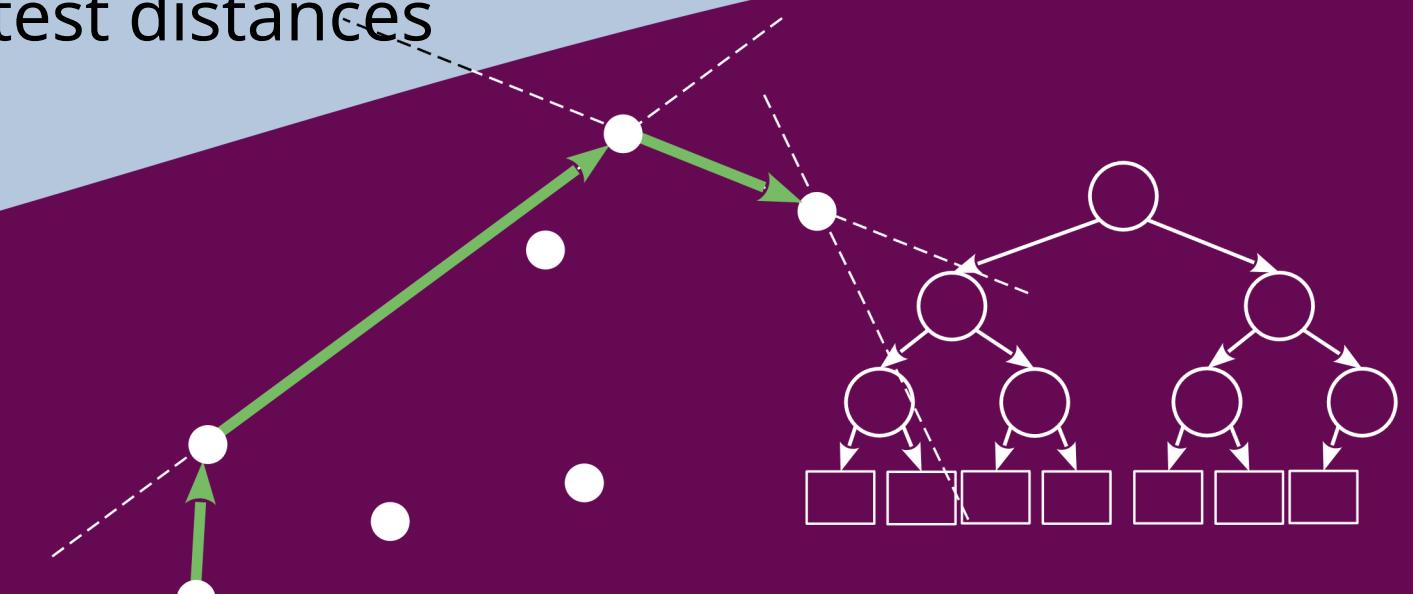
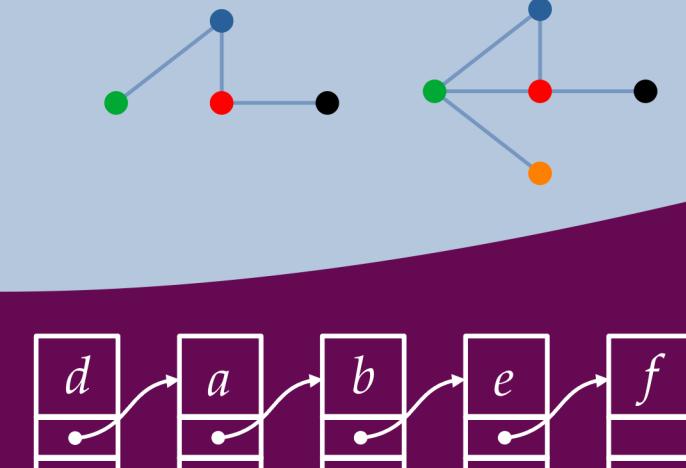


Advanced algorithms and structures data

Week 9:Algorithms over graphs
Graph theory, Shortest distances



Basics of graph theory

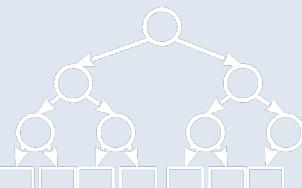
- **Graph**(graph) is an ordered pair non-empty of the final set of vertices (vertex) and a set of (possibly empty) edges (edge)
 $= (V, E)$
- **Brid**(edge) is an element of the product of the set
 $\subseteq \times, != (", #) \in$
 - Each edge is an ordered pair of vertices $!= (" , #)$
 - Brid can only be marked shorter " # or simply just !, where it is $!= " #$

Alternative literature:

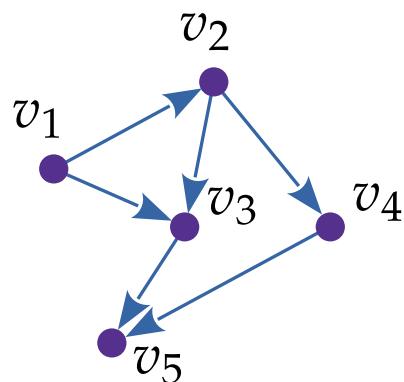
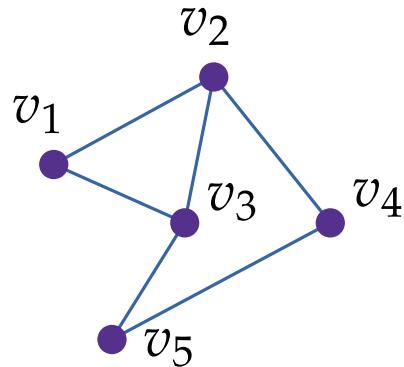
- Thomas H. C., Charles E. L., Ronald L. R., & Clifford, S. (2016). *Introduction to Algorithms.*, Chapter VI

Basics of graph theory

- **Order**_(order)**graph** is the number of vertices in it or $|V|$; often simplified only
 - **Size**_(size)**graph** is the number of edges that the graph contains $|E|$; simplified only
 - **Loop**_(loop) is an edge that starts and ends at the same vertex != "
 - **Degree**_(degree)**top** is the number of adjacent edges_(incident) that top (merged with it)
 - Peak degree we will denote $sdeg(V)$
 - Return loops are $\text{indeg}(V)$ they count twice
- $$\forall V \in \exists S \subseteq V, \#(V) = \sum_{v \in S} \text{deg}(v) = |S|$$



Graph theory - types of graphs

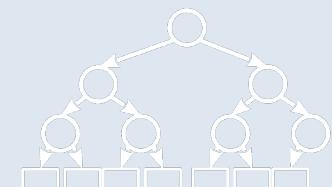


- **Undirected(undirected)graph**

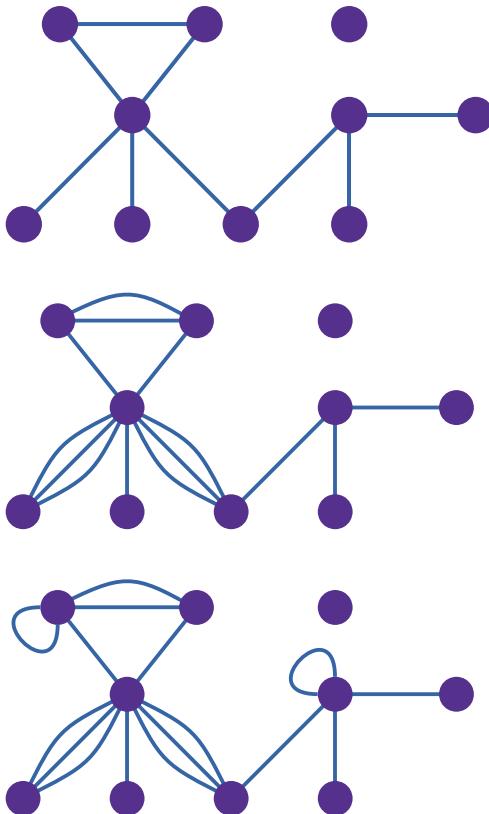
- For each edge " #it's worth it " $\# = \#$ "
- Vertices are considered adjacent(adjacent)if it is " $\#\in$
- Such is the lying edge(incident)peaks " $\#$

- **Directed(directed)graph**

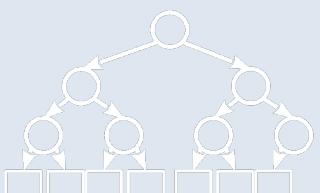
- For the edge " #it doesn't have to be valid " $\# = \#$ "
- Top $\#$ is considered a neighbor to the peak "only if it exists " $\#\in$
- Brid " $\#$ is called the output edge(outrudege)top "and the entrance edge(in edge)top #



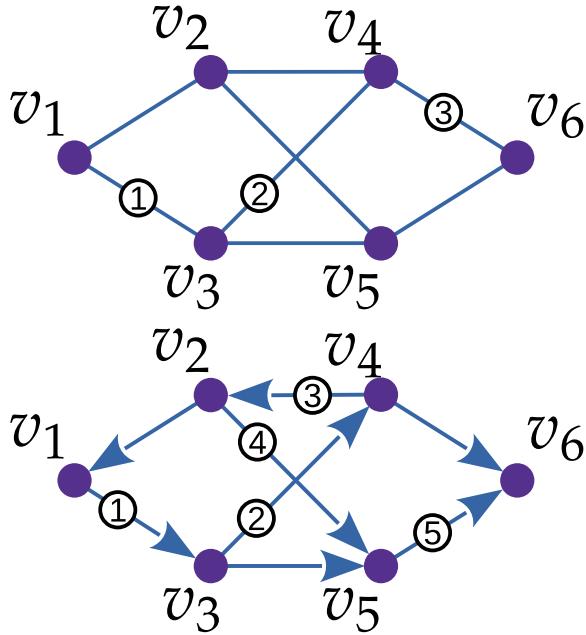
Graph theory - types of graphs



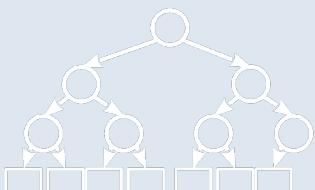
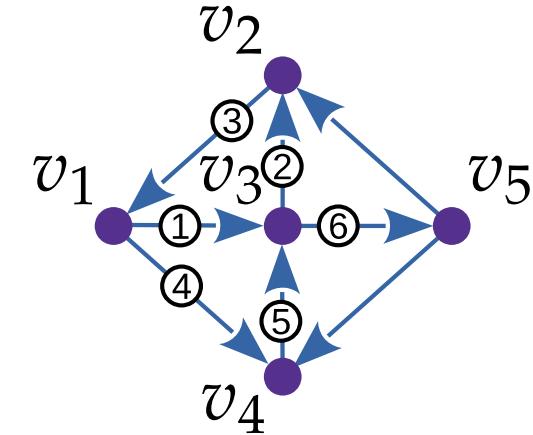
- **A simple graph**(simple graph) is a graph (usually undirected) that has at most one edge between every two vertices and in which there are no loops
- **Multigraph**(multigraph) is a graph that can have more than one edge between two vertices
- **Pseudograph**(pseudograph) is a multigraph in which feedback loops can exist



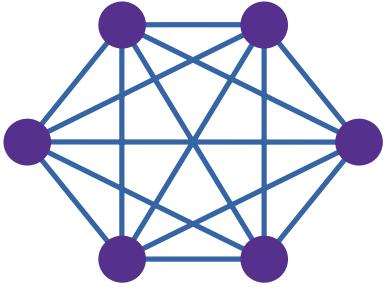
Graph theory - tour, path, ...



