Definitions

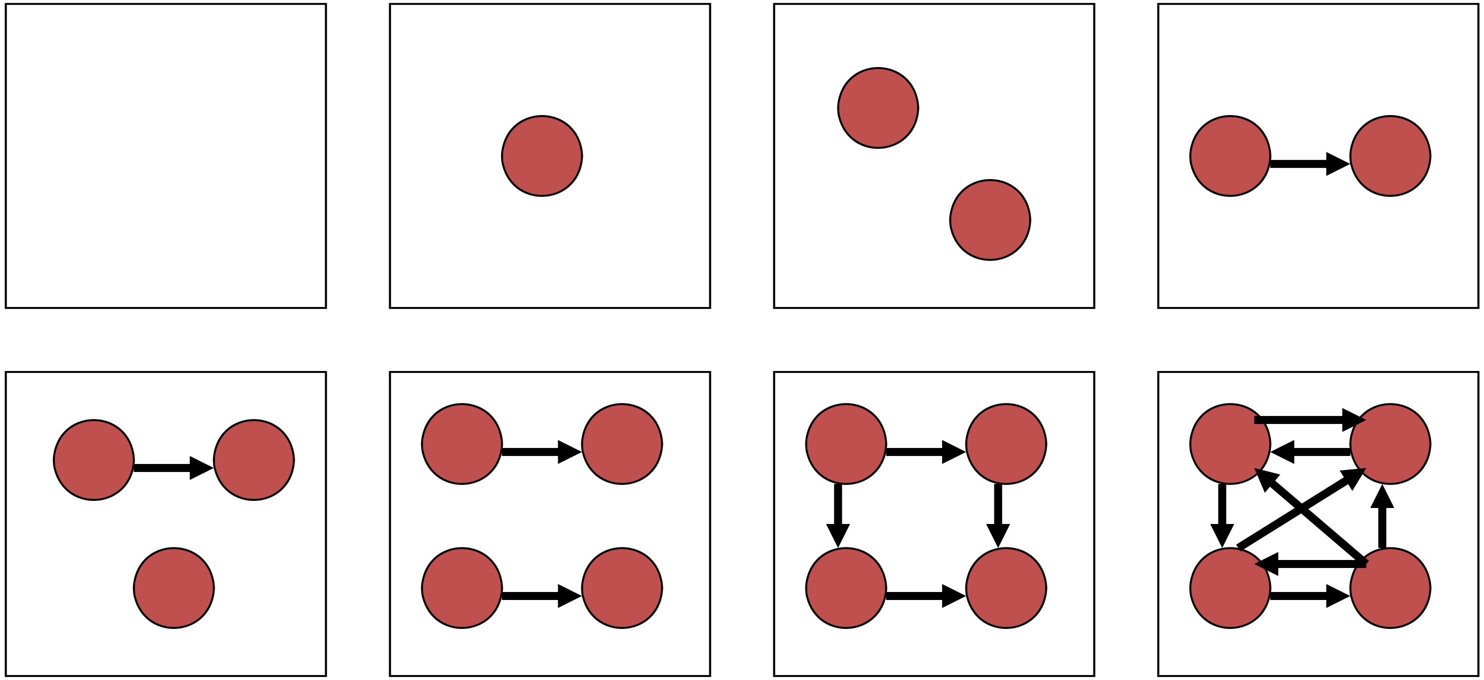
G = (V, E) V = set of vertices (vertex / node)
E = set of edges (v, w) (v, w in V)

(v, w) ordered => directed graph (digraph)
(v, w) non-ordered => undirected graph

digraph:
w is **adjacent** to v if there is an edge from v to w

edge may be (v, w, c) where c is a cost component
(e.g. distance)

Examples

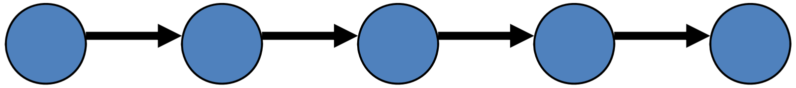


Meaning & Use

- A graph is used to represent arbitrary relationships among data objects
- e.g. undirected graphs
  - communications network
  - transport network (road, rail, air, sea) with costs/distances
  - (travelling salesman problem)
- e.g. directed graphs (digraph)
  - flow of control in computer programs
  - University course planning (dependency graph)
  - state transition diagrams

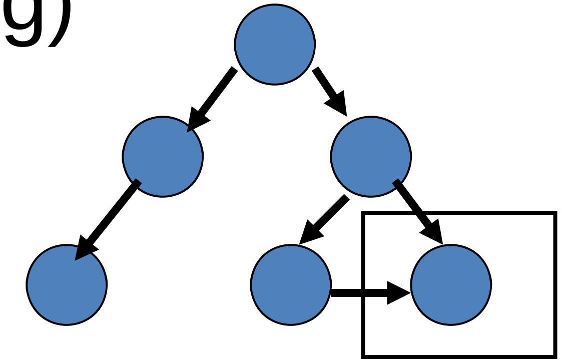
Other ADTs

linked list

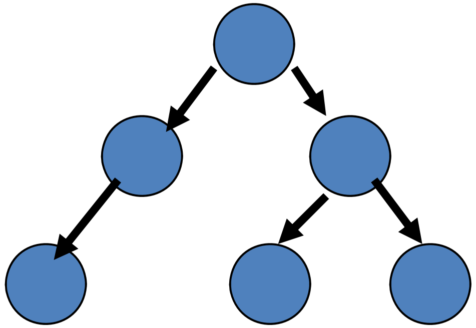


directed acyclic graph (dag)

(dag)



tree



Terminology

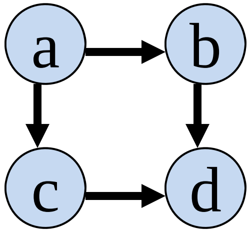
PATH: a sequence of vertices v_1, v_2, …v_n such that v_1 -> v_2, v_2 -> v_3, … v_{n-1} -> v_n are edges

