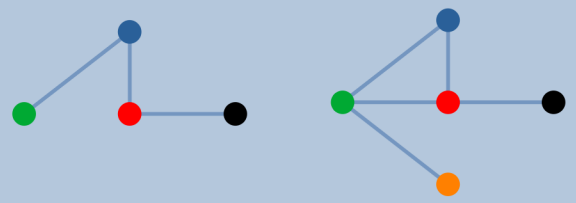
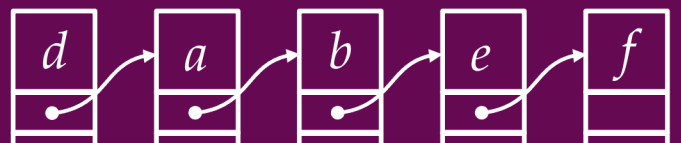
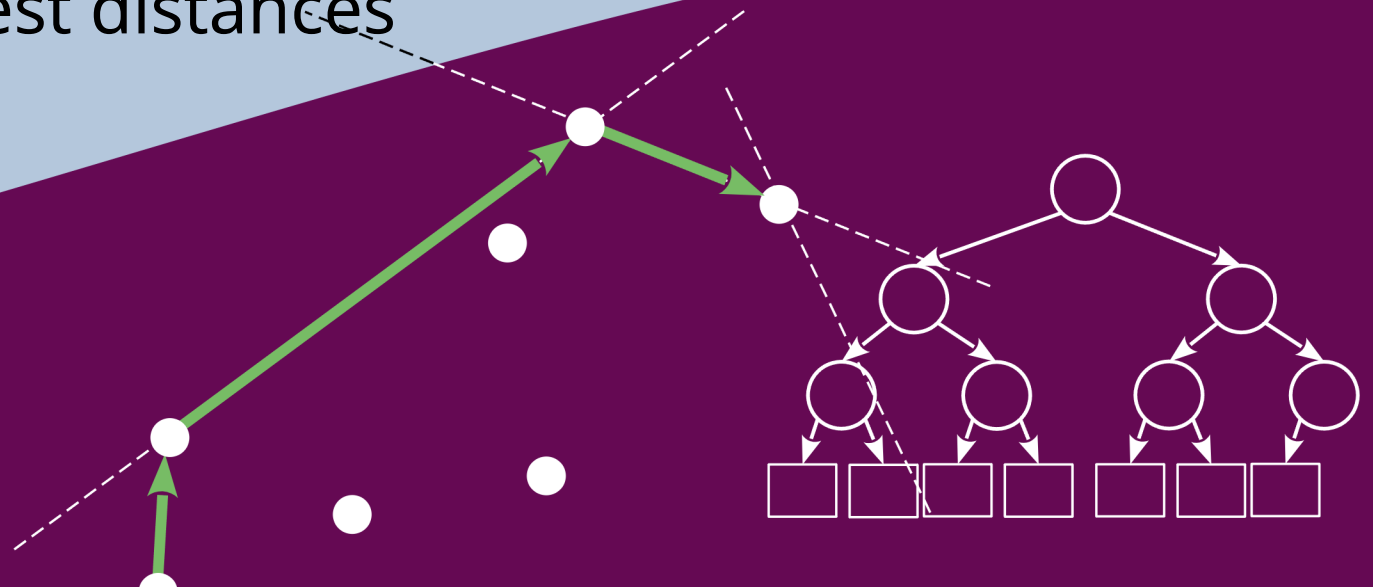
Advanced algorithms and structures data

