & 4 & 0 & 7^+ & 1 & 2 \\ 3 & 3 & 8 & 0 & 2 & 6 \\ 4 & \infty & 1 & \infty & 0 & 4 \\ 5 & \infty & 1 & 6 & 4 & 0 \\ \hline \end{array} \qquad P^{(1)} = \begin{array}{c|ccccc|} & 1^* & 2 & 3 & 4 & 5 \\ \hline {}^*1 & - & 1 & 1 & 1 & 1 \\ 2 & 2 & - & 1^+ & 2 & 2 \\ 3 & 3 & 3 & - & 3 & 3 \\ 4 & 4 & 4 & 4 & - & 4 \\ 5 & 5 & 5 & 5 & 5 & - \\ \hline \end{array}$$

Shortest Path Problem (continued)

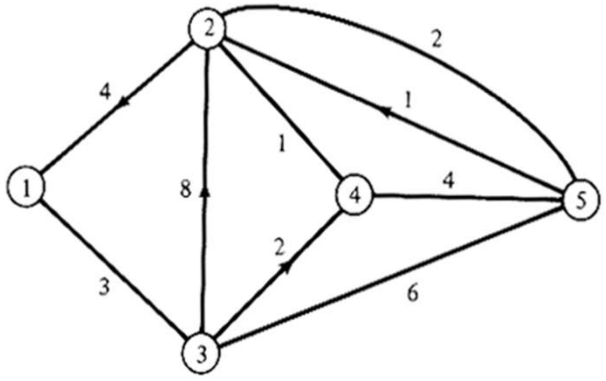
- Floyd Algorithm - Example (cont'd)

Through Node 2:

$$D^{(2)} = \begin{array}{c|ccccc} & 1 & 2^{\bullet} & 3 & 4 & 5 \\ \hline 1 & 0 & \infty & 3 & \infty & \infty \\ {}^{\bullet}2 & 4 & 0 & 7 & 1 & 2 \\ 3 & 3 & 8 & 0 & 2 & 6 \\ 4 & 5^{+} & 1 & 8^{+} & 0 & 3^{+} \\ 5 & 5^{+} & 1 & 6 & 2^{+} & 0 \end{array} \qquad P^{(2)} = \begin{array}{c|ccccc} & 1 & 2^{\bullet} & 3 & 4 & 5 \\ \hline 1 & — & 1 & 1 & 1 & 1 \\ {}^{\bullet}2 & 2 & — & 1 & 2 & 2 \\ 3 & 3 & 3 & — & 3 & 3 \\ 4 & 2^{+} & 4 & 1^{+} & — & 2^{+} \\ 5 & 2^{+} & 5 & 5 & 2^{+} & — \end{array}$$

Through Node 3:

$$D^{(3)} = \begin{array}{c|ccccc} & 1 & 2 & 3^{\bullet} & 4 & 5 \\ \hline 1 & 1 & 11^{+} & 3 & 5^{+} & 9^{+} \\ 2 & 4 & 0 & 7 & 1 & 2 \\ {}^{\bullet}3 & 3 & 8 & 0 & 2 & 6 \\ 4 & 5 & 1 & 8 & 0 & 3 \\ 5 & 5 & 1 & 6 & 2 & 0 \end{array} \qquad P^{(3)} = \begin{array}{c|ccccc} & 1 & 2 & 3^{\bullet} & 4 & 5 \\ \hline 1 & — & 3^{+} & 1 & 3^{+} & 3^{+} \\ 2 & 2 & — & 1 & 2 & 2 \\ {}^{\bullet}3 & 3 & 3 & — & 3 & 3 \\ 4 & 2 & 4 & 1 & — & 2 \\ 5 & 2 & 5 & 5 & 2 & — \end{array}$$