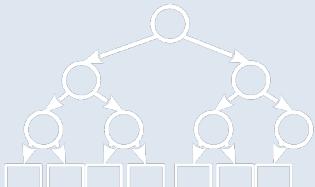
- **Tour or walk**(walk) from the top %to the top & is an alternating series of vertices and edges %, ', ', (,..., &, &
 - It is denoted shorter !, ",..., #or just !"..." #
- **Path**(trail)is a tour in which all the edges are different (it means that each edge is passed only once, but the vertices can be repeated)
- **Road or path**(path)is a trajectory(trail)in which all vertices are different (so they cannot be repeated)
- **Lap**(circuit)is a trajectory(trail)in which it is %=% &
- **Cycle**(cycles)is a track(path)on which it is %=% &



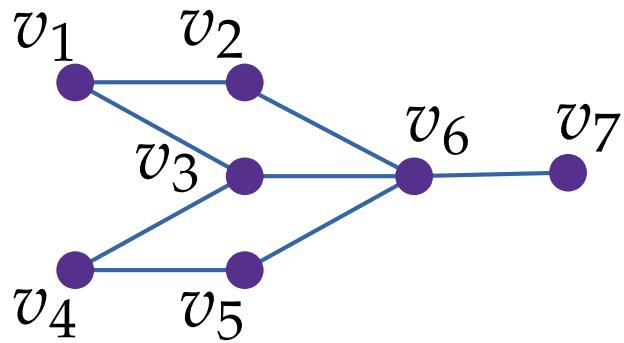
Graph theory



- **Complete**(complete)graph is a graph in which there is exactly one edge between every two vertices
 - Applies only to undirected graphs
 - Complete graph n -of order (n peaks) is denoted by $s!$
 - The number of edges in the complete graph = the number of binomial subsets of the set of vertices
$$\text{edges} \quad \binom{n}{2} = \frac{n!}{(n-2)!2!} = \frac{(n-1)}{2} = \binom{n}{2}$$
- Let it be ' $\subset V$ ' and ' $\subset E$ ', then it is ' $= (V', E')$ '**subgraph**(subgraph)
graph = (V', E')
 - If we have " $\subset V'$ ", then it is "[]"induced subgraph
graph , which contains all edges from the set , and which connect vertices from the set "

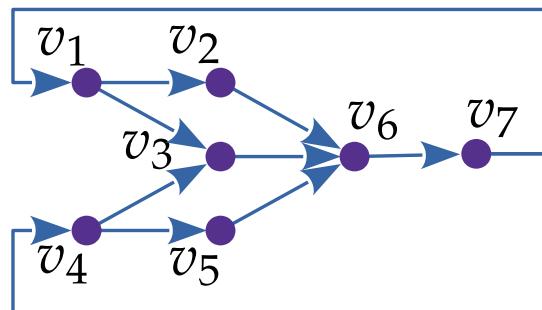
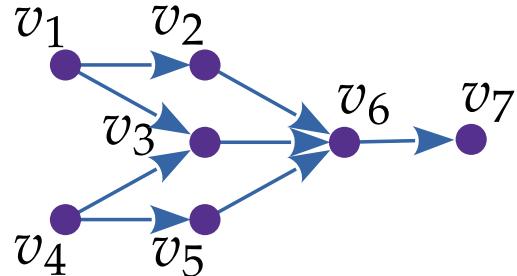


Graph connectivity – undirected graphs



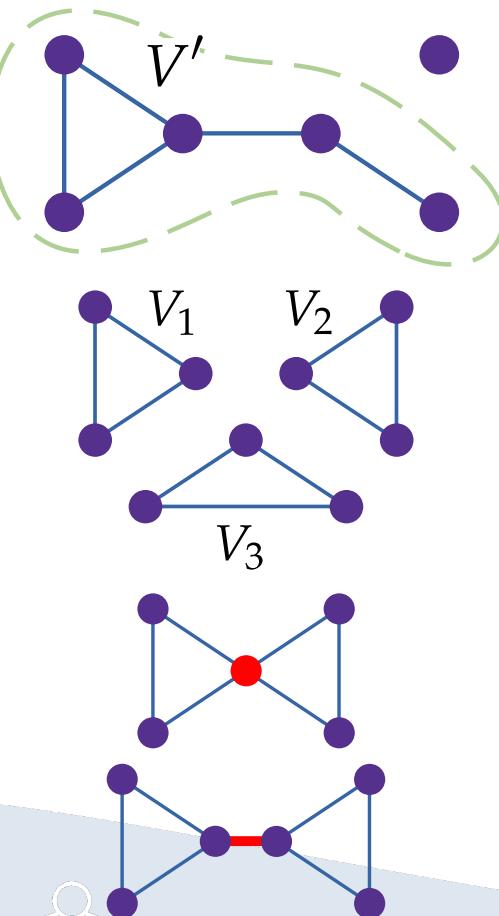
- We define a binary relation ' \sim ' such that by traversing the graph connects the vertices and . If the vertices and connected is valid
- A **connected undirected graph** = (V, \sim) -the undirected graph is connected_(connected)only if it holds for all pairs of vertices
 - which means it is the top **reachable**(reachable)from the top
 - but also vice versa, which is in accordance with the definition of an undirected graph

Graph connectivity – directed graphs



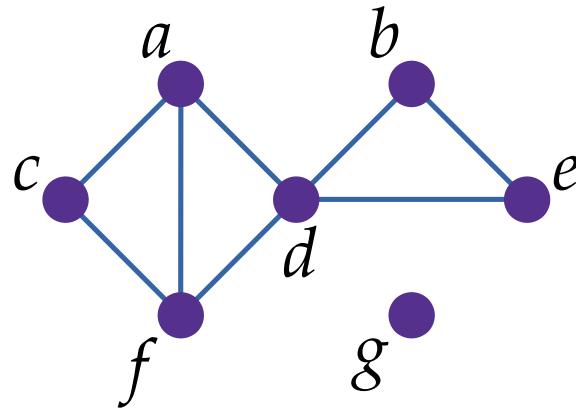
- **Weakly connected directed graph** (**weakly connected**)—is a directed graph whose undirected equivalent is connected according to the definition in the previous representation
- **A strongly connected directed graph G** (**strongly connected**)—is a directed graph where all pairs of vertices are mutually reachable
 - Note that the definition for a strongly connected directed graph is identical to the definition for a connected undirected graph
 - The difference is more than obvious, considering that in a directed graph, circles and cycles are created to make it strongly connected

Graph connectivity



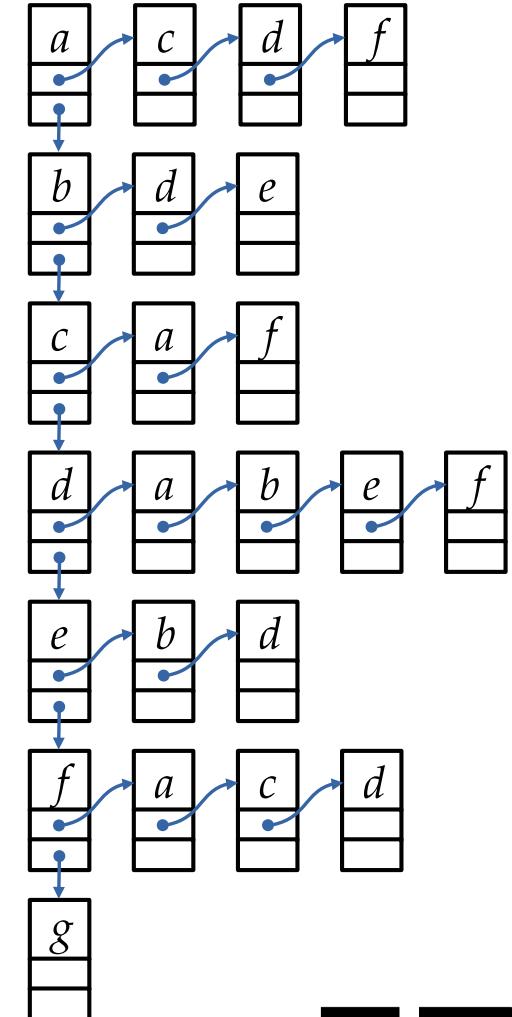
- **A disconnected graph** (disconnected graph)
 - We define a subset of vertices $' \subset V$ such that is an induced subgraph $G[V']$ connected and there is no tour between vertices in the set V' and the rest of the peaks /
$$\nexists i \in V' \nexists j \in V \setminus V': (i, j) \in E \vee (j, i) \in E$$
 - Such a graph is considered disconnected
- **Breakpoint**(articulation point)—is a vertex whose removal makes the graph disconnected
- **the bridge**(bridge)—is an edge whose removal makes the graph disconnected.

Graph storage

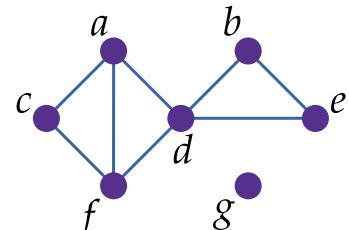


a	c	d	f
b	d	e	
c	a	f	
d	a	b	e
e	b	d	
f	a	c	d
g			

- It is possible to store using lists and matrices
- The advantage of the list is access to all the neighbors of a vertex because it is necessary $\deg(\)$ of steps in relation to $|V|$ for matrices
- The advantage of matrices is in individual interventions (addition or removal of edges) due to faster access and complexity (1) during maintenance
- An example in the form of a table (old representation) (left) and in the form of a two-dimensional linked list (right)



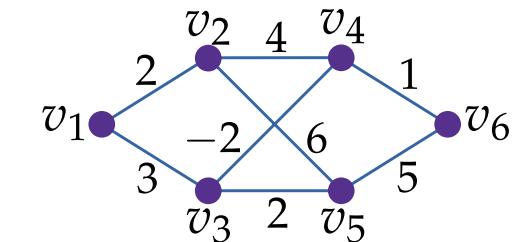
Graph storage



$$\mathbf{A} = \begin{matrix} & \begin{matrix} a & b & c & d & e & f & g \end{matrix} \\ \begin{matrix} a \\ b \\ c \\ d \\ e \\ f \\ g \end{matrix} & \begin{bmatrix} 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{matrix}$$

$$\mathbf{B} = \begin{matrix} & \begin{matrix} ac & ad & af & bd & be & cf & de & df \end{matrix} \\ \begin{matrix} a \\ b \\ c \\ d \\ e \\ f \\ g \end{matrix} & \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{matrix}$$

- An example in the form of an adjacency matrix (adjacency matrix) (**left-center**)
 - Symmetric for undirected graphs
- In the form of an incidence matrix (incidence matrix) (**left-down**)
- In the form of a weight matrix neighborhood (weighted adjacency matrix) (**right**)



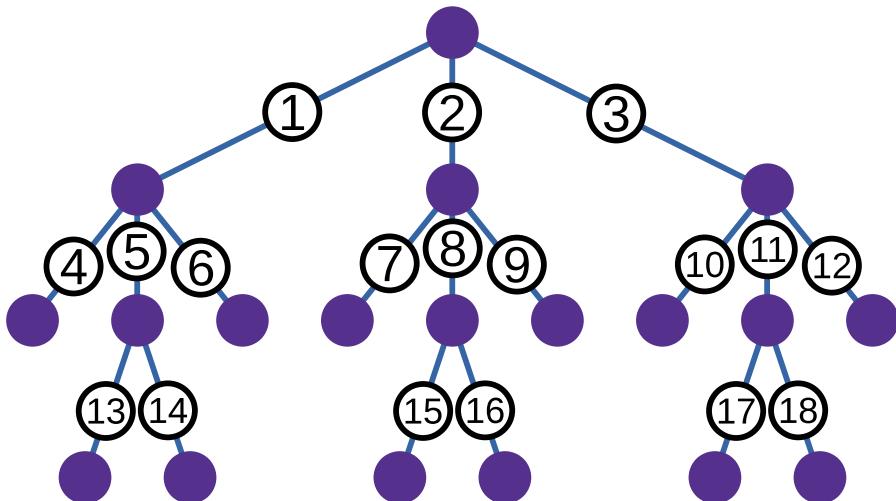
$$\mathbf{W} = \begin{matrix} & \begin{matrix} v_1 & v_2 & v_3 & v_4 & v_5 & v_6 \end{matrix} \\ \begin{matrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \end{matrix} & \begin{bmatrix} 0 & 2 & 3 & 0 & 0 & 0 \\ 2 & 0 & 0 & 4 & 6 & 0 \\ 3 & 0 & 0 & -2 & 2 & 0 \\ 0 & 4 & -2 & 0 & 0 & 1 \\ 0 & 6 & 2 & 0 & 0 & 5 \\ 0 & 0 & 0 & 1 & 5 & 0 \end{bmatrix} \end{matrix}$$