Week 9: Algorithms over graphs
Graph theory, Shortest distances



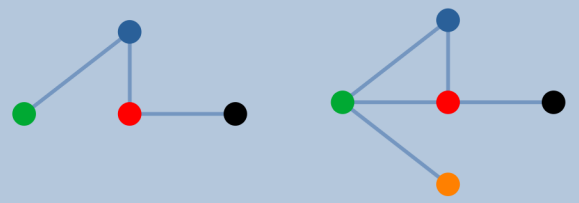


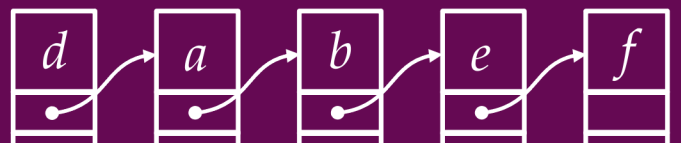


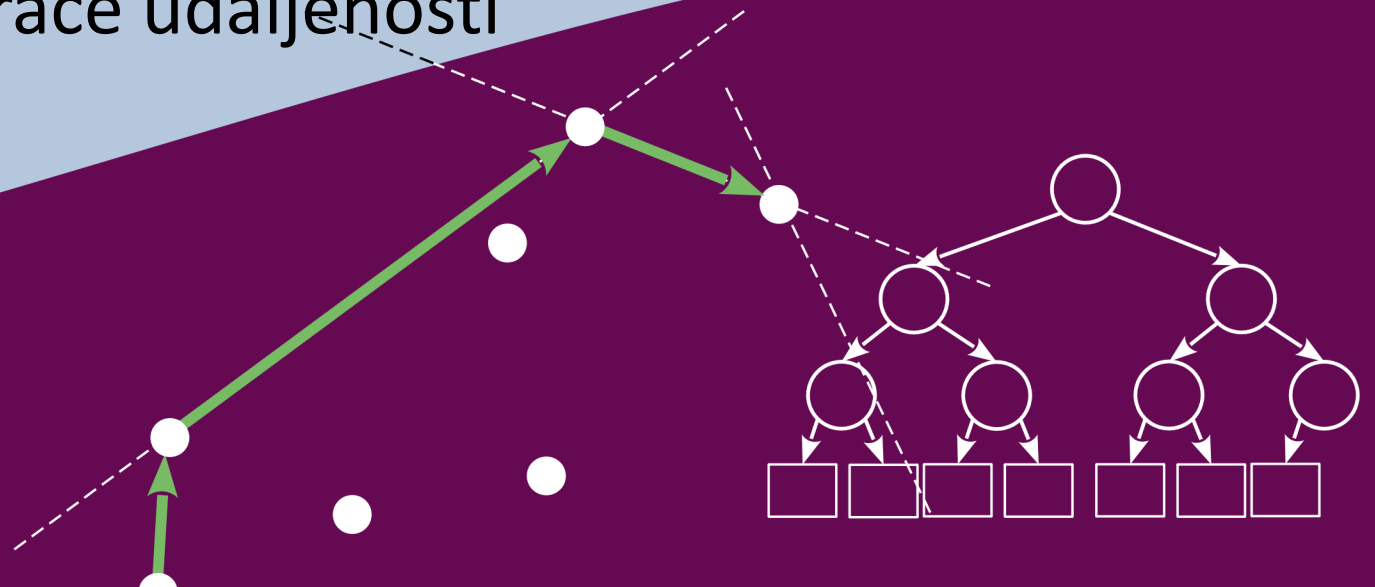
Napredni algoritmi i strukture podataka

Tjedan 9: Algoritmi nad grafovima

Teorija grafova, Najkraće udaljenosti







Basics of graph theory

- **Graph**(graph)is an ordered pairnon-empty of the final set of vertices (vertex)and a set of (possibly empty) edges (edge)

$$= (,)$$

- **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**

Osnove teorije grafova

- **Graf** (graph) je uređeni par nepraznog konačnog skupa vrhova V (vertex) i skupa (moguće praznog) bridova E (edge)

$$G = (V, E)$$

- **Brid** (edge) je element produkta skupa V

$$E \subseteq V \times V, e_k = (v_i, v_j) \in E$$

  - Svaki brid je uređeni par vrhova $e_k = (v_i, v_j)$
  - Brid se kraće može označiti samo $v_i v_j$ ili jednostavno samo e_k, pri čemu je $e_k = v_i v_j$

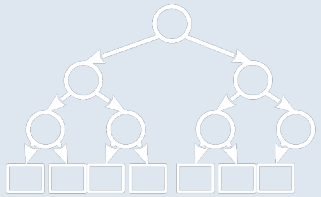
Alternativna literatura:

- **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 | |;often simplified only
- **Size**(size)**graph**is the number of edges that the graph contains | |; 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 ( )
  - Return loops are indeg( )they count twice

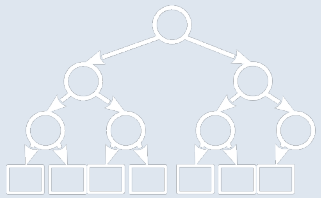
∀ ∈ ∃ $⊆ ∀ ", #(∈ $:)= ∨ #= ⇒ deg = | $| ( )



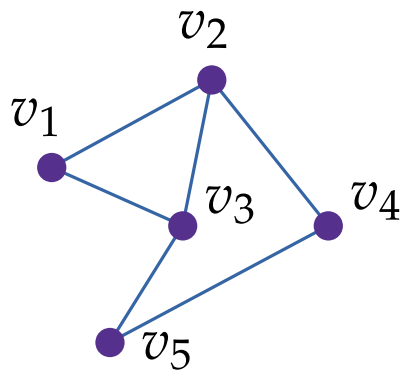
Osnove teorije grafova

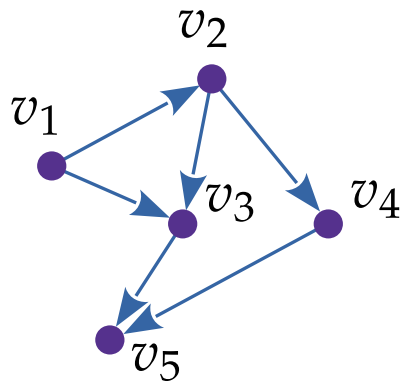
- **Red** (order) **grafa** je broj vrhova u njemu ili $|V|$; često pojednostavljeno samo V
- **Veličina** (size) **grafa** je broj bridova koje graf sadrži $|E|$; pojednostavljeno samo E
- **Petlja** (loop) je brid koji počinje i završava u istom vrhu $e_k = v_i v_i$
- **Stupanj** (degree) **vrha** je broj bridova koji su prileže ći (incident) tom vrhu (spojeni s njim)
  - Stupanj vrha v označavat ćemo s $\deg(v)$
  - Povratne petlje se u $\deg(v)$ ubrajaju dva puta

$$\forall v \in V \exists E' \subseteq E \forall (v_i, v_j) \in E' : v_i = v \vee v_j = v \Rightarrow \deg(v) = |E'|$$



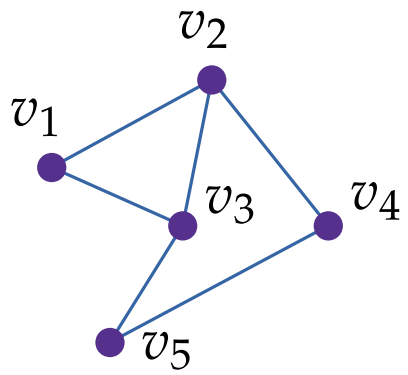
Graph theory - types of graphs

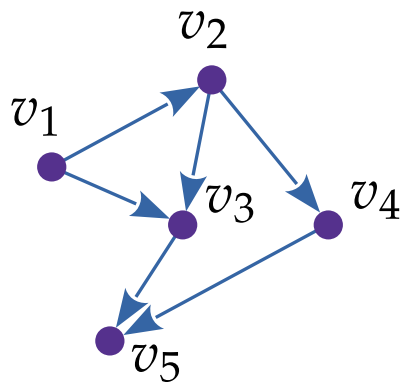




- **Undirected**(undirected)**graph**
  - For each edge " #it's worth it " #= # "
  - Vertices are considered adjacent(adjacent)if it is " #∈
  - Such is the lying edge(incident)peaks " #
- **Directed**(directed)**graph**
  - For the edge " #it doesn't have to be valid " #= # "
  - Top #is considered a neighbor to the peak "only if it exists " #∈
  - Brid " #is called the output edge(outedge)top "and the entrance edge(in edge)top #