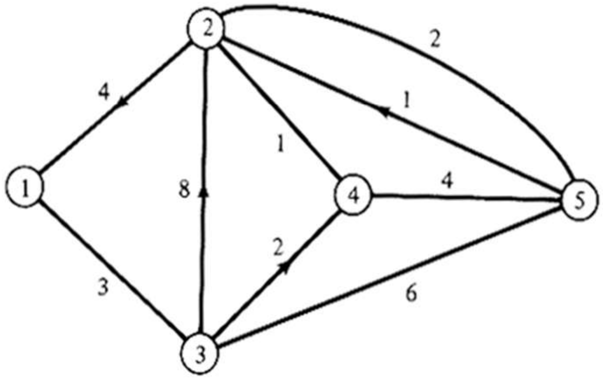


Shortest Path Problem (continued)

- Floyd Algorithm - Example (cont'd)

Through Node 4:

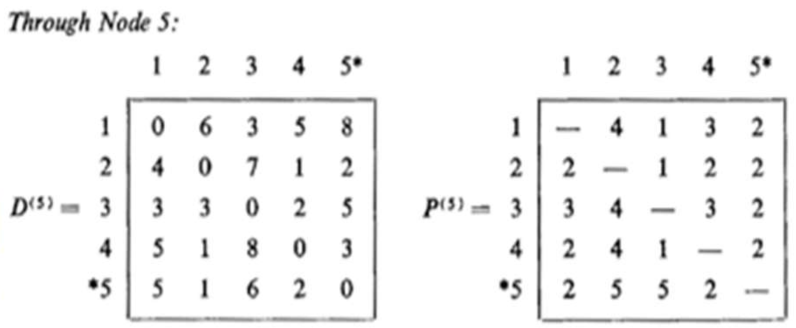
$$D^{(4)} = \begin{array}{c|ccccc} & 1 & 2 & 3 & 4^{\bullet} & 5 \\ \hline 1 & 0 & 6^{+} & 3 & 5 & 8^{+} \\ 2 & 4 & 0 & 7 & 1 & 2 \\ 3 & 3 & 3^{+} & 0 & 2 & 5^{+} \\ {}^{\bullet}4 & 5 & 1 & 8 & 0 & 3 \\ 5 & 5 & 1 & 6 & 2 & 0 \end{array} \qquad P^{(4)} = \begin{array}{c|ccccc} & 1 & 2 & 3 & 4^{\bullet} & 5 \\ \hline 1 & — & 4^{+} & 1 & 3 & 2^{+} \\ 2 & 2 & — & 1 & 2 & 2 \\ 3 & 3 & 4^{+} & — & 3 & 2^{+} \\ {}^{\bullet}4 & 2 & 4 & 1 & — & 2 \\ 5 & 2 & 5 & 5 & 3 & — \end{array}$$



Through Node 5:

$$D^{(5)} = \begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5^{\bullet} \\ \hline 1 & 0 & 6 & 3 & 5 & 8 \\ 2 & 4 & 0 & 7 & 1 & 2 \\ 3 & 3 & 3 & 0 & 2 & 5 \\ 4 & 5 & 1 & 8 & 0 & 3 \\ {}^{\bullet}5 & 5 & 1 & 6 & 2 & 0 \end{array} \qquad P^{(5)} = \begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5^{\bullet} \\ \hline 1 & — & 4 & 1 & 3 & 2 \\ 2 & 2 & — & 1 & 2 & 2 \\ 3 & 3 & 4 & — & 3 & 2 \\ 4 & 2 & 4 & 1 & — & 2 \\ {}^{\bullet}5 & 2 & 5 & 5 & 2 & — \end{array}$$