A tour of the graph_(graph traversal)

- Algorithms for traversal **trees** are not satisfactory for general graphs because:
 - A graph can have cycles so the tree algorithm could end up in an infinite loop
 - A graph can have separate and disconnected vertices, so we run the risk of not finding all connected partitions of vertices ()
- The two most famous graph traversal algorithms are:
 - Tour (first) in width (*Breadth First Search*; BFS)
 - Tour (first) in depth (*Depth First Search*; DFS)
- Except in simple applications (e.g. graph traversal, cycle detection, checking the connectivity of individual vertices), DFS and BFS are not interchangeable due to essential logical differences
- Both are the basis for many more complex algorithms and are theoretically equally fast, but in reality DFS is slightly slower because it is recursive

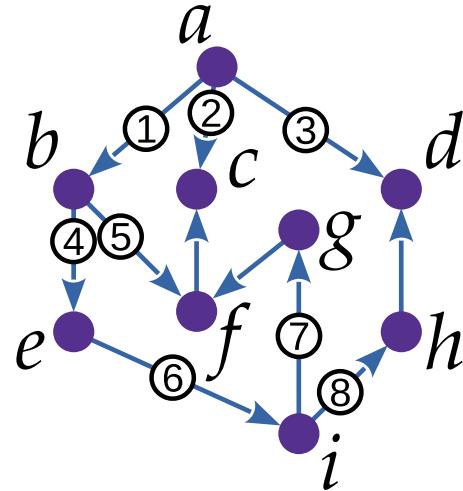
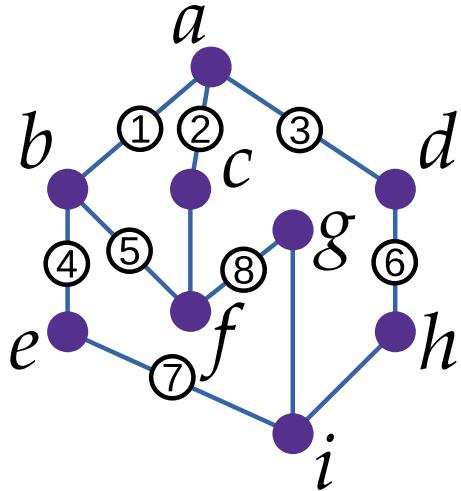


Traversing the graph (first) in width - BFS



- The concept of the BFS algorithm
 - Initially, we have only the root vertex in the list - the list is a natural data structure for the BFS algorithm
 - We take the first vertex from the list as the current vertex
 - First we go through ALL neighboring vertices of the current vertex
 - As we go around the neighboring peaks, we put them in the list
 - After we have visited all neighboring vertices, we take the next vertex from the list as the current one...
 - We repeat this procedure until we have visited all vertices in the graph
- If we have unconnected subgraphs, the procedure is repeated until there are unvisited vertices

Traversing the graph (first) in width - BFS

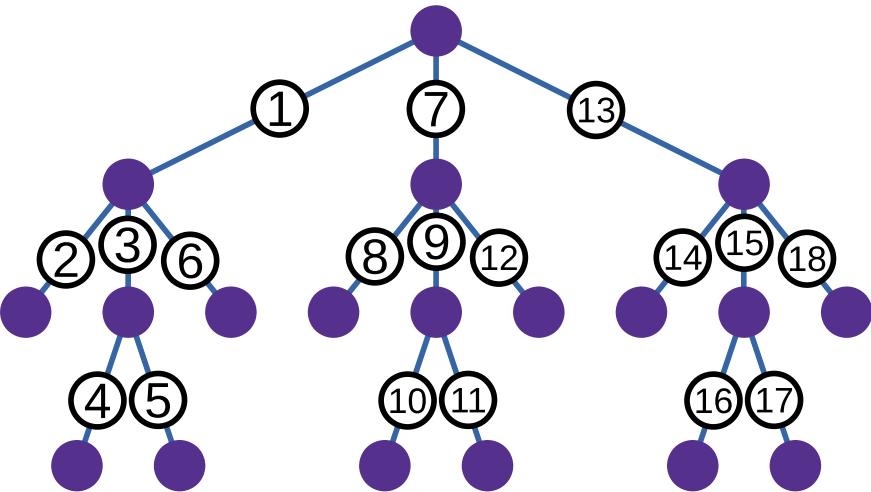


Step	u	queue Q
0		a
1	a	bcd
2	b	$cdef$
3	c	def
4	d	efh
5	e	fhi
6	f	hig
7	h	ig
8	i	g
9	g	

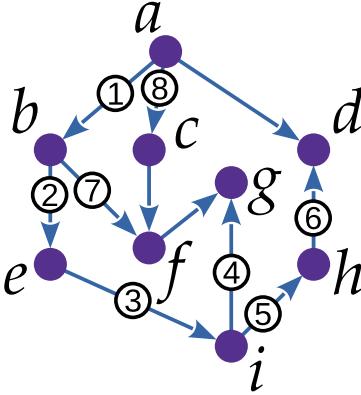
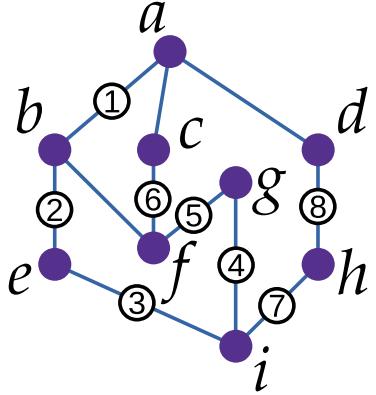
```
procedure BFS( $G$ )
    initialize all vertices in  $G$  as not visited
     $Q \leftarrow$  empty queue
    while there is an unvisited vertex  $u_0$  in  $G$  do
        mark  $u_0$  as visited
         $Q.\text{enqueue}(u_0)$ 
        while  $Q$  is not empty do
             $u \leftarrow Q.\text{dequeue}$ 
            process  $u$ 
            for  $v$  in adjacent vertices of  $u$  do
                if  $v$  is not visited then
                    mark  $v$  as visited
                     $Q.\text{enqueue}(v)$ 
```

Touring the graph (first) in depth - DFS

- Concept of DFS algorithm - recursive
 - We take the first unprecedented peak
 - We recursively descend to the first neighbor
 - The recursion is repeated, until we reach a leaf, then we go back to the first vertex where we still have unvisited neighbors, as soon as we start the recursive descent to the leaves again
 - A stack is a natural data structure for the DFS algorithm
- If we have unconnected subgraphs, the procedure is repeated until there are unvisited vertices



Touring the graph (first) in depth - DFS



Step	u	visited	stack S
0			a
1	a	No	bcd
2	b	No	$efcd$
3	e	No	$ifcd$
4	i	No	$ghfcd$
5	g	No	$fhdgc$
6	f	No	$chfd$
7	c	No	$hfcd$
8	h	No	$dfcd$
9	d	No	fcd
10	f	Yes	cd
11	c	Yes	d
12	d	Yes	

```

procedure DFS( $G$ )
    initialize all vertices in  $G$  as not visited
     $S \leftarrow$  empty stack
    while there is an unvisited vertex  $u_0$  in  $G$  do
         $S.push(u_0)$ 
        while  $S$  is not empty do
             $u \leftarrow S.pop$ 
            if  $u$  is not visited then
                mark  $u$  as visited
                process  $u$ 
                for  $v$  in reversed list of adjacent vertices of  $u$  do
                    if  $v$  is not visited then
                         $S.push(v)$ 

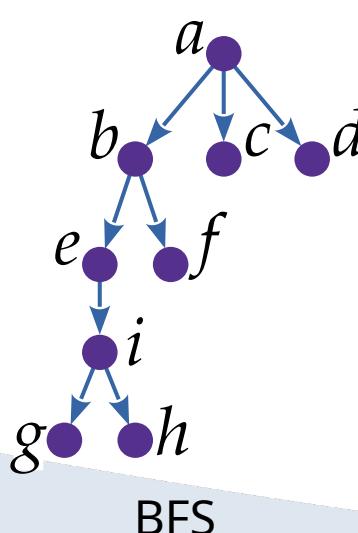
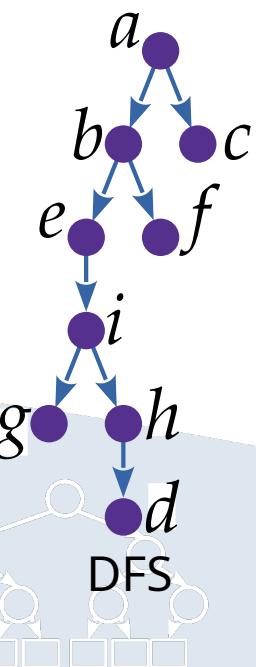
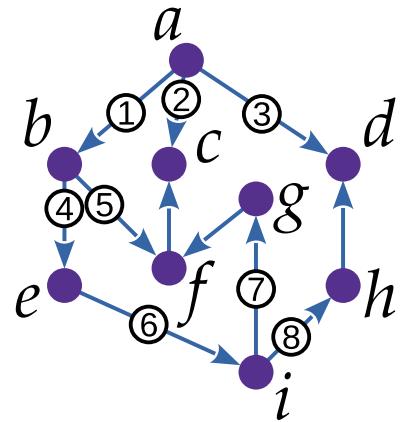
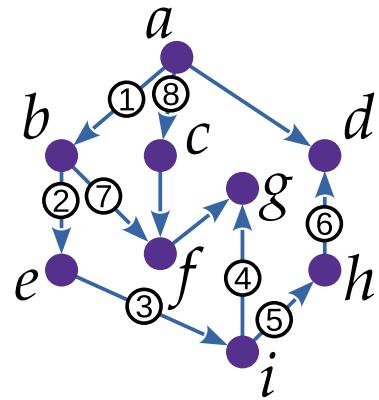
```

```

procedure DFS( $G$ )
    initialize all vertices in  $G$  as not visited
    while there is an unvisited vertex  $u_0$  in  $G$  do
        DFS_recursive( $G, u_0$ )
procedure DFS_recursive( $G, u$ )
    mark  $u$  as visited
    process  $u$ 
    for  $v$  in adjacent vertices of  $u$  do
        if  $v$  is not visited then
            DFS_recursive( $G, v$ )

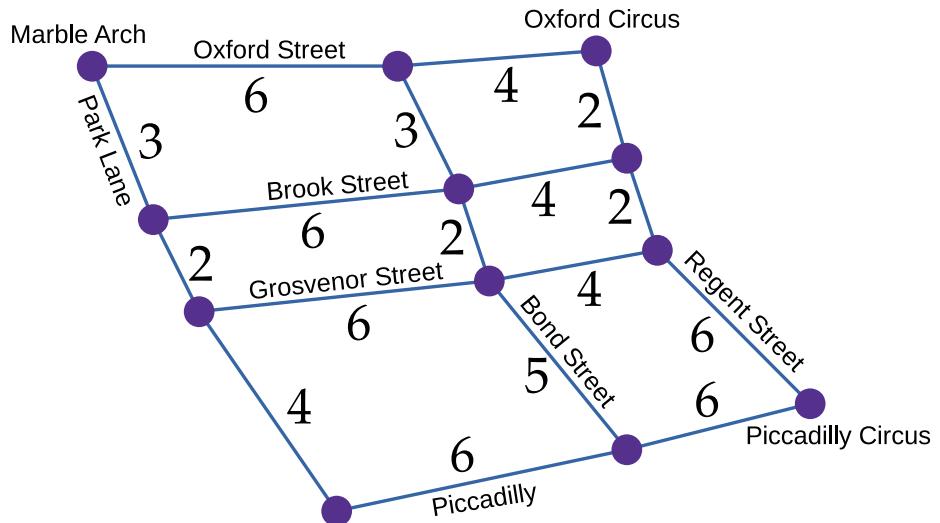
```