LENGTH:	number of edges in a path (v denotes a path length 0 from v to v)
SIMPLE PATH:	**all vertices are distinct** (except possibly the first and the last)
SIMPLE CYCLE:	simple path of length >= 1 that begins (directed graph) and ends at the same vertex

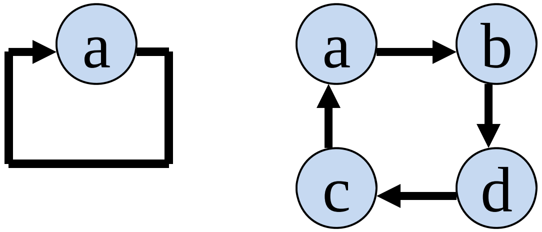
Graphs & Cycles

- A cycle is a path which begins and ends at the same vertex
- A graph with no cycles is **acyclic**
- A directed graph with no cycles is a directed acyclic graph (DAG)

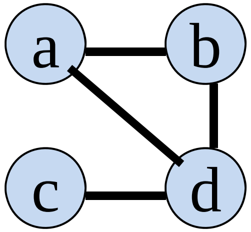
DAG



directed graphs



undirected graphs

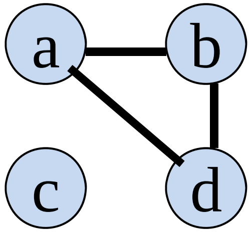


Undirected Graphs

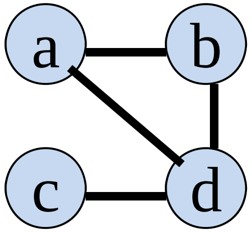
For cycles in undirected graphs, the edges must be distinct since (u,v) and (v, u) are the same edge

connected: if there exists a path from every vertex to every other vertex

non-connected:



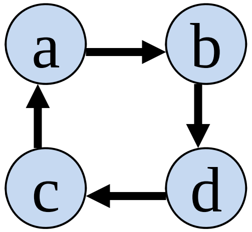
connected:



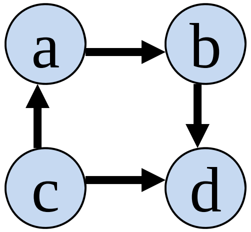
Directed Graphs

- A connected directed graph is called **strongly connected** i.e. there is a path from every vertex to every other vertex
- if the digraph is not strongly connected BUT the underlying graph, without distinction to the direction, is connected, then the graph is said to be **weakly connected**

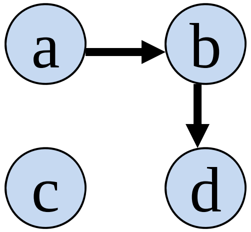
strong:



weak:

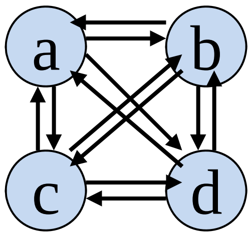


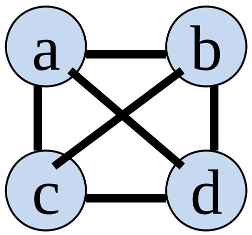
not:



Complete Graph

A graph is **complete** if there is an edge between every pair of vertices





12 edges

n * (n - 1)

6 edges

n * (n - 1) / 2

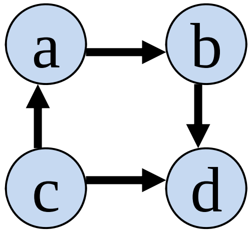
Adjacency Matrix

For each edge (u, v) set a[u, v] = 1

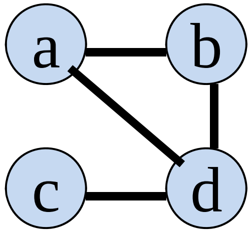
storage => omega(n^2)

read in/

search => O(n^2)



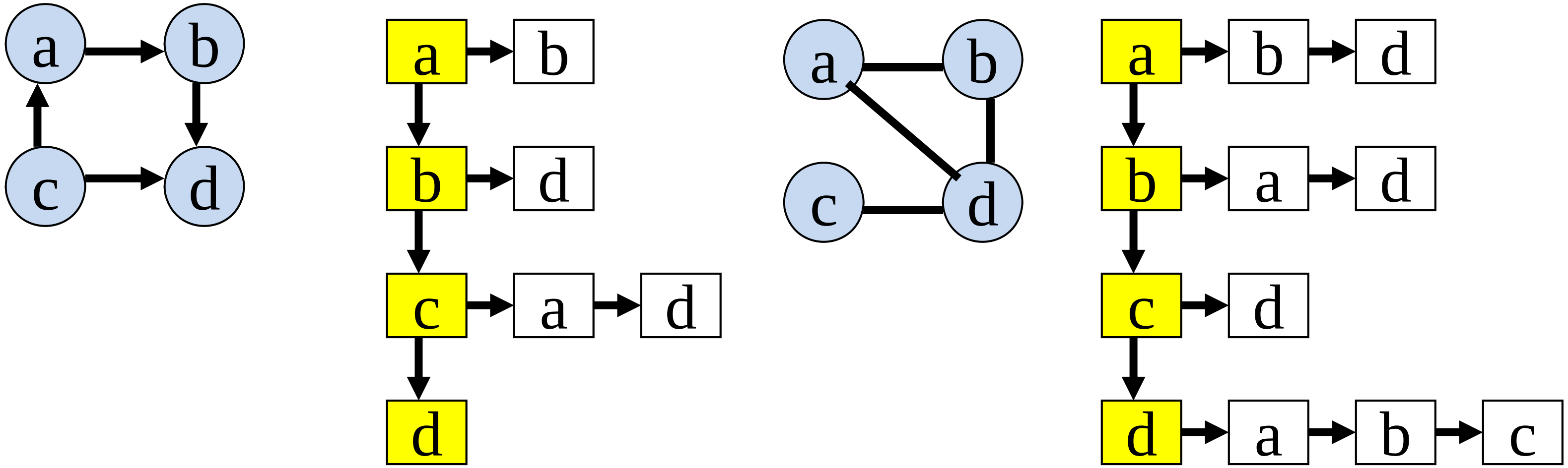
	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	1	0	0	1
d	0	0	0	0