Shortest Path Proble



Through Node 5:

$$D^{(5)} = \begin{array}{c|ccccc|} & 1 & 2 & 3 & 4 & 5^{*} \\ \hline 1 & 0 & 6 & 3 & 5 & 8 \\ 2 & 4 & 0 & 7 & 1 & 2 \\ 3 & 3 & 3 & 0 & 2 & 5 \\ 4 & 5 & 1 & 8 & 0 & 3 \\ {}^{*}5 & 5 & 1 & 6 & 2 & 0 \\ \hline \end{array} \qquad P^{(5)} = \begin{array}{c|ccccc|} & 1 & 2 & 3 & 4 & 5^{*} \\ \hline 1 & - & 4 & 1 & 3 & 2 \\ 2 & 2 & - & 1 & 2 & 2 \\ 3 & 3 & 4 & - & 3 & 2 \\ 4 & 2 & 4 & 1 & - & 2 \\ {}^{*}5 & 2 & 5 & 5 & 2 & - \\ \hline \end{array}$$

Floyd Algorithm - Example (cont'd)

From D(5) and P(5),

- The shortest path from node 1 to node 5 has length d(1,5) = 8 units
- This shortest path is the path {1, 3, 4, 2, 5}.
- To identify that shortest path, we examined row 1 of the P(5) matrix
  - Entry p5 says that the predecessor node to 5 in the path from 1 to 5 is node 2;
  - then, entry p5(1, 2) says that the predecessor node to 2 in the path from 1 to 2 is node 4;
  - similarly, we backtrace the rest of the path by examining p5(1, 4) = 3 and p5(1, 3) = 1.
  - In general, backtracing stops when the predecessor node is the same as the initial node of the required path.

Shortest Path Problem (continued)

- Floyd Algorithm - Example (cont'd)

From D(5) and P(5),

- The shortest path from node 4 to node 3 is d(4, 3) = 8 units long
- The path is {4, 2, 1, 3}. The predecessor entries that must be read are, in order, p5(4, 3) = 1, p5(4, 1) = 2, and finally p5(4, 2) = 4 -- at which point we have "returned" to the initial node.

- Check Sections 6.2.3 and 6.2.4 for an application of Floyd Algorithm

Computational Complexity of Algorithms

- A question may arise as to which algorithm is more efficient to compute the shortest path for all pairs of nodes.
- Algorithm complexity is usually measured by the number of elementary operations required
  - Elementary equations include addition, subtraction, multiplication, division, comparison, branching
  - Complexity is measured by the elementary operations to reach the "worst-case conditions"

Computational Complexity of Algorithms (cont'd)

- For the Floyd algorithm, each iteration includes
  - $(n-1)^2$ additions - (Step 2)
  - $(n-1)^2$ comparisons and updates - (Step 2)
  - $(n-1)^2$ comparisons and updates of the predecessor matrix - (Step 3)
  - 3 operations before looping to the next iteration: check if the current iteration=n (number of nodes), update iteration counter and branching to Step 2

- With n iterations, the number of elementary iterations (T)
  - $T = n\ .\ [3(n-1)^2 + 3] = 3n^3 – 6n^2 + 6n$

Computational Complexity of Algorithms (cont'd)

- As ***n*** increases, T is determined largely by $3n^3$
  - $\therefore$ we say that the complexity of the Floyd algorithm is $O(n^3)$, or the algorithm requires $O(n^3)$ time
- We can show that using Dijkstra's algorithm to find the shortest path between all pairs is also $O(n^3)$ complex
  - From any *given* source node to all other nodes, it is $O(n^2)$
  - To find the shortest path between all nodes, repeat the algorithm n time.
  - Therefore, it is $O(n^3)$

Computational Complexity of Algorithms (cont'd)