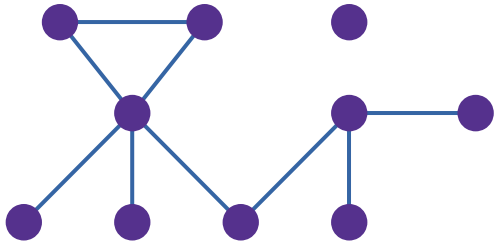
Teorija grafova – vrste grafova

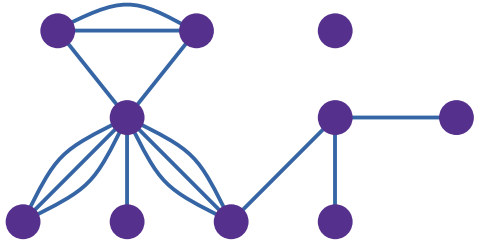


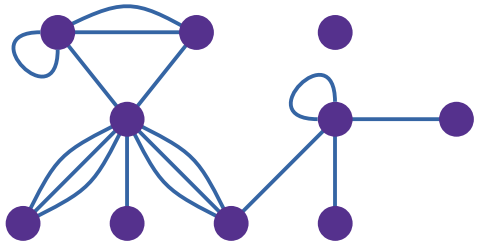


- **Neusmjereni** (undirected) **graf**
  - Za svaki brid $v_i v_j$ vrijedi $v_i v_j = v_j v_i$
  - Vrhovi se smatraju susjednima (adjacent) ako je $v_i v_j \in E$
  - Takav je brid prileže ći (incident) vrhovima $v_i \; i \; v_j$
- **Usmjereni** (directed) **graf**
  - Za brid $v_i v_j$ ne mora vrijediti $v_i v_j \; = v_j v_i$
  - Vrh v_j se smatra susjedom vrhu v_i samo ako postoji $v_i v_j \in E$
  - Brid $v_i v_j$ se naziva izlaznim bridom (outedge) vrha v_i i ulaznim bridom (inedge) vrha v_j

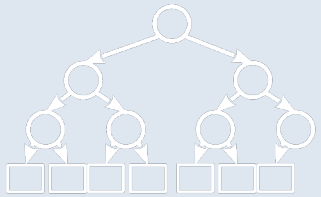
Graph theory - types of graphs



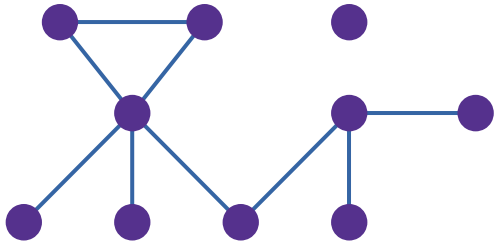


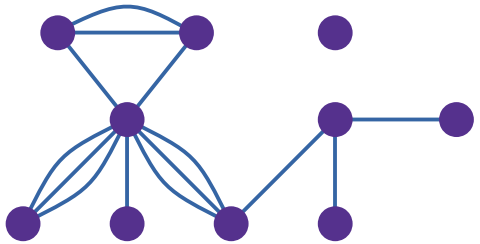


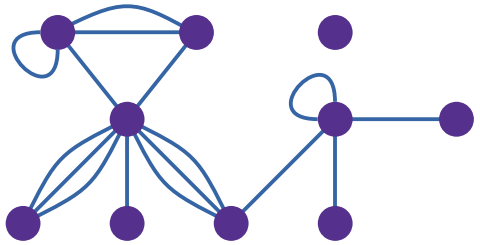
- **A simple graph**(simple graph)is a graph (usually undirected) that has at most one edge between every two vertices and in which there are no loops
- **Multigraph**(multigraph)is a graph that can have more than one edge between two vertices
- **Pseudograph**(pseudograph)is a multigraph in which feedback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