	a	b	c	d
a	0	1	0	1
b	1	0	0	1
c	0	0	0	1
d	1	1	1	0

Adjacency List

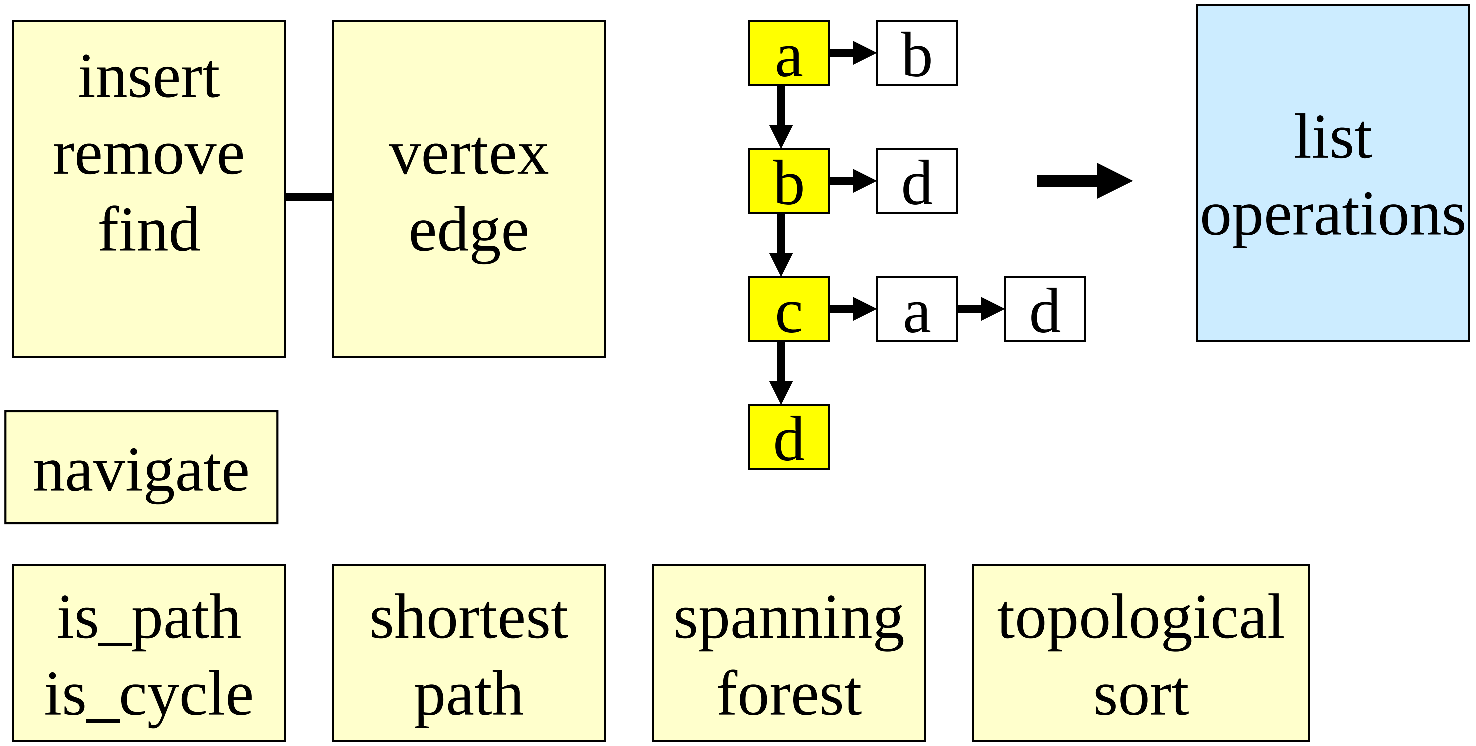
- Use a list of nodes where each node points to a list of **adjacent** nodes (better for sparse graphs)



space = O(|V| + |E|)

(for named vertices - use a hash table)

Operations



insert
remove
find

vertex
edge

a → b
b → d
c → a → d
d

list
operations

navigate

is_path
is_cycle

shortest
path

spanning
forest

topological
sort

Shortest Path 1 **Dijkstra**'s algorithm

Single source shortest path (non-negative costs)
Determines the **shortest path** from **a source to every other vertex** in the graph where the **length** of the path is the sum of the costs of the edges

S - set of vertices; shortest distance from source **already known**

- **each step adds a vertex v whose distance from S is as short as possible** – the visited vertices (nodes)

special path: **shortest path from the source to v passing through u**
array D: length of **shortest special path** to each vertex
C[i,j]: cost of v_i to v_j (no edge ➔ cost is infinite §)

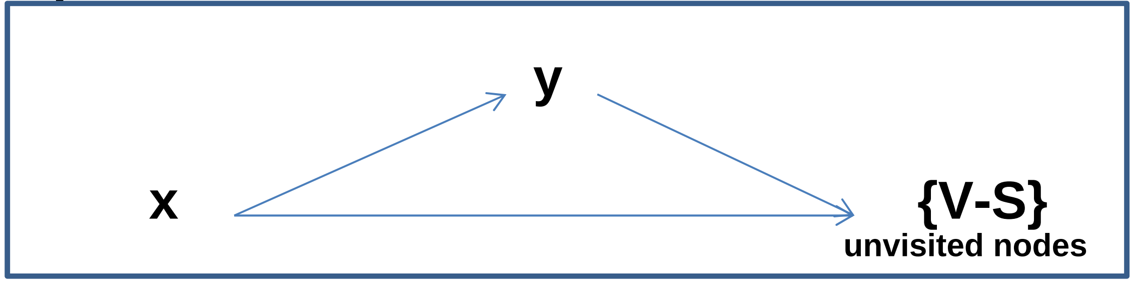
Dijkstra’s algorithm principles

Given a start node x, note the edge lengths from x to the remaining nodes in the graph. Choose the shortest edge from x to a node y. Mark nodes x and y as visited. S = {x,y}

Check to see if there is a shorter path to the remaining (unvisited) nodes, {V-S}, in the graph from x via y.