Crucifixion tree(spanning tree)



- The resulting path of both graph traversal algorithms (BFS and DFS) forms a tree in which all the vertices of the graph are. Such a tree is called **crucifixion tree**(spanning tree).
 - We record only the edges that represent **advancement** BFS and DFS algorithms according to unvisited vertices -**advanced edges** (forward edges)
 - We do not record the edges that return the algorithms to vertices that have already been visited -**reversible edges**(back edges)

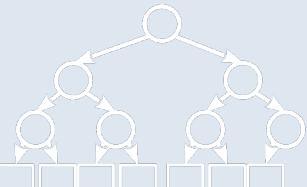
Shortest paths in the graph(shortest paths)



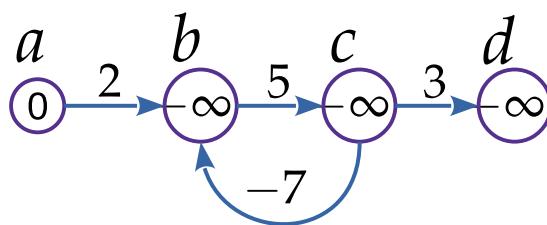
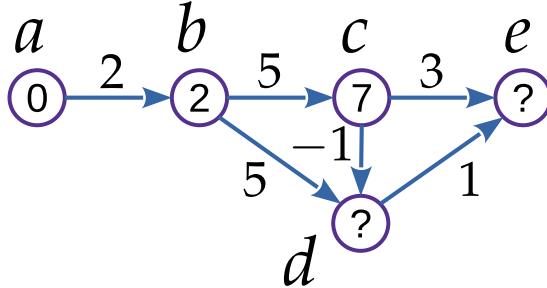
- Basic algorithms for numerous applications: transport, communications, distribution energy networks, designing integrated electronic circuits and more...
- Edges, as an abstraction (model) of links between two factors of a system, receive "weights"; label (,)
 - The term intermediate peak is also important. it is a vertex on the path between some two vertices in the graph, so neither the starting point nor the ending point
 - Labels(label)= distance of the peak from some reference peak (point)
 - It is most often stored as a member variable of the structure 'top'
- Two basic groups of algorithms:
 - Algorithms that find the shortest path between two specified vertices
 - Algorithms that find the shortest paths between all vertices in the graph(all-to-all)

Shortest paths in the graph

- The selection of the algorithm for searching for the shortest path depends on the weights of the edges in the graph; there are two basic groups of these algorithms:
 - **Label-setting**algorithms - once entered, the label is no longer changed
 - The label entered in this way is no longer checked
 - They work for graphs with exclusively positive edge weights - Dijkstra's algorithm
 - **Label-correcting**algorithms - all labels can be changed until the end of the procedure (convergence)
 - They work for graphs with any edge weights – Bellman-Ford algorithm



Shortest paths in the graph - negative weights

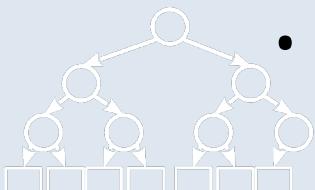


Iteration	$d(b)$	$d(c)$
1	2	7
2	0	5
3	-2	3
4	-4	1
5	-6	-1
...
∞	$-\infty$	$-\infty$

- Negative weight is a problem for**labeling** algorithms - let's take the example in the picture
 - If we reach the top came by tour , then it's his label () = 7
 - If we reach the top came by tour , then it's his label () = 6
- The second problem is a negative cycle, which is also a problem for**label-correcting** algorithms
 - The algorithm gets involved in an endless loop of label updates
 - At the moment when the algorithm should converge, it does not converge but continues to update the labels
 - Such a problem is insoluble

Dijkstra's algorithm

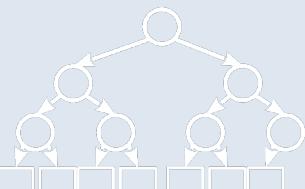
- It belongs to algorithms that calculate the shortest distance between two vertices -and .
- Label-setting algorithm
 - Once updated, the vertex label is no longer changed, except through another path
 - It is not able to work with negative edge weights
- For some trajectory
$$= - / \dots .$$
 - the concept of distance additivity can be applied
$$(\#, !) = (\#, \$) + (\$, !)$$
- If the trajectory also the shortest, then it is valid
$$\%\$!(\#, !) = \%\$!(\#, \$) + \%\$!(\$, !)$$
- Looking back, if each vertex on the shortest path "knows" (at least) its immediate predecessor, the entire path to the starting vertex can be reconstructed.
 - The idea is to count $(\#, !) = (\#, \•) + (\•, \•)$, from of = to = 2



Dijkstra's algorithm

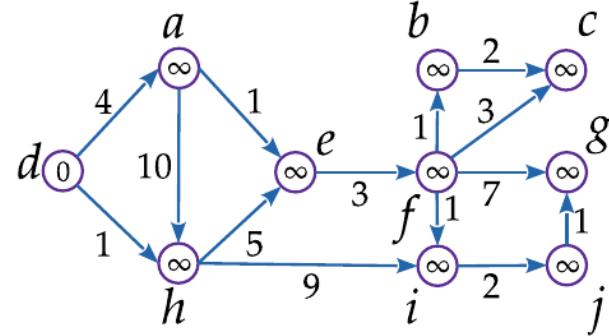
```
procedure DIJKSTRA( $G, source, destination$ )
  for each vertex  $v$  in  $V(G)$  do
     $d(v) \leftarrow \infty$ 
    predecessor( $v$ )  $\leftarrow null$ 
   $d(source) \leftarrow 0$ 
  work  $\leftarrow V(G)$ 
  while work is not empty do
     $u \leftarrow$  take a vertex from work having minimal  $d(u)$ 
    if  $u = destination$  then
      return
    for vertices  $v$  adjacent to  $u$  and in work do
      temp  $\leftarrow d(u) + w(uv)$ 
      if  $temp < d(v)$  then
         $d(v) \leftarrow temp$ 
        predecessor( $v$ )  $\leftarrow u$ 
```

- We initialize the labels of all vertices except the initial one at ∞ , and predecessors on $null$. We initialize the initial peak to 0.
- We start from the initial vertex, and calculate and update the distances to all neighbors
- During the tour
 - If the neighbor's distance is smaller than the current one, then we update the label and put a new predecessor to that neighbor
- The tour is based on the BFS algorithm which is prioritized by distance - less distant nodes go first
- We finish in the final peak

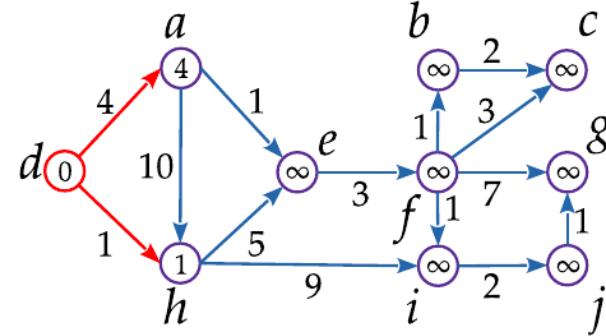


Dijkstra's algorithm - an example

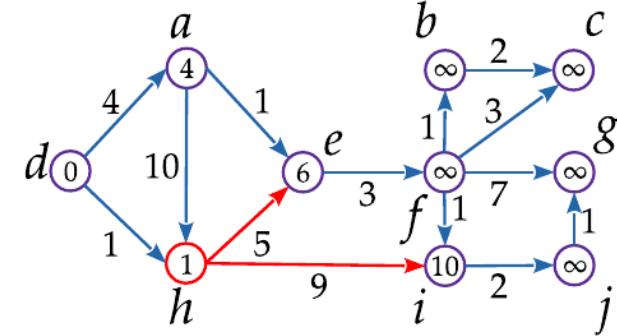
Iteration 0: The input graph G and initialization
 $work = \{a, b, c, d, e, f, g, h, i, j\}$



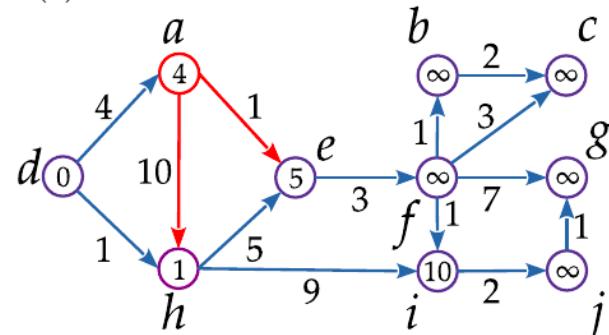
Iteration 1: $u = d$.
 $work = \{a, b, c, e, f, g, h, i, j\}$
 $d(a) = 4, d(h) = 1$



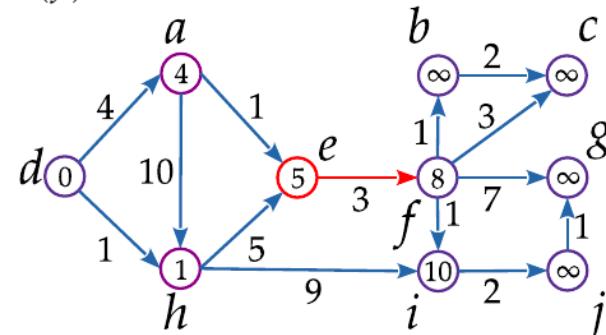
Iteration 2: $u = h$.
 $work = \{a, b, c, e, f, g, i, j\}$
 $d(e) = 6, d(i) = 10$



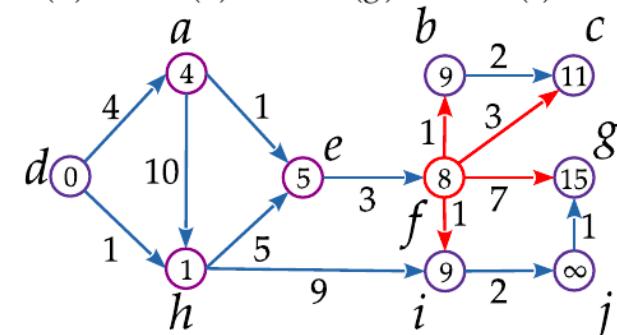
Iteration 3: $u = a$
 $work = \{b, c, e, f, g, i, j\}$
 $d(e) = 5$