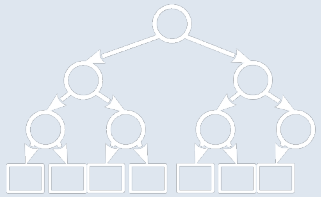
Teorija grafova – vrste grafova



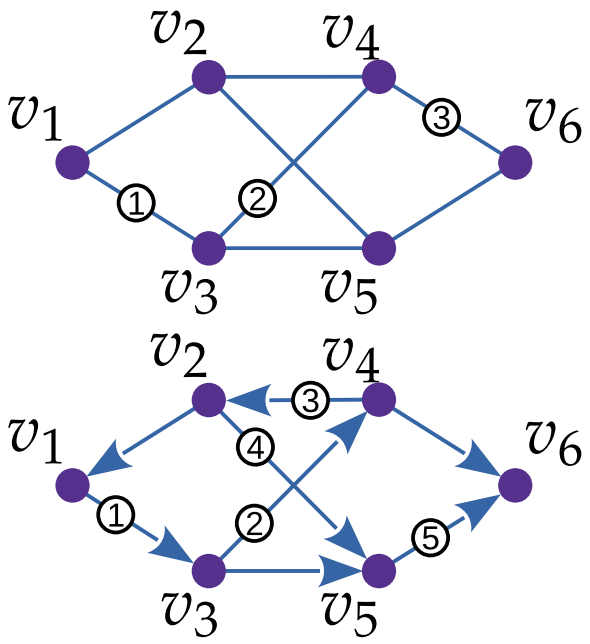




- **Jednostavni graf** (simple graph) je graf (obično neusmjereni) koji između svaka dva vrha ima najviše jedan brid i u kojem nema petlji
- **Multigraf** (multigraph) je graf koji može imati više od jednog brida između dva vrha
- **Pseudograf** (pseudograph) je multigraf u kojem mogu postojati povratne petlje



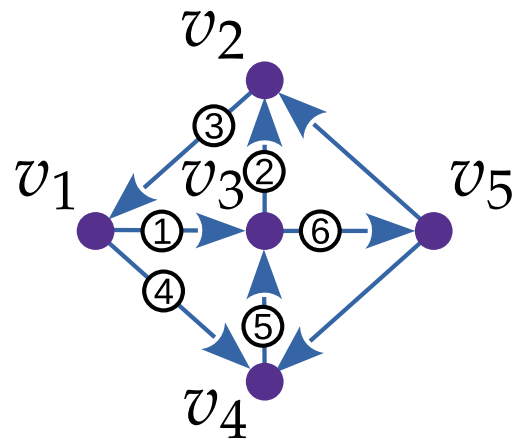
Graph theory - tour, path, ...

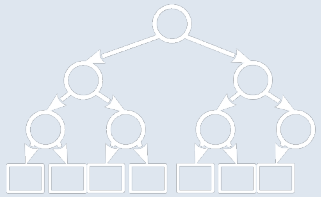


- **Tour or walk**(walk)from the top %to the top & is an alternating series of vertices and edges