If so, update the path lengths so far calculated.

Repeat the process until all nodes have been visited.



Dijkstra - Example

(a b 10) (a d 30) (a e 100) (b c 50) (c e 10) (d c 20) (d e 60)

Start **a** – visited {a}, unvisited {b, c, d, e}, **shortest path (a b 10)**

§ = infinity   b c d e

Visited {a, **b**}, **unvisited {c, d, e}**   **D = [10, §, 30, 100]**

(**a-b-c 60**) (**a-b**-d §) (**a-b**-e §)   **D = [10, 60, 30, 100]**

Shortest **path (a-d 30)** – visited {a, b, **d**}, **unvisited {c, e}**

(**a-d-c 50**) (**a-d-e 90**)   **D = [10, 50, 30, 90]**

Shortest **path (a-c 50)** – visited {a, b, **c**, d}, **unvisited {e}**

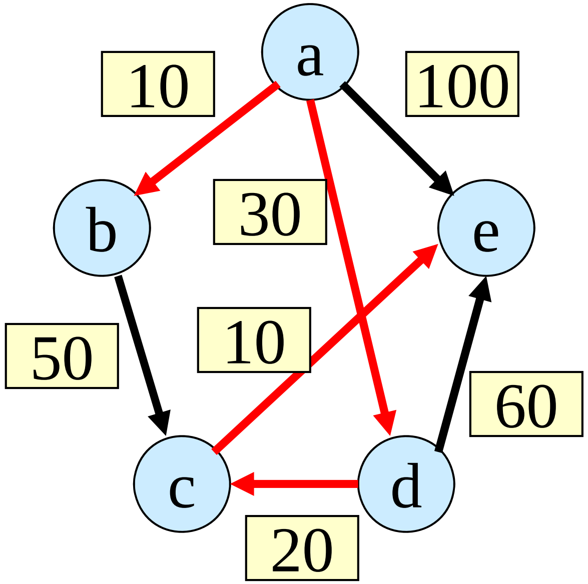
(**a-c-e 60**)   **D = [10, 50, 30, 60]**

Shortest **path (a-e 60)** – visited {a, b, c, d, **e**}, **unvisited { }**

No nodes left!   Final answer:-   **D = [10, 50, 30, 60]**

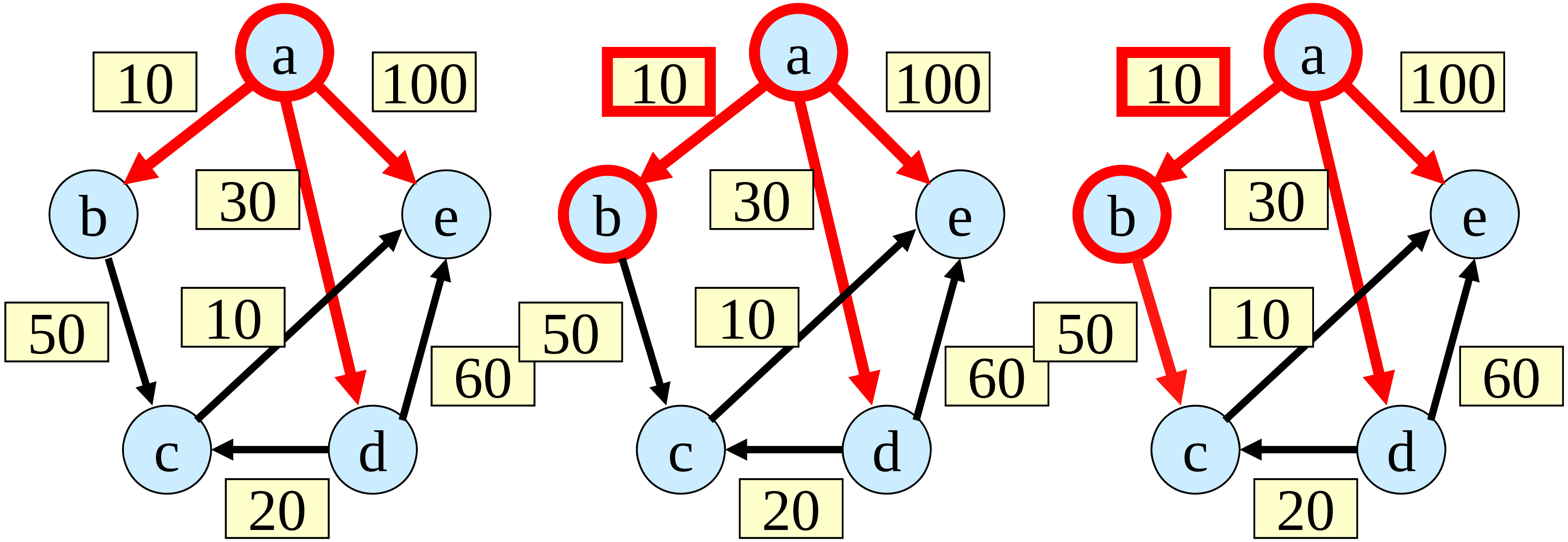
Dijkstra - Example

Iteration	S	w	D[b]	D[c]	D[d]	D[e]
initial	**{a}**	**-**	**10**	**§**	**30**	**100**
1	**{a,b}**	**b**	**10**	**60**	**30**	**100**
2	**{a,b,d}**	**d**	**10**	**50**	**30**	**90**
3	**{a,b,d,c}**	**c**	**10**	**50**	**30**	**60**
4	**{a,b,d,c,e}**	**e**	**10**	**50**	**30**	**60**