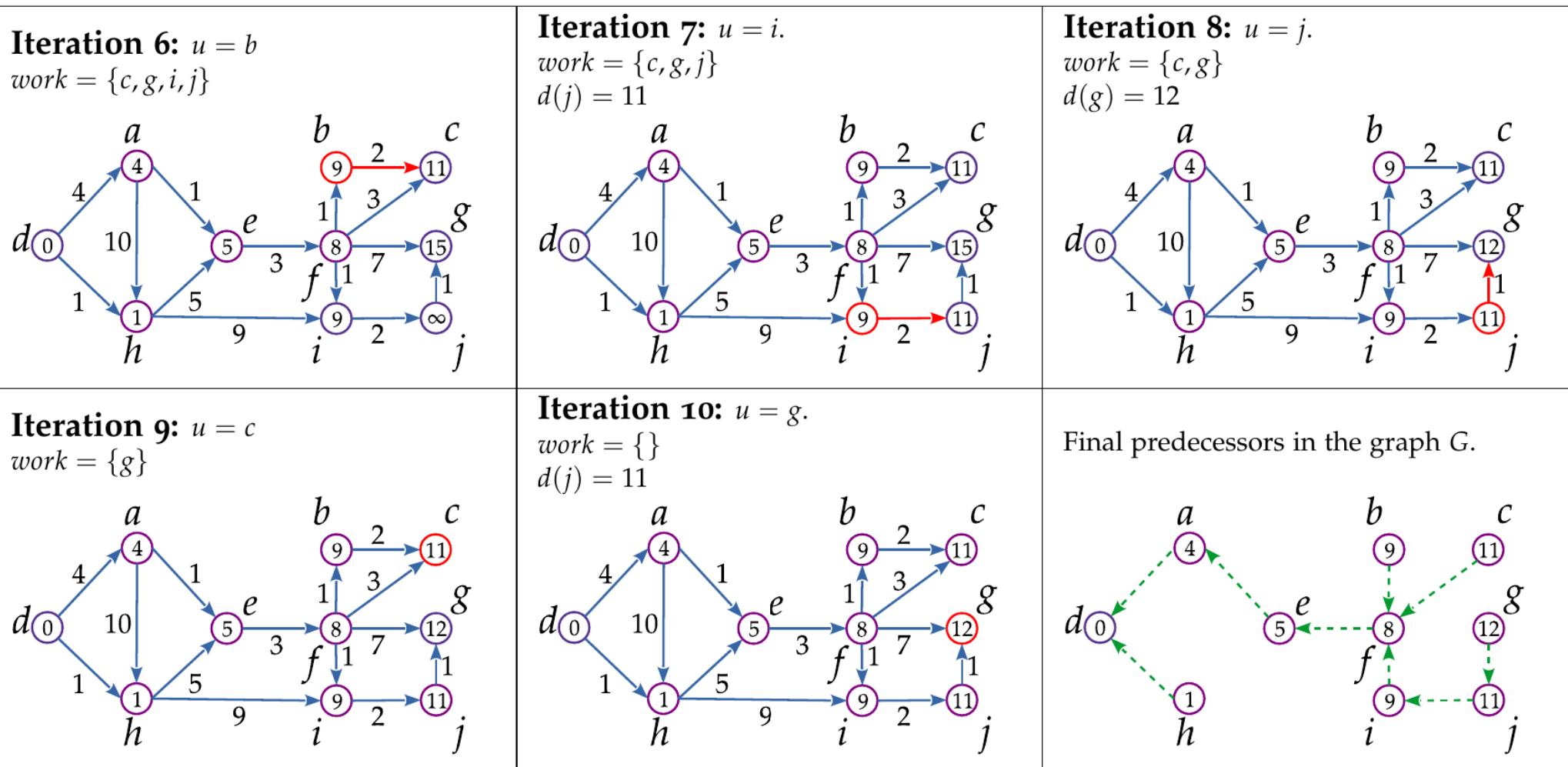
Iteration 4: $u = e$.
 $work = \{b, c, f, g, i, j\}$
 $d(f) = 8$



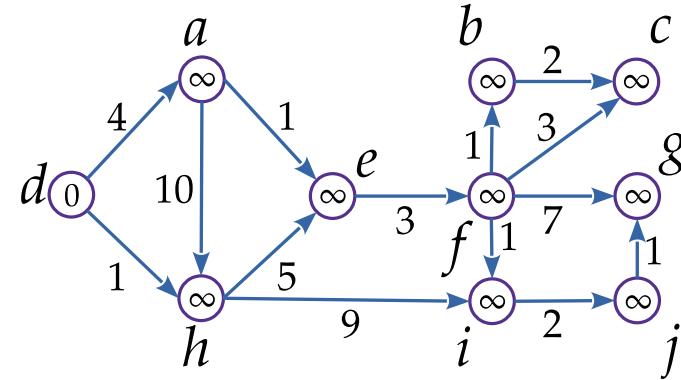
Iteration 5: $u = f$.
 $work = \{b, c, g, i, j\}$
 $d(b) = 9, d(c) = 11, d(g) = 15, d(i) = 9$



Dijkstra's algorithm - an example



Dijkstra's algorithm - an example

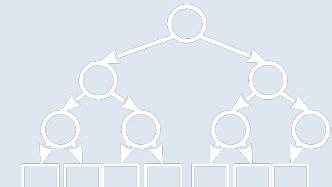


→ → i → f → → →

iteration current vertex	0	1 <i>d</i>	2 <i>h</i>	3 <i>a</i>	4 <i>e</i>	5 <i>f</i>	6 <i>b</i>	7 <i>i</i>	8 <i>j</i>	9 <i>c</i>	10 <i>g</i>
<i>a</i>	∞	$4/d$									
<i>b</i>	∞	∞	∞	∞	∞		$9/f$				
<i>c</i>	∞	∞	∞	∞	∞		$11/f$				
<i>d</i>	0										
<i>e</i>	∞	∞	$6/h$	$5/a$							
<i>f</i>	∞	∞	∞	∞	$8/e$						
<i>g</i>	∞	∞	∞	∞	∞		$15/f$	$15/f$	$15/f$	$12/j$	
<i>h</i>	∞	$1/d$									
<i>i</i>	∞	∞	$10/h$	$10/h$	$10/h$	$9/f$					
<i>j</i>	∞	∞	∞	∞	∞	∞		$11/i$			

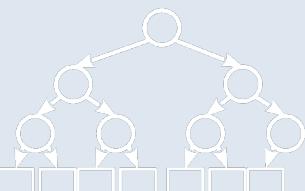
Dijkstra's algorithm - conclusion

- Algorithm complexity if the nearest vertex from *worklists* are determined sequentially - $()$
- If *work* the list is implemented as a Fibonacci stack (Fibonacci heap), then the complexity drops to $(+ \log())$

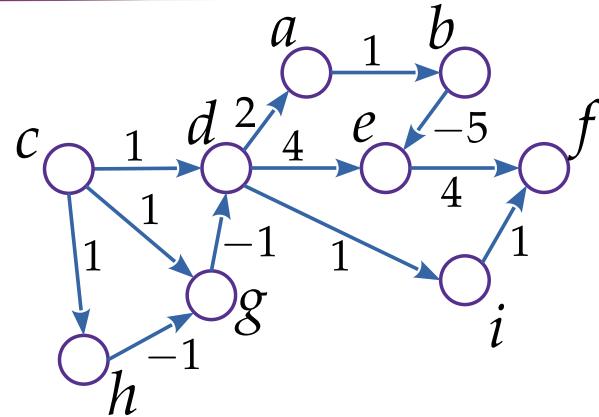


Bellman-Ford algorithm

- It belongs to algorithms that calculate the shortest distance between the initial vertex and all other vertices
- Label-correcting algorithm
 - All labels are updated until the moment of convergence - until there are no further label updates
 - It can work with negative weights in the graph
 - It is not able to work with negative cycles
- Slower than Dijkstrin's algorithm
- Works with edges
 - It checks all the edges in the graph and updates the distances of the vertices based on them
 - **Convergence is set to:**
 - While there are no more label updates on vertices
 - The program limit is -1 iteration through all edges of the graph



Bellman-Ford algorithm



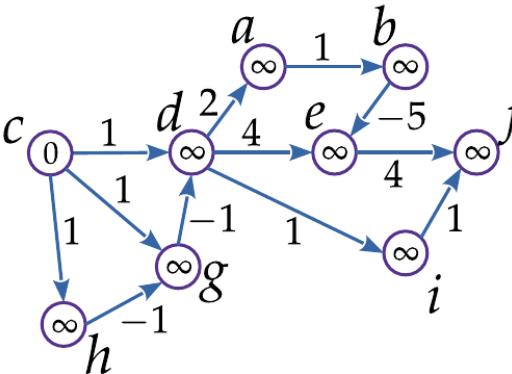
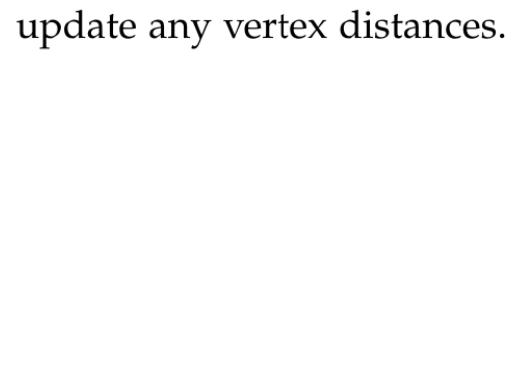
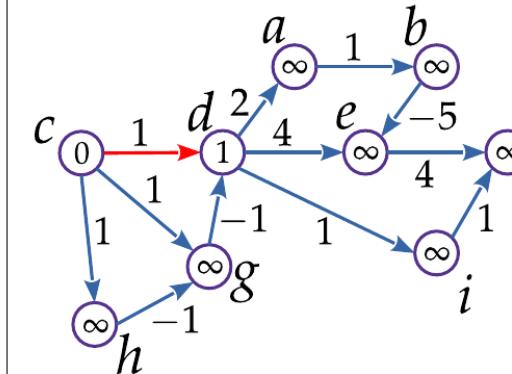
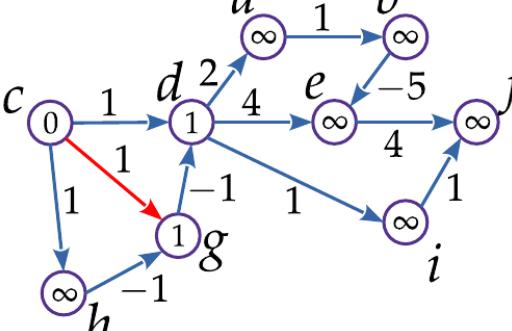
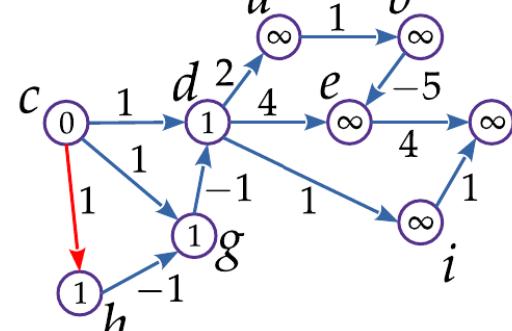
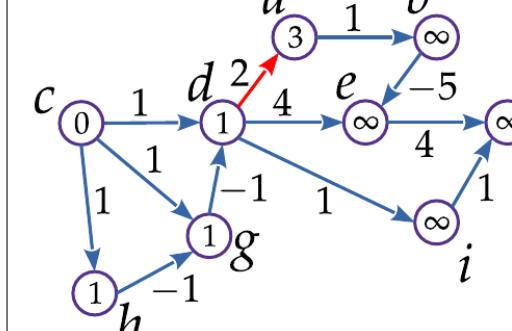
```
procedure BELLMAN-FORD( $G, source$ )
  for each vertex  $v$  in  $V(G)$  do
     $d(v) \leftarrow \infty$ 
     $predecessor(v) \leftarrow null$ 
     $d(source) \leftarrow 0$ 
  loop  $|V(G)| - 1$  times
    for each edge  $uv \in E(G)$  do
      if  $d(u) + w(uv) < d(v)$  then
         $d(v) \leftarrow d(u) + w(uv)$ 
         $predecessor(v) \leftarrow u$ 
    for each edge  $uv \in E(G)$  do
      if  $d(u) + w(uv) < d(v)$  then
        raise exception 'negative cycle has been detected'
```