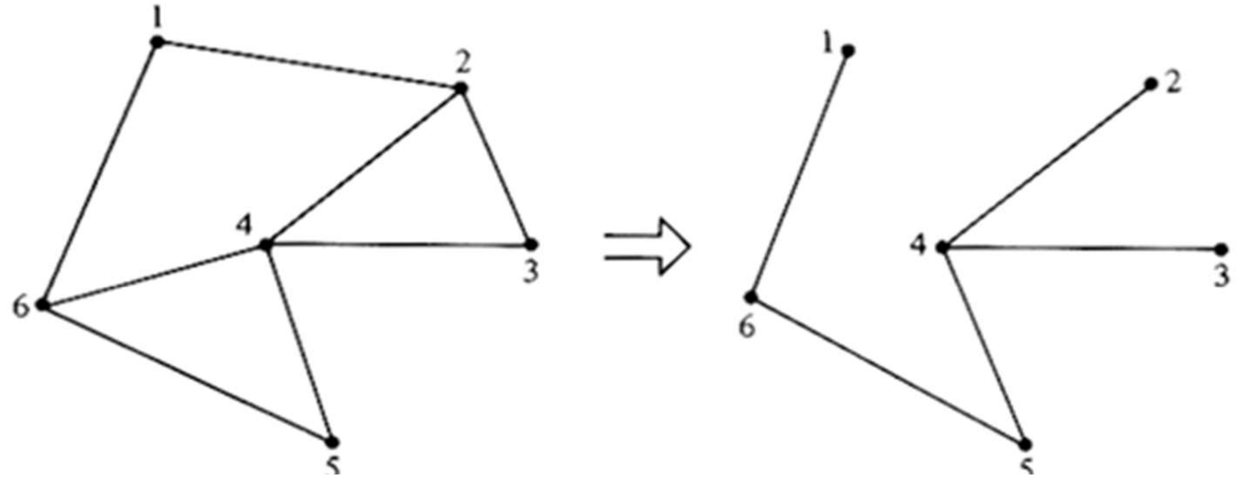
- The complexity of Floyd and Dijkstra Algorithms is the same, O(n3), under worst-case conditions.
  - Although it is true that Dijkstra Algorithm is more flexible and can thus better take advantage of special network structures
  - On the other hand, Floyd Algorithm is easier to program on a computer
- Both algorithms are considered *polynomial* (as apposed to *exponential* ones)
- Both algorithms are considered exact (as apposed to *heuristic* ones); that is, they are guaranteed to terminate given sufficient time with an optimal solution

Minimum Spanning Tree (MST) Problem

- Recall
  - A tree of an undirected graph is a connected subgraph with no cycles
  - A tree with *t* nodes has *t-1* arcs
  - A single path exists between any 2 nodes on a tree
  - A spanning tree of the graph *G(N,A)* has *n* nodes and *n-1* links
- The MST problem has direct applications in problems of transportation network design, and node-covering problems

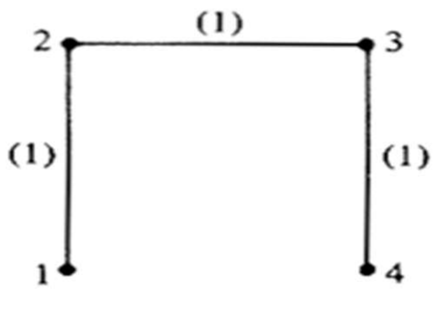
Minimum Spanning Tree (MST) Problem (cont'd)

- Consider the case in which map locations of n rural towns are given along with a matrix listing the Euclidean distances between all possible pairs of towns. The MST produces the minimum length of roads needed to connect, directly or indirectly, all pairs of towns

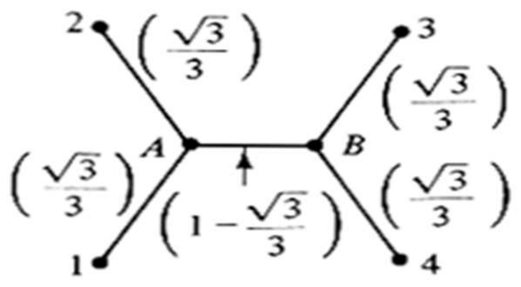


Minimum Spanning Tree (MST) Problem (cont'd)

- MST vs. "Steiner problem"
  - The latter has the same objective as the former but allows the introduction of artificial "nodes"
  - The total line length needed to connect the four points can be further reduced. The solution to Steiner's problem in this case involves creation of two artificial nodes, the total length of the Steiner tree is 1 + √3



(a)