See separate notes on a worked example:-
(i) Revision notes (ii) study plan

Dijkstra – Example - pictures

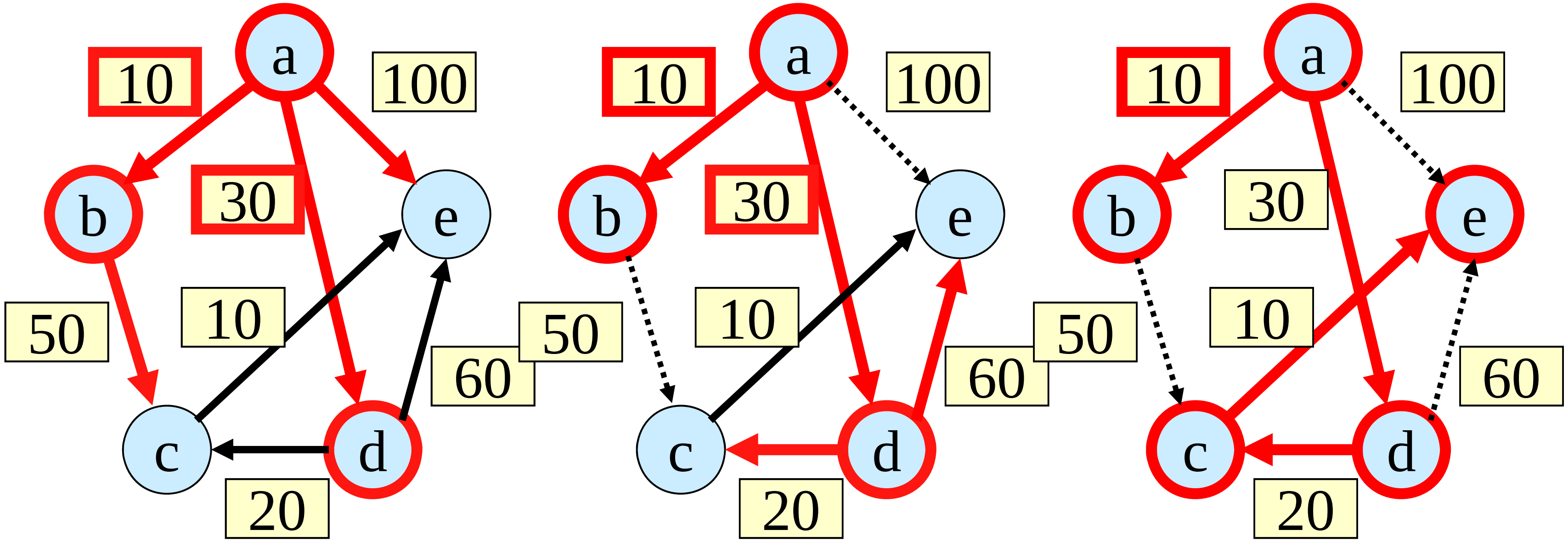


	b	c	d	e
D	**10**	**§**	**30**	**100**
E	**a**	**a**	**a**	**a**
L	**10**	**§**	**30**	**100**

	b	c	d	e
D	**10**	**§**	**30**	**100**
E	**a**	**a**	**a**	**a**
L	**10**	**§**	**30**	**100**

	b	c	d	e
D	**10**	**60**	**30**	**100**
E	**a**	**b**	**a**	**a**
L	**10**	**50**	**30**	**100**

Dijkstra – Example - pictures



b c **d** e

D [**10**, 60, **30**, 100]

E [**a**, b, **a**, **a**]

L [**10**, 50, 30, **100**]

b c **d** e

D [**10**, **50**, **30**, **90**]

E [**a**, **d**, **a**, **d**]

L [**10**, **20**, **30**, **60**]

b **c** **d** e

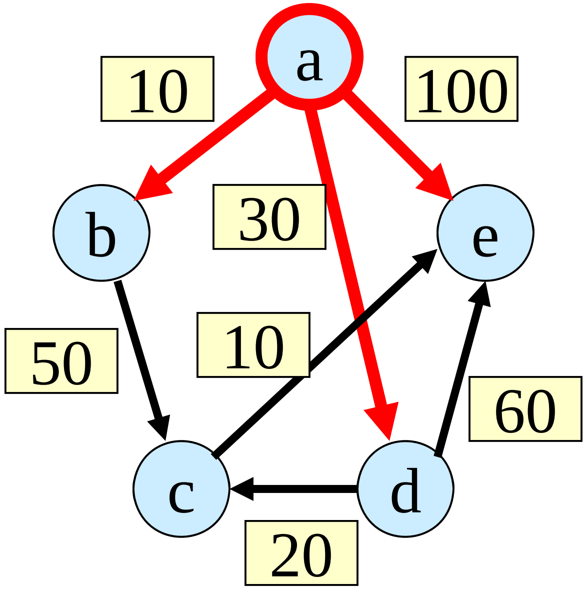
D [**10**, **50**, **30**, **60**]

E [**a**, **d**, **a**, **c**]

L [**10**, **20**, **30**, **10**]

Dijkstra’s Algorithm

- Graph (G) + Cost Matrix (C)



	a	b	c	d	e
a		10		30	100
b			50		
c					10
d			20		60
e					

- **NB count the number of edges in the <u>graph</u> and the <u>cost matrix</u>**

Dijkstra's Algorithm

```
Dijkstra (a)
{   S = {a}                                      // G = (V, E)

    for ( i in V-S) D[i] = C[a, i]               // initialisation

    while (!is_empty(V-S)) {
        choose w in V-S such that D[w] is a minimum

        S = S + {w}
        foreach ( v in V-S) D[v] = min(D[v], D[w]+C[w,v])

        }                                        // process
    } // D[i] = distance; C[i,j] = cost matrix; S = {visited nodes}
```