- We sort the list of edges of the graph in some way, for example $() = \{ , \dots, h, \dots, , h, \dots \}$
- We pass through the edges in $()$. For the edge uv we update the vertex distance if we have

$$d(v) < d(u) + w(uv)$$

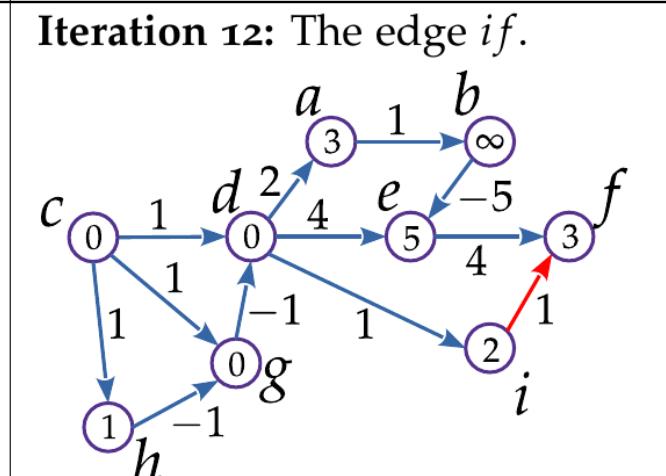
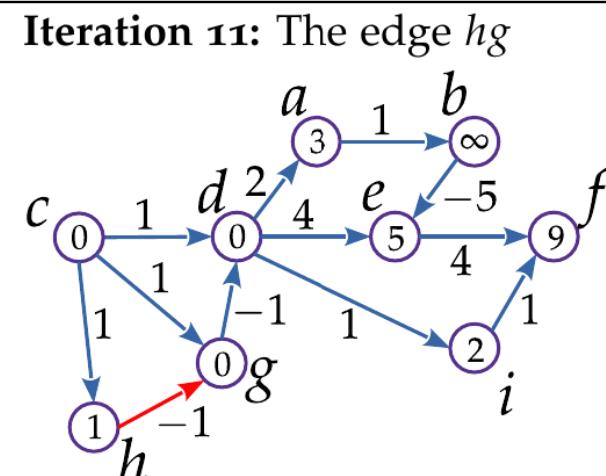
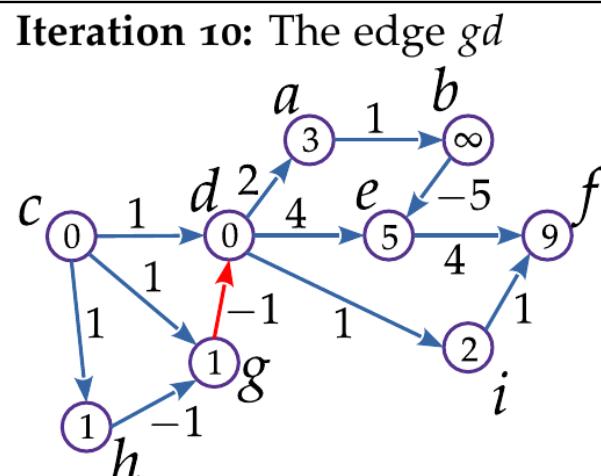
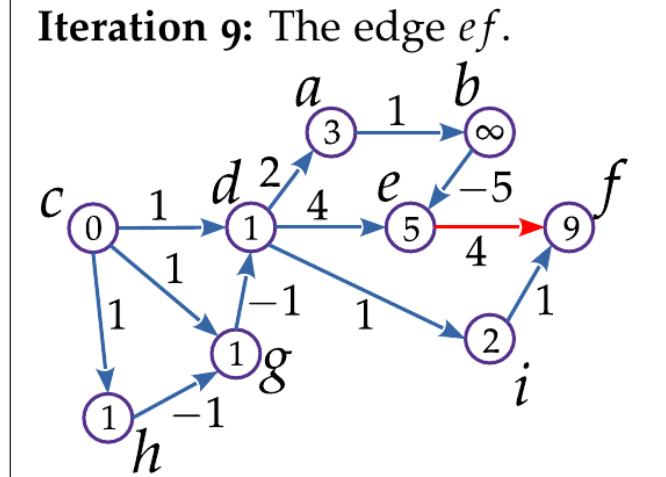
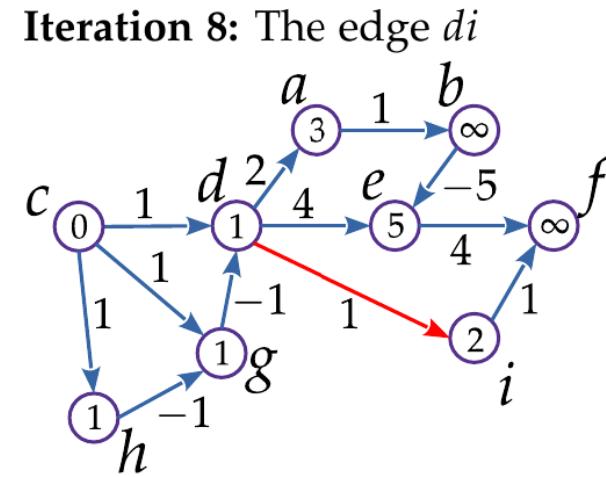
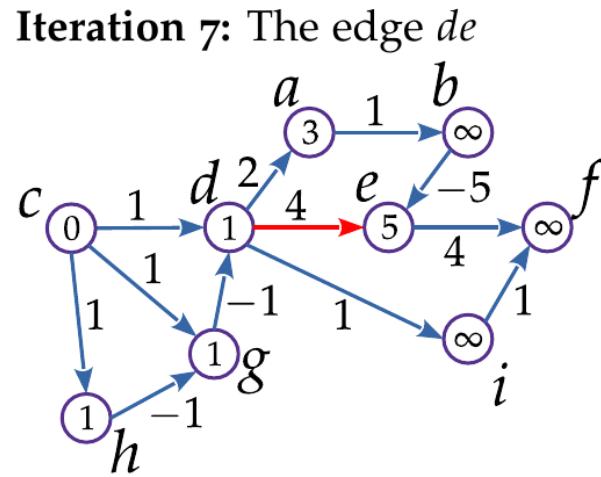
- We repeat this as much as possible $\vdash 1$ times
- At the end, we go through all the edges one more time, so if we still have a vertex whose label we can update - negative cycle

Bellman-Ford algorithm - example

Outer loop iteration 1		
Iteration 0: The input graph G and initialization 	Iterations 1 and 2 are ineffective, since edges ab and be do not update any vertex distances. 	Iteration 3: The edge cd . 
Iteration 4: The edge cg 	Iteration 5: The edge ch 	Iteration 6: The edge da . 

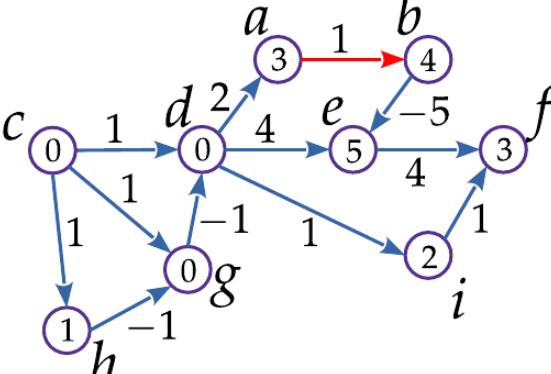
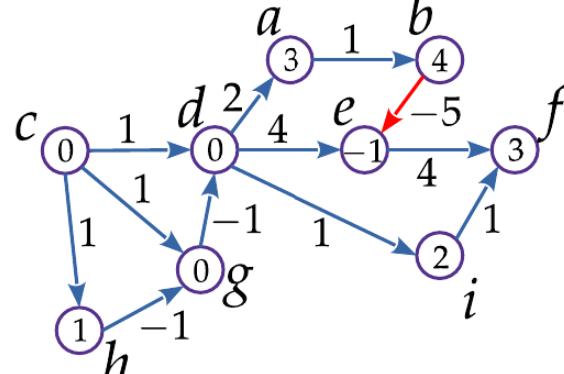
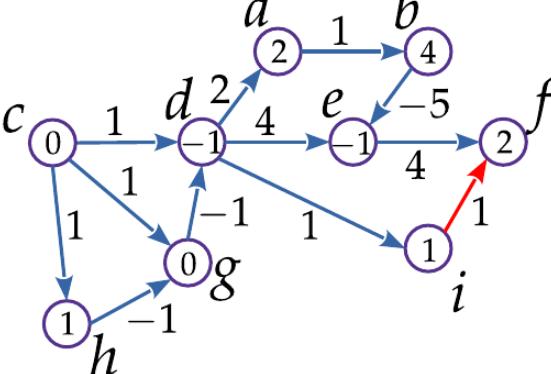
$\{ \ , \ , \ , \ , \ , h, \ , \ , \ , \ , h, \ , \ }$

Bellman-Ford algorithm - example



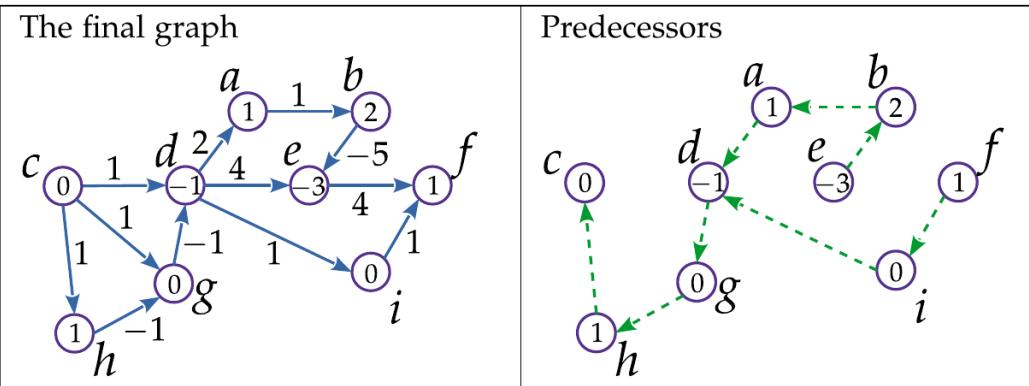
$() = \{ , , , , , h, , , , , , h, , \}$

Bellman-Ford algorithm - example

Outer loop iteration 2		
Iteration 1: The edge ab 	Iteration 2: The edge be 	We skip a number of the inner edge iterations here.
Iteration 12: The edge if 		

$\{ \ , \ , \ , \ , \ , h, \ , \ , \ , \ , h, \ , \}$

Bellman-Ford algorithm - example



	Iteration					
	0	1		2	3	4
a	∞	3		2	1	
b	∞			4	3	2
c	0					
d	∞	1	0	-1	-1	
e	∞	5		-1	-2	-3
f	∞	9	3	2	1	
g	∞	1	0			
h	∞	1				
i	∞	2		1	0	

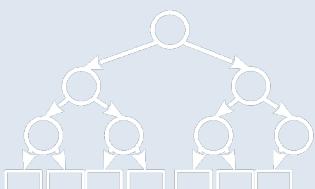
- After + 1 iteration we get the final result
 - The right graph represents the predecessors, and thus the shortest paths from to all other vertices
- An example of solving through a table
 - We follow ()and we update the distances in the column
- The complexity of the Bellman-Ford algorithm is (*).The outer loop iterates through the vertices, while the inner loop iterates through the edges