2
3
1
4
A
B
$\left(\frac{\sqrt{3}}{3}\right)$
$\left(\frac{\sqrt{3}}{3}\right)$
$\left(\frac{\sqrt{3}}{3}\right)$
$\left(\frac{\sqrt{3}}{3}\right)$
$\left(1-\frac{\sqrt{3}}{3}\right)$

(b)

Minimum Spanning Tree (MST) Problem (cont'd)

- MST Problem for an Undirected Graph
  - Find a shortest-length spanning tree of *G(N,A)*
  - This problem arises when we are designing networks (e.g. transit network design)
  - Note that for a given graph *G(N,A)*, MST may not be unique (more on this later)

Minimum Spanning Tree (MST) Problem (cont'd)

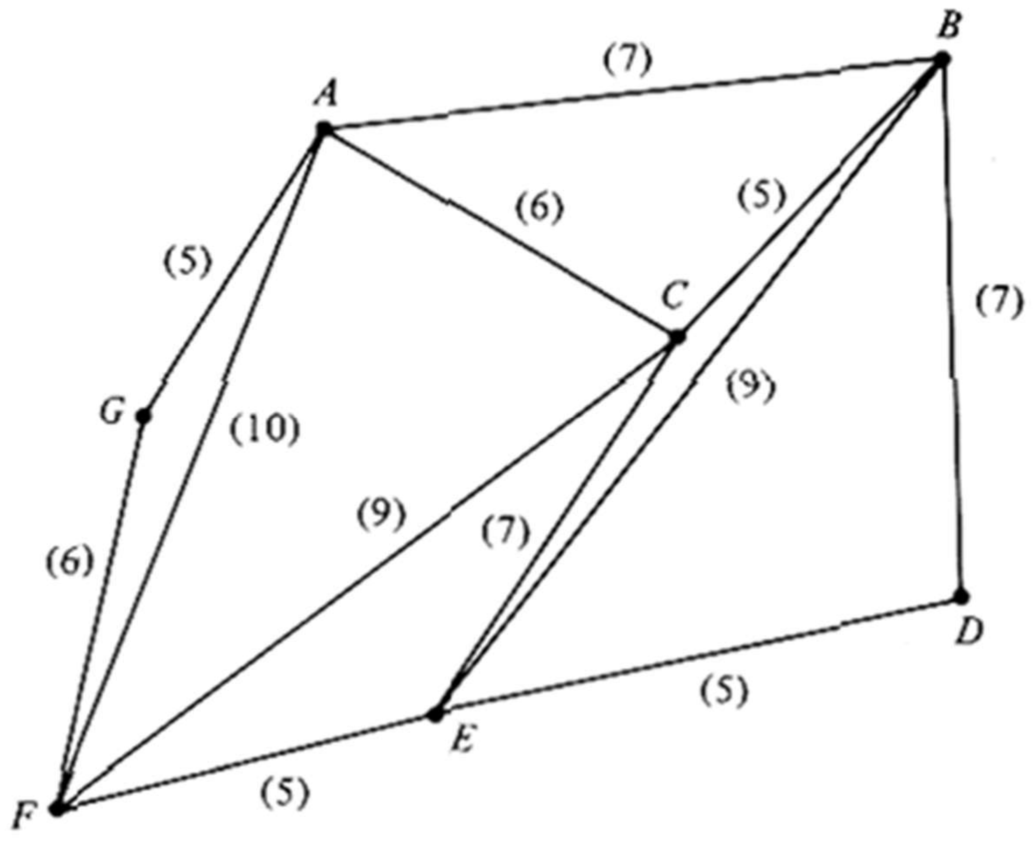
- Fundamental Property
  - The shortest link out of any sub-tree (during the construction of the MST) must be a part of the MST
  - There is no specific assumptions about the procedure for constructing the MST
    - Begin the procedure with ***n*** trees, with each tree consisting of a single isolated node and no links
    - Connect any arbitrarily chosen tree (node) to its nearest tree (node) and continue this procedure until all nodes are finally connected
- Corollary
  - In an undirected network G, the link of the shortest length out of any node is part of the MST