Dijkstra's Algorithm

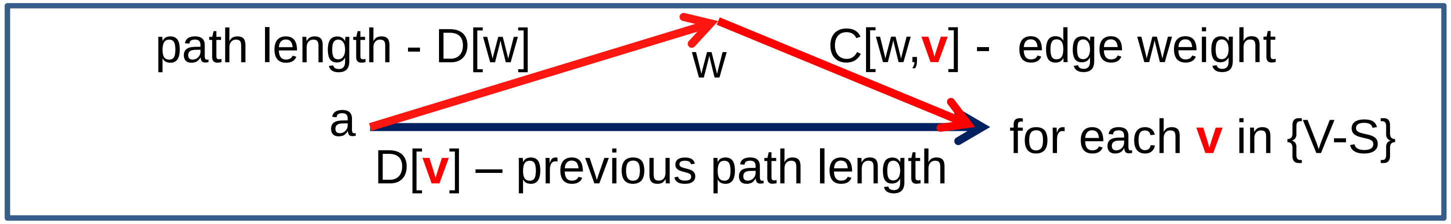
```
Dijkstra (a)                -- a is the start node
{    S = {a}                -- S represents the nodes visited

     for ( i in V-S) D[i] = C[a, i]   -- initialise D (path lengths)
                                      -- from the start node a
     while (!is_empty(V-S)) {
          choose w in V-S such that D[w] is a minimum
          -- unvisited node with shortest path from start_node
          S = S + {w}  -- add this node to visited nodes
          foreach ( v in V-S) D[v] = min(D[v], D[w]+C[w,v])
          -- recalculate paths via w to unvisited nodes
          }
     }
```

Dijkstra - Comment

- "**greedy**" algorithm - **local best solution** is best overall
- choose **w in V-S** such that **D[w] is a minimum** (meaning?)
- recall that **D[i]** means the **length** of the **shortest** **path** to each vertex
- note that the algorithm partitions the nodes into **two spaces** **S** (initially with the start node) and **V-S** (the remaining nodes) **visited / unvisited**
- **foreach (v in V-S) D[v] = min(D[v], D[w]+C[w,v])** (meaning?)
  - **w** is the node with the minimum distance from the source (not in S)
  - **D[v]** means the **shortest (special) path length** **so far calculated**
  - **D[w]** is the cost (so far) to w (again a shortest (special) path length)
  - **C[w,v]** is the cost from node w to node v (**edge cost**)

Dijkstra – principles revisited

- Choose the start node – **a** **S = {a}** **G = (V, E)**
- **S** represents visited nodes; **V-S** represents unvisited nodes
- **D** represents the path lengths from **a** to the remaining nodes **(V-a)**
- **C[x,y]** represents the cost matrix – the cost of the **edge** x→y
- **Algorithm**
- Choose the shortest **path** to an unvisited node **(may be an edge)**
- Add this node (**w**) to the set of visited nodes **S**
- Calculate an **alternative path via w** to all nodes **v** in **{V-S}**
- choose if distance **D[w]+C[w,v]** is shorter than **D[v]**



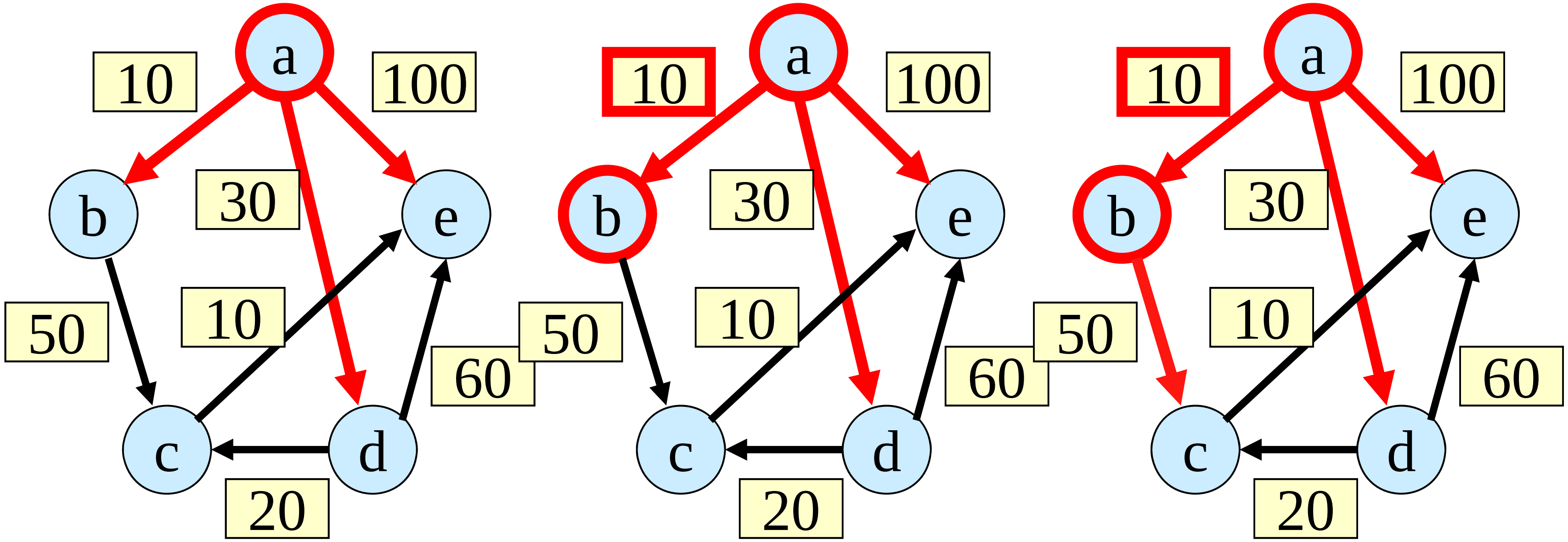
Dijkstra’s Algorithm + Shortest Path Tree

```
Dijkstra (a)
 {   S = {a}

     for ( i in V-S) { D[i] = C[a, i];  E[i] = a; L[i] = C[a,i]; }

     while (!is_empty(V-S)) {
          choose w in V-S such that D[w] is a minimum

          S = S + {w}
          foreach ( v in V-S) if (D[w]+C[w,v])< D[v] )
             { D[v] = D[w]+C[w,v]; E[v] = w; L[v] = C[w,v]; }
          }
     }
```