() = { , , , , h, , , , , h, , }

Bellman-Ford algorithm – faster version

```
procedure BELLMAN-FORD( $G, source$ )
  for each vertex  $v$  in  $V(G)$  do
     $d(v) \leftarrow \infty$ 
    predecessor( $v$ )  $\leftarrow null$ 
   $d(source) \leftarrow 0$ 
   $Q \leftarrow$  empty queue
   $Q.enqueue(source)$ 
  while  $Q$  is not empty do
     $u \leftarrow Q.dequeue$ 
    for  $v$  in adjacent vertices of  $u$  do
      if  $d(u) + w(uv) < d(v)$  then
         $d(v) \leftarrow d(u) + w(uv)$ 
        predecessor( $v$ )  $\leftarrow u$ 
        if  $v$  not in  $Q$  then
           $Q.enqueue(v)$ 
```

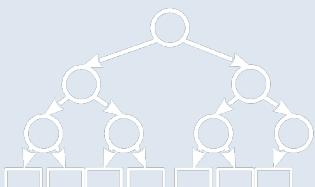
- The main difference is that we do not process all edges
- The initial vertex is placed in the list, and then only vertices (neighbors) whose label has changed
- Which means that only subgraphs are processed where there will potentially be some label change - it can happen if we have circles and cycles in the graph
- The complexity is still there (*)



Warshall-Floyd-Ingerman algorithm

- It belongs to algorithms that calculate the shortest distance between all graph vertices(all-to-all)
- Label-correcting algorithm
 - All labels are updated until the end of the algorithm
 - It can work with negative weights in the graph
 - It is not able to work with negative cycles
- Works with distance matrices(distance matrix)
- Let's imagine a set of vertices $= \{ , , , , \}$
- We map the vertices from so let's mark them with ordinal numbers

$$\begin{aligned} \forall \in : &= , 1 \leq \leq | \\)= , & (= , *= , += , , = \end{aligned}$$

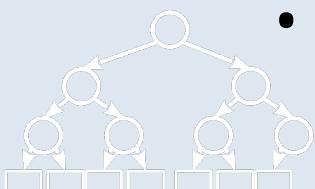


Warshall-Floyd-Ingerman algorithm

- The distance matrix for the previous set of vertices looks like

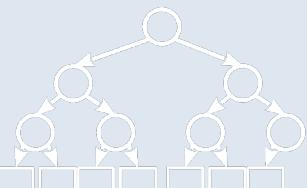
$$\mathbf{D} = \begin{matrix} & v_1 = a & v_2 = b & v_3 = c & v_4 = d & v_5 = e \\ v_1 = a & \left[\begin{matrix} d_{11} & d_{12} & d_{13} & d_{14} & d_{15} \\ d_{21} & d_{22} & d_{23} & d_{24} & d_{25} \\ d_{31} & d_{32} & d_{33} & d_{34} & d_{35} \\ d_{41} & d_{42} & d_{43} & d_{44} & d_{45} \\ d_{51} & d_{52} & d_{53} & d_{54} & d_{55} \end{matrix} \right] \\ v_2 = b \\ v_3 = c \\ v_4 = d \\ v_5 = e \end{matrix}$$

- distance between vertices "and # is denoted as "#
- it is to be expected that at least one path through the graph will be found between the vertices "and #-this is not necessary if the graph is not connected, but suppose it is
- you should choose the path that is the shortest or ;"&(", #)



Warshall-Floyd-Ingerman algorithm

- If we have a new trajectory that passes through an intermediate vertex !
=)... !... -
 - is considered to be a trajectory shorter if valid) %, !(!, < %(& &)
- The WFI algorithm iterates over the vertices of the graph placing them as intermediate vertices !
 - in this way, it is tested whether it is an intermediate peak on the minimum path between " and #
 - since the WFI algorithm calculates the distances between all vertices, we have three loops with which we iterate through the vertices of the graph, selecting in each of them ", #and !



Warshall-Floyd-Ingerman algorithm

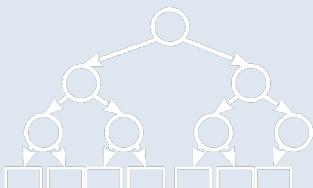
- Distances are calculated as

$$d_{01} = ; \quad d_{23} = \begin{cases} 0 & , \\ \min(d_{01}, d_{02} + d_{21}) & , \end{cases} \geq 1$$

- when we look at the intermediate peak i , the distance is the minimum between the distances calculated for the intermediate peak i and the distance of the path passing through the intermediate peak i
- Paths (predecessors) are saved in the path matrix as

$$C_{01} = \begin{cases} 23-01 & , \\ 23-21 & , \end{cases} \leq \begin{cases} 02^{23-} + 23-21 \\ 02^{23-} + 21^{23-} \end{cases}$$

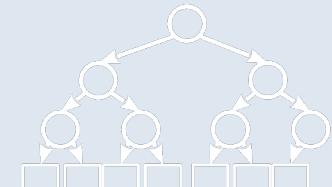
- which means that if we updated the distance d_{ij} then we must also put that the predecessor of the vertex j is the intermediate peak i



Warshall-Floyd-Ingerman algorithm

- At the beginning, we have the initial distance matrix, which is $. =$, and contains the distances of only directly connected vertices, the calculation of other distances is a matter of the WFI algorithm
- The initial path matrix is defined from the neighborhood weight matrix In a way

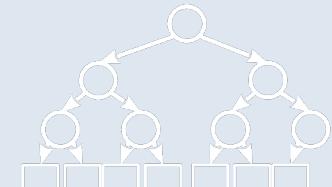
$$\begin{array}{l} .\ "#= J \\ \quad , \quad = \quad = 0_{\#} \\ \quad , \quad \neq \quad "# \neq 0 \end{array}$$



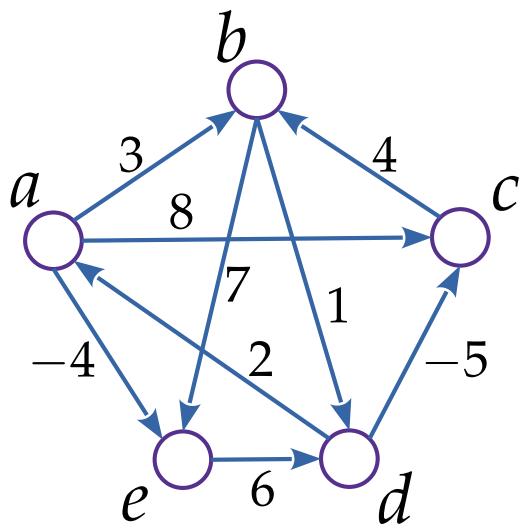
Warshall-Floyd-Ingerman algorithm

```
procedure WFI(W)
    Create initial distance matrix  $D = D^0$  from  $W$ 
    Create initial path matrix  $\Pi = \Pi^0$  from  $W$ 
    for  $k$  from 1 to  $|V|$  do
        for  $i$  from 1 to  $|V|$  do
            for  $j$  from 1 to  $|V|$  do
                if  $D[i, k] + D[k, j] < D[i, j]$  then
                     $D[i, j] = D[i, k] + D[k, j]$ 
                     $\Pi[i, j] = \Pi[k, j]$ 
```

- The algorithm is simple, it has three loops that we iterate through **peaks ", #and !**
- Thus, the complexity of the algorithm is somewhat high (*)

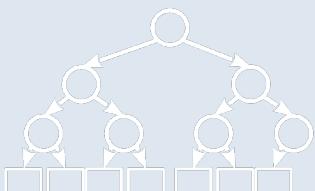


WFI algorithm - example

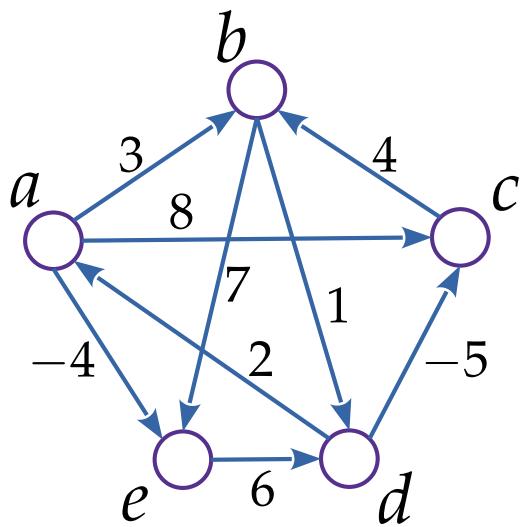


$$\mathbf{D}^0 = \begin{matrix} & \begin{matrix} a & b & c & d & e \end{matrix} \\ \begin{matrix} a \\ b \\ c \\ d \\ e \end{matrix} & \left[\begin{matrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{matrix} \right] \end{matrix}, \Pi^0 = \begin{matrix} & \begin{matrix} a & b & c & d & e \end{matrix} \\ \begin{matrix} a \\ b \\ c \\ d \\ e \end{matrix} & \left[\begin{matrix} \text{null} & 1 & 1 & \text{null} & 1 \\ \text{null} & \text{null} & \text{null} & 2 & 2 \\ \text{null} & 3 & \text{null} & \text{null} & \text{null} \\ 4 & \text{null} & 4 & \text{null} & \text{null} \\ \text{null} & \text{null} & \text{null} & 5 & \text{null} \end{matrix} \right] \end{matrix}$$

$$\mathbf{D}^1 = \begin{matrix} & \begin{matrix} a & b & c & d & e \end{matrix} \\ \begin{matrix} a \\ b \\ c \\ d \\ e \end{matrix} & \left[\begin{matrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5(\infty) & -5 & 0 & -2(\infty) \\ \infty & \infty & \infty & 6 & 0 \end{matrix} \right] \end{matrix}, \Pi^1 = \begin{matrix} & \begin{matrix} a & b & c & d & e \end{matrix} \\ \begin{matrix} a \\ b \\ c \\ d \\ e \end{matrix} & \left[\begin{matrix} \text{null} & 1 & 1 & \text{null} & 1 \\ \text{null} & \text{null} & \text{null} & 2 & 2 \\ \text{null} & 3 & \text{null} & \text{null} & \text{null} \\ 4 & 1(\text{null}) & 4 & \text{null} & 1(\text{null}) \\ \text{null} & \text{null} & \text{null} & 5 & \text{null} \end{matrix} \right] \end{matrix}$$

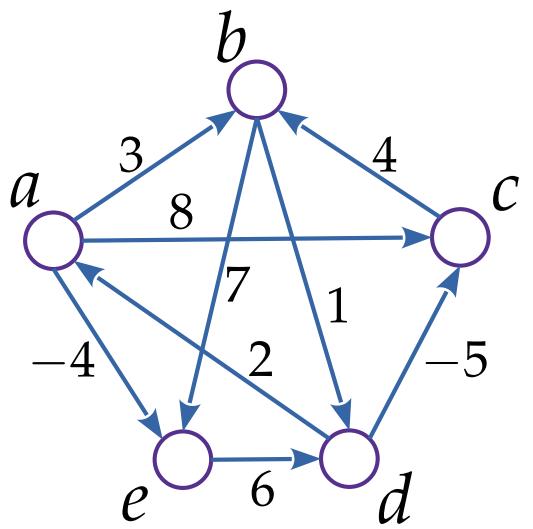


WFI algorithm - example

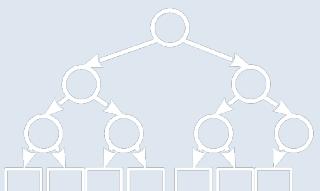


$$\begin{aligned}
 \mathbf{D}^2 &= \begin{matrix} a & b & c & d & e \\ a & 0 & 3 & 8 & 4(\infty) & -4 \\ b & \infty & 0 & \infty & 1 & 7 \\ c & \infty & 4 & 0 & 5(\infty) & 11(\infty) \\ d & 2 & 5 & -5 & 0 & -2 \\ e & \infty & \infty & \infty & 6 & 0 \end{matrix}, \quad \mathbf{\Pi}^2 = \begin{matrix} a & b & c & d & e \\ a & \text{null} & 1 & 1 & 2(\text{null}) & 1 \\ b & \text{null} & \text{null} & \text{null} & 2 & 2 \\ c & \text{null} & 3 & \text{null} & 2(\text{null}) & 2(\text{null}) \\ d & 4 & 1 & 4 & \text{null} & 1 \\ e & \text{null} & \text{null} & \text{null} & 5 & \text{null} \end{matrix} \\
 \mathbf{D}^3 &= \begin{matrix} a & b & c & d & e \\ a & 0 & 3 & 8 & 4 & -4 \\ b & \infty & 0 & \infty & 1 & 7 \\ c & \infty & 4 & 0 & 5 & 11 \\ d & 2 & -1(5) & -5 & 0 & -2 \\ e & \infty & \infty & \infty & 6 & 0 \end{matrix}, \quad \mathbf{\Pi}^3 = \begin{matrix} a & b & c & d & e \\ a & \text{null} & 1 & 1 & 2 & 1 \\ b & \text{null} & \text{null} & \text{null} & 2 & 2 \\ c & \text{null} & 3 & \text{null} & 2 & 2 \\ d & 4 & 3(1) & 4 & \text{null} & 1 \\ e & \text{null} & \text{null} & \text{null} & 5 & \text{null} \end{matrix}
 \end{aligned}$$