Minimum Spanning Tree (MST) Problem (cont’d)

- MST Algorithm (section 6.3.1)
  - **STEP 1**
    - Begin at some arbitrary node, say node i. Find the node closest to i, say j, and connect it to i
    - Break ties, if any, arbitrarily
  - **STEP 2**
    - If all nodes have been connected, stop
    - If there are still isolated nodes, go to Step 3.
  - **STEP 3**
    - Find the one among the still isolated nodes which is closest to the already connected nodes and connect it with the already connected nodes
    - Break ties, if any, arbitrarily. Return to Step 2

Minimum Spanning Tree (MST) Problem (cont'd)

- MST Example
  - Find the MST for this network, starting at node A



Minimum Spanning Tree (MST) Problem (cont'd)

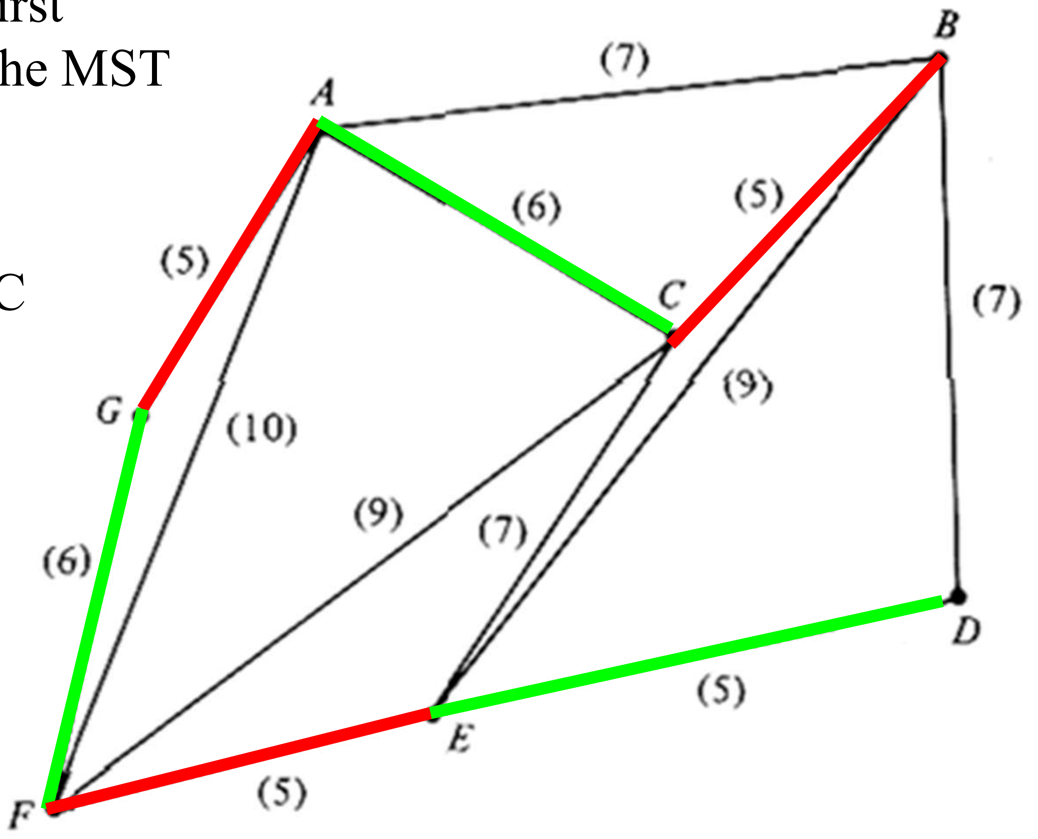
- MST Example

1) G is closest to A, G and the link (A, G) first become part of the MST

2) Break the tie between F and C
Choose C

3) Choose B, with link (C,B)