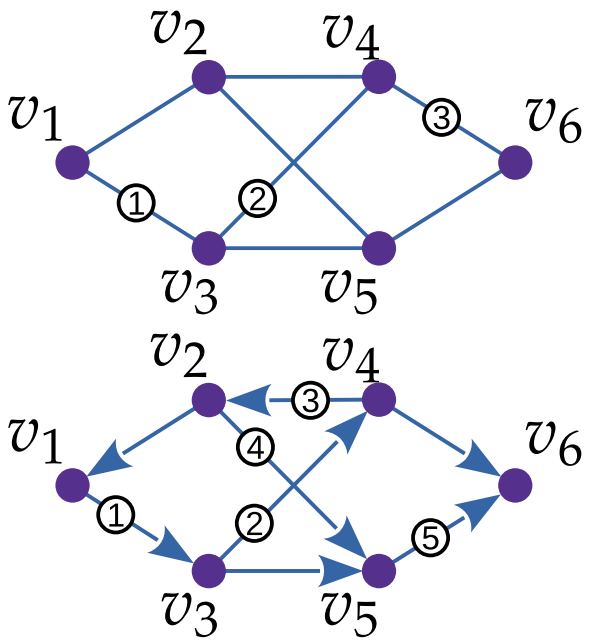
  %, ', ', (,... , &, &

  - 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 %= &





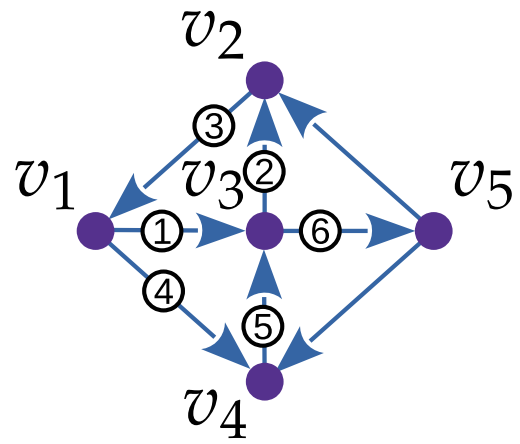
Teorija grafova – obilazak, putanja, ...

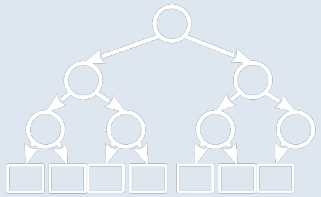


- **Obilazak ili šetnja** (walk) od vrha v_1 do vrha v_n je naizmjenični niz vrhova i bridova

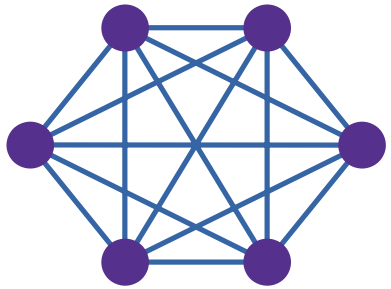
$$v_1, e_2, v_2, e_3, \dots, e_n, v_n$$

  - Kraće se označava $v_1, v_2, \dots, v_n$ ili samo $v_1 v_2 \dots v_n$
- **Putanja** (trail) je obilazak u kojem su svi bridovi različiti (znači svakim bridom se prolazi samo jednom, ali vrhovi se mogu ponavljati)
- **Put ili staza** (path) je putanja (trail) u kojoj su svi vrhovi različiti (dakle ne mogu se ponavljati)
- **Krug** (circuit) je putanja (trail) u kojoj je $v_1 = v_n$
- **Ciklus** (cycle) je staza (path) na kojoj je $v_1 = v_n$





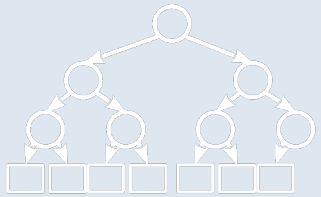
Graph theory



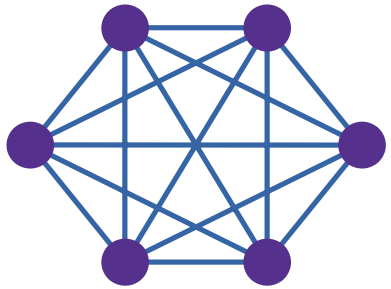
- **Complete**(complete)**graph**is a graph in which there is exactly one edge between every two vertices
  - Applies only to undirected graphs
  - Complete graph*n*-of order (*n*peaks) is denoted by s !
  - The number of edges in the complete graph = the number of binomial subsets of the set of vertices

$$!=(\quad) \quad \binom{}{2} = \frac{!}{(-2!(2!} = \frac{(-1)}{2} = (")$$

- Let it be ′⊂ and ′⊂ ,then it is ' = (', ')**subgraph**(subgraph) graph = (,)
  - If we have "⊂ ,then it is ["]induced subgraph graph ,which contains all edges from the set ,and which connect vertices from the set "



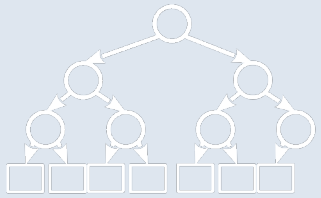
Teorija grafova