Dijkstra's Algorithm + Shortest Path Tree

```
Dijkstra (a)                -- a is the start node
{    S = {a}                -- S represents the nodes visited

     for ( i in V-S) { D[i] = C[a, i];  E[i] = a; L[i] = C[a,i]; }
      -- initialise D + SPT (E + L)
     while (!is_empty(V-S)) {
          choose w in V-S such that D[w] is a minimum
           -- unvisited node with shortest path from start_node
          S = S + {w}
          foreach ( v in V-S) if (D[w]+C[w,v])< D[v] )
            { D[v] = D[w]+C[w,v]; E[v] = w; L[v] = C[w,v]; }
          -- recalculate paths and SPT (E + L)
          }
     }
```

Dijkstra – Example - pictures

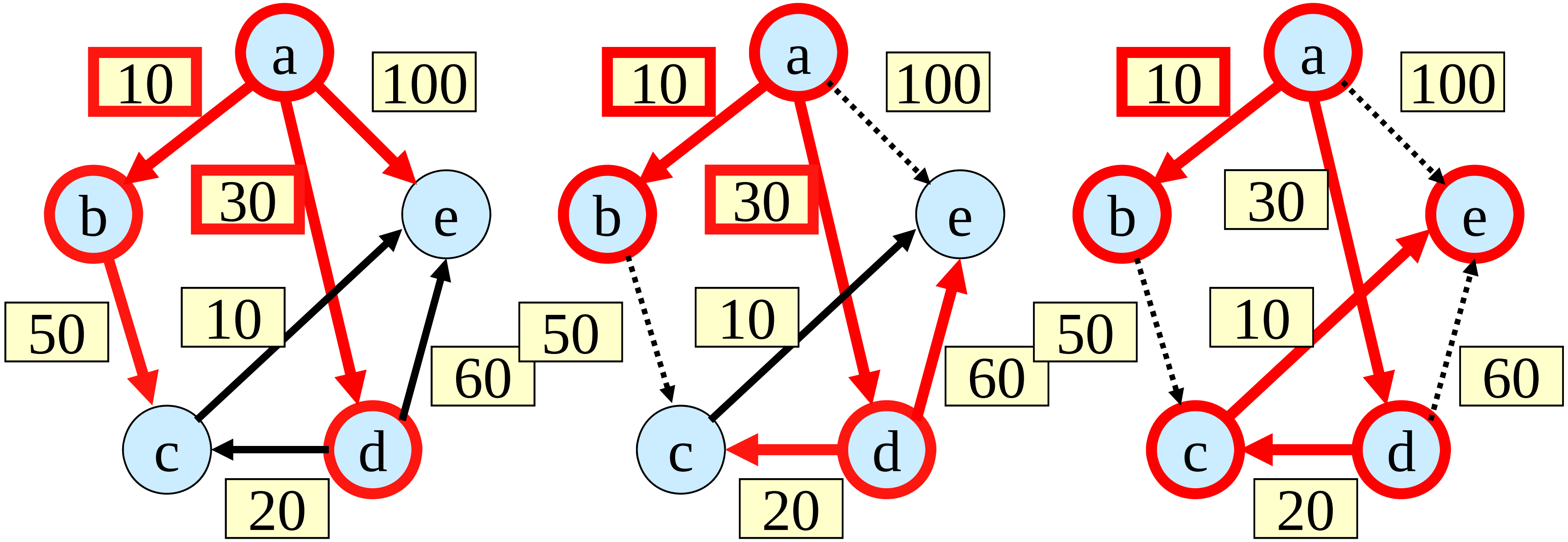


	b	c	d	e
D	**10**	**§**	**30**	**100**
E	**a**	**a**	**a**	**a**
L	**10**	**§**	**30**	**100**

	b	c	d	e
D	**10**	**§**	**30**	**100**
E	**a**	**a**	**a**	**a**
L	**10**	**§**	**30**	**100**

	b	c	d	e
D	**10**	**60**	**30**	**100**
E	**a**	**b**	**a**	**a**
L	**10**	**50**	**30**	**100**

Dijkstra – Example - pictures



	b	c	**d**	e
D	**10**	**60**	**30**	**100**
E	**a**	**b**	**a**	**a**
L	**10**	**50**	**30**	**100**

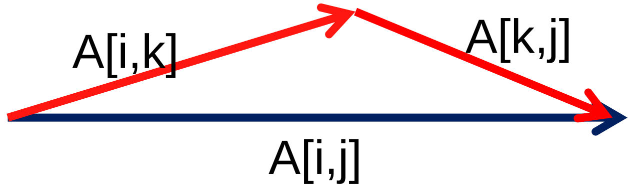
	b	c	**d**	e
D	**10**	**50**	**30**	**90**
E	**a**	**d**	**a**	**d**
L	**10**	**20**	**30**	**60**

	b	**c**	**d**	e
D	**10**	**50**	**30**	**60**
E	**a**	**d**	**a**	**c**
L	**10**	**20**	**30**	**10**

Shortest Path 2

- **All pairs shortest path** problem (i.e. Shortest path between any two vertices)
  - apply **Dijkstra's** algorithm to each node in turn
  - apply **Floyd's** algorithm
- **Floyd**
  - given G = (V,E), non-negative costs **C[v,w]**, for each ordered pair (v,w) find the shortest path
  - note the initial conditions
    - use an array **A[i,j]** which is initialised to **C[i,j]**, i.e. the initial edge costs
  - if no edge exists **C[i,j]=§** (infinite cost)
  - for n vertices there are n iterations over the array A
  - **Floyd is thus O(n^3)**

Floyd’s Algorithm

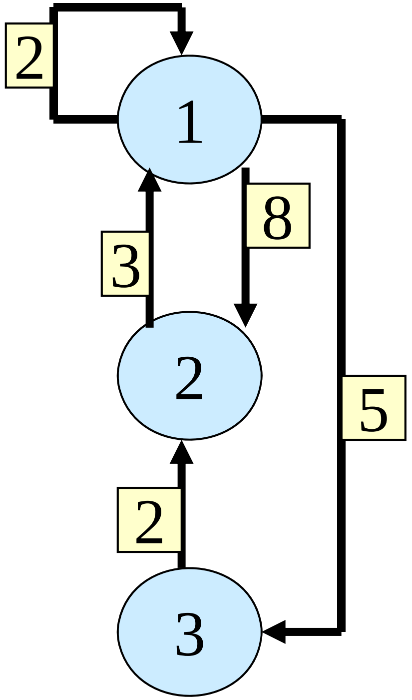


```
Floyd ( )
{
    for (i in 1..n) for (j in 1..n) if (i <> j) A[i, j] = C[i, j]
    for (i in 1..n) A[i, i] = 0                -- initialisation

    for (k in 1..n) for (i in 1..n) for (j in 1..n)
        if ( A[i, k] + A[k, j] < A[i, j])  A[i, j] = A[i, k] + A[k, j]
    }
```