4) Choose F, with link (G,F)



5) Choose E, with link (F,E)

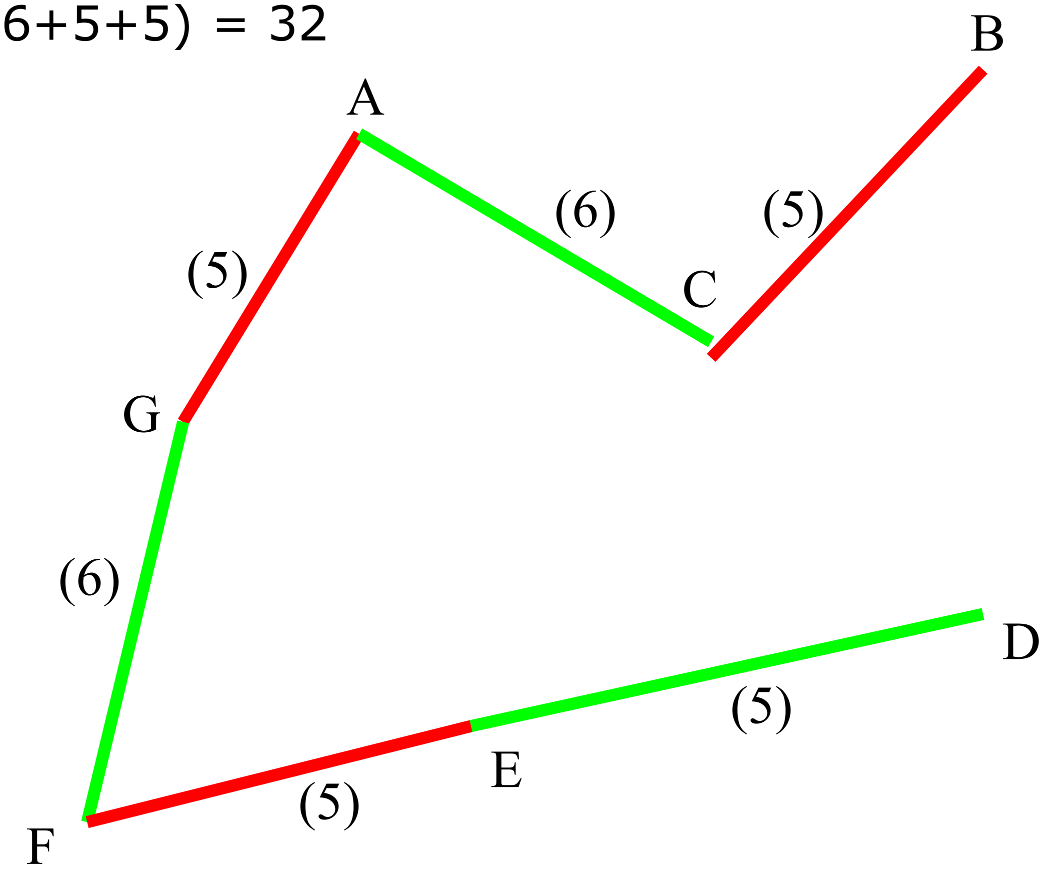
6) Choose D, with link (E,D)

STOP
No nodes left

Minimum Spanning Tree (MST) Problem (cont'd)

- MST Example

Total length for the MST
(5+6+5+6+5+5) = 32



If you choose G, you will get the same MST --

Try it again with link (C,F)= 5

Routing Problems…

- Design of routes for vehicles or people. These routes must be designed so that they traverse in an exhaustive way the streets in a neighborhood
  - called *edge-covering* problem
  - e.g. cleaning and sweeping of streets
  - The Chinese Postman's Problem

- Alternatively, the objective may be to visit a set of given geographical points in a city in order to provide some service there or for delivery purposes
  - called *node-covering* problem
  - e.g. the distribution of newspapers to newsstands
  - The Travelling Salesman Problem