WFI algorithm - example



$$\begin{aligned}
 D^4 &= \begin{bmatrix} a & b & c & d & e \\ a & 0 & 3 & -1(8) & 4 & -4 \\ b & 3(\infty) & 0 & -4(\infty) & 1 & -1(7) \\ c & 7(\infty) & 4 & 0 & 5 & 3(11) \\ d & 2 & -1 & -5 & 0 & -2 \\ e & 8(\infty) & 5(\infty) & 1(\infty) & 6 & 0 \end{bmatrix}, \Pi^4 = \begin{bmatrix} a & b & c & d & e \\ a & \text{null} & 1 & 4(1) & 2 & 1 \\ b & 4(\text{null}) & \text{null} & 4(\text{null}) & 2 & 1(2) \\ c & 4(\text{null}) & 3 & \text{null} & 2 & 1(2) \\ d & 4 & 3 & 4 & \text{null} & 1 \\ e & 4(\text{null}) & 3(\text{null}) & 4(\text{null}) & 5 & \text{null} \end{bmatrix} \\
 D^5 &= \begin{bmatrix} a & b & c & d & e \\ a & 0 & 1(3) & -3(-1) & 2(4) & -4 \\ b & 3 & 0 & -4 & 1 & -1 \\ c & 7 & 4 & 0 & 5 & 3 \\ d & 2 & -1 & -5 & 0 & -2 \\ e & 8 & 5 & 1 & 6 & 0 \end{bmatrix}, \Pi^5 = \begin{bmatrix} a & b & c & d & e \\ a & \text{null} & 3(1) & 4(4) & 5(2) & 1 \\ b & 4 & \text{null} & 4 & 2 & 1 \\ c & 4 & 3 & \text{null} & 2 & 1 \\ d & 4 & 3 & 4 & \text{null} & 1 \\ e & 4 & 3 & 4 & 5 & \text{null} \end{bmatrix}
 \end{aligned}$$

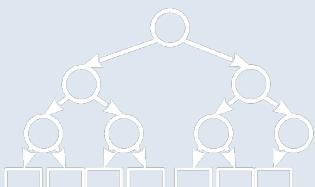


WFI algorithm - example

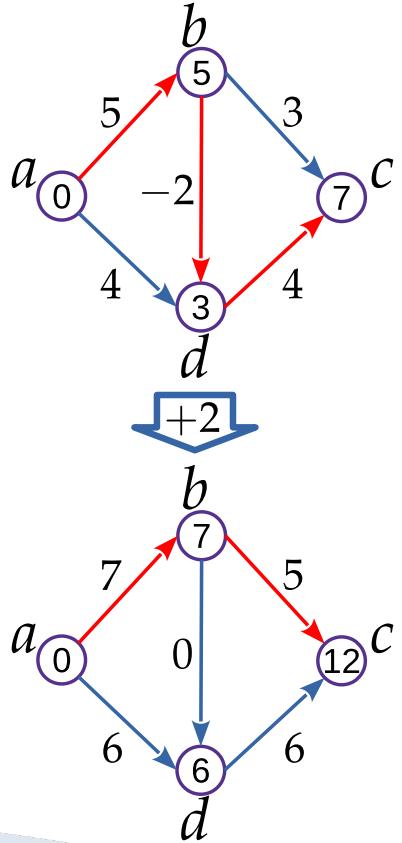
k	vertex
$\pi_{35}^5 = 1$	$v_5 = e$
$\pi_{31}^5 = 4$	$v_1 = a$
$\pi_{34}^5 = 2$	$v_4 = d$
$\pi_{32}^5 = 3$	$v_2 = b$
$\pi_{33}^5 = \text{null}$	$v_3 = c$

$$\Pi^5 = \begin{bmatrix} a & b & c & d & e \\ a & \text{null} & 3(1) & 4(4) & 5(2) & 1 \\ b & 4 & \text{null} & 4 & 2 & 1 \\ c & 4 & 3 & \text{null} & 2 & 1 \\ d & 4 & 3 & 4 & \text{null} & 1 \\ e & 4 & 3 & 4 & 5 & \text{null} \end{bmatrix}$$

- We look at the distance and the shortest path between $i = 3 =$ and $j = 5 =$
- We read the distance directly from $\$,$ that is $\$ \% \$ = 3$
- Reading the path matrix is interpreted as
 - If it is initial peak, a "final, then we have a trajectory
 $! \$ \% ... \$ \% & \$$
 - In the matrix Π we have ' $! =$ ', and then ' $! \$ =$ ' $- 1$, and ' $! (\$ \% &) =$ ' $- 1$, until ' $! (\$ \%) =$
 - ' $!$ ' will by definition be null and we end there
- For the trajectory we go backwards through the matrix of $\$ \% \$$ where $\$ = 3$, And $= 5$, so it is
 - $= 5$ and that's the top
 - $= 4$ $! \$ = 4$ and that's the top
 - $- 1 = 3$ $! \$ = 3$ and that's the top
 - $- 2 = 2$ $! (\$ \% &) = 2$ and that's the top
 - $- 3 = 1$ $! (\$ \%) = 1$ and that's the top
 - ' $!$ ' and we end there.



Weight transformation



- We really want to use Dijkstra's algorithm, but the negative weights on the edges bother us
- Is it possible to somehow transform the graph to remove the negative weights?
- A naive attempt would be an identical translation towards the most negative edge weight
- The shortest path from a to c is in the original graph with a distance of 7
- Adding a weight of 2 to all edges disrupts this and is now the shortest path with a distance of 12

Weight transformation

- What we know from previous displays is

$$\leq) + (()$$

- means the vertex distance is not greater than the vertex distance increased by the weight of the edge $()$

- if we write it down differently, we get that it is

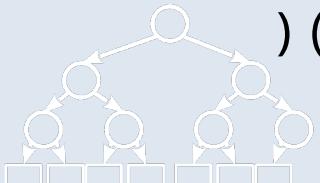
$$0 \leq (+) - (=)() ()$$

- If we use the previous expression on the path $=) (... !$ we get the distance

$!=)$

$$), (!) ; = T ' (" " >) =$$

$$) ((+) -) (+ \cdot (\cdot +) !=) ! + (!=) - (" <) () ()$$



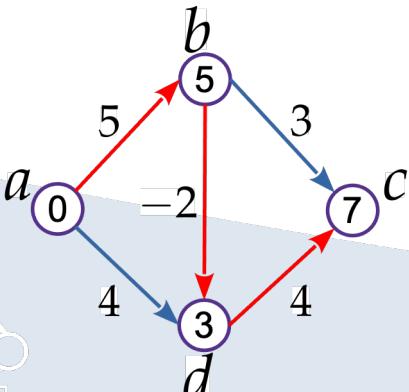
Weight transformation

- Also valid

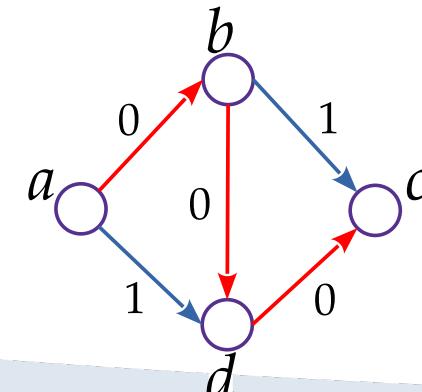
$$;) , (!= T) " >) \begin{pmatrix} != \\ < \\ (+) - ! () \end{pmatrix} + () - (!) =$$

- The transformation is bijective, so it is valid

$$) (= ') + ! - ()) () ()$$



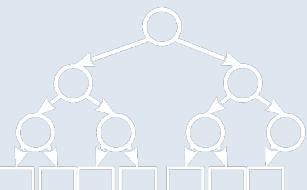
v_i	v_j	$w(v_i v_j)$	$d(v_i)$	$d(v_j)$	$w'(v_i v_j)$
a	b	5	0	5	$5+0-5=0$
a	d	4	0	3	$4+0-3=1$
b	d	-2	5	3	$-2+5-3=0$
b	c	3	5	7	$3+5-7=1$
d	c	4	3	7	$4+3-7=0$



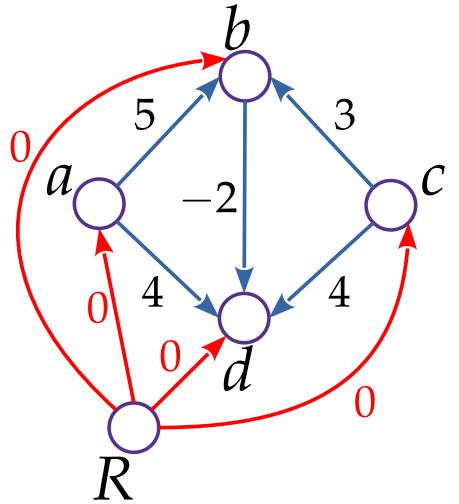
Weight transformation

- 1: Using a *label-correcting* algorithm, determine the shortest distance to every vertex in the input graph G from an arbitrary reference vertex v_r . All vertices in the input graph G must be reachable from the reference vertex v_r .
- 2: Transform the input graph G into the non-negative weighted graph G' using the bijective transformation given in (3.51).
- 3: **for** $\forall v_i, v_k \in V(G')$ **do**
- 4: Find the shortest path between source and destination vertices v_i and v_k in the transformed graph G' using a *label-setting* algorithm. The length of this path is $L'(v_i v_k)$.
- 5: Use the inverse transformation in (3.51) to get the original length of the shortest path as $L(v_i v_k) = L'(v_i v_k) + d(v_k) - d(v_i)$.

- In order to perform the weight transformation, we first need to determine the Bellman-Ford algorithm the distance of all vertices from one particular vertex
- Then we do the transformation
- After that, using Dijkstra's algorithm, we determine the distance between all pairs of vertices in the graph, which gives us *all-to-all* distance



Weight transformation



- If we have a disconnected graph, then it is impossible to calculate the distance of all vertices from one reference vertex
- Then we add an artificial vertex (let's say R) which we connect with all other vertices of the graph by edges
 - The weights of these artificially added edges are 0
- Now we can determine the distance of all vertices from the artificial vertex R and based on that make a transformation

v_i	v_j	$w(v_i v_j)$	$d(v_i)$	$d(v_j)$	$w'(v_i v_j)$
a	b	5	0	0	$5+0-0=5$
a	d	4	0	-2	$4+0+2=6$
b	d	-2	0	-2	$-2+0+2=0$
c	b	3	0	0	$3+0-0=3$
c	d	4	0	-2	$4+0+2=6$