Floyd - Example

$$A_0[i,j] = \begin{matrix} 0 & 8 & \boxed{5} \\ \boxed{3} & 0 & \S \\ \S & 2 & 0 \end{matrix}$$

$$A_1[i,j] = \begin{matrix} 0 & 8 & 5 \\ \boxed{3} & 0 & \boxed{8} \\ \S & \boxed{2} & 0 \end{matrix}$$



$$A_2[i,j] = \begin{matrix} 0 & 8 & \boxed{5} \\ 3 & 0 & 8 \\ \boxed{5} & \boxed{2} & 0 \end{matrix}$$

$$A_3[i,j] = \begin{matrix} 0 & \boxed{7} & 5 \\ 3 & 0 & 8 \\ 5 & 2 & 0 \end{matrix}$$

§ = infinity

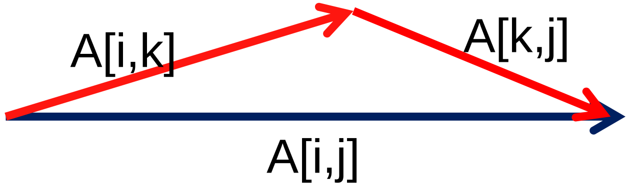
Floyd - Comment

- Initialisation is the costs in C (i.e. Initial edge costs) with the diagonal (i.e. v => v) set to 0
- for each node (k = 1..n) go through the array (i, j = 1..n) and compute costs - i.e. check if there is a **cheaper path** from node i to node j via node k - if so change A[i, j]
- **if (A[i, k] + A[k, j] < A[i, j]) A[i, j] = A[i, k] + A[k, j]**

Transitive Closure & Warshall's Algorithm

- Determine if a **path** exists from vertex i to vertex j
- C[i, j] = 1 if an **edge** exists (i <> j), otherwise = 0
- compute A[i, j], such that A[i, j] = 1 if there exists a **path** of length 1 or more from vertex i to vertex j
- A is called the **transitive closure** of the adjacency matrix
- Note that this is a special case of Floyd's where we are not directly interested in the costs

Warshall’s Algorithm



```
Warshall ( )
{
    for (i in 1..n) for (j in 1..n) A[i, j] = C[i, j]
    for (i in 1..n) A[i, i] = 0            -- initialisation

    for (k in 1..n) for (i in 1..n) for (j in 1..n)
        if (A[i, j] = 0) A[i, j] = A[i, k] and A[k, j]
}
```

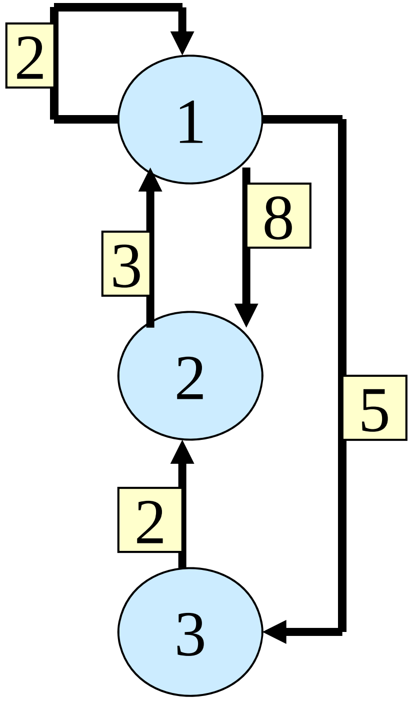
Warshall - Example

$$A_0[i,j] = \begin{matrix} 0 & \boxed{1} & \boxed{1} \\ \boxed{1} & 0 & 0 \\ 0 & 1 & 0 \end{matrix}$$

$$A_1[i,j] = \begin{matrix} 0 & 1 & \boxed{1} \\ \boxed{1} & \boxed{1} & \boxed{1} \\ 0 & \boxed{1} & 0 \end{matrix}$$

$$A_2[i,j] = \begin{matrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ \boxed{1} & 1 & \boxed{1} \end{matrix}$$

$$A_3[i,j] = \begin{matrix} \boxed{1} & 1 & 1 \\ 1 & \boxed{1} & 1 \\ 1 & 1 & \boxed{1} \end{matrix}$$



Warshall - Comment

- **if (A[i, j] = 0) A[i, j] = A[i, k] and A[k, j]** - (meaning ?)
- i.e. there is a path from **node i** to **node j** **IF** there is a path from **node i** to **node k** **AND** a path from **node k** to **node j**
- at various stages in the calculation for k, i, j, the different paths are discovered
  - **(1, 2, 2) - A[2,2] =A[2,1] and A[1,2] - i.e. 2 to 1 to 2 => 2 to 2**
  - **(1, 2, 3) - A[2,3] =A[2,1] and A[1,3] - i.e. 2 to 1 to 3 => 2 to 3**
  - **(2, 3, 1) - A[3,1] =A[3,2] and A[2,1] - i.e. 3 to 2 to 1 => 3 to 1**
  - **(2, 3, 3) - A[3,3] =A[3,2] and A[2,3] - i.e. 3 to 2 to 3 => 3 to 3**

Summary – directed graphs

- Definitions & implementations
- Algorithms

○ Dijkstra	**single node shortest path**
○ Dijkstra-SPT	**+ shortest path tree**
○ Floyd	**all pairs shortest path**
○ Warshall	**transitive closure** **is there a path from a to b ?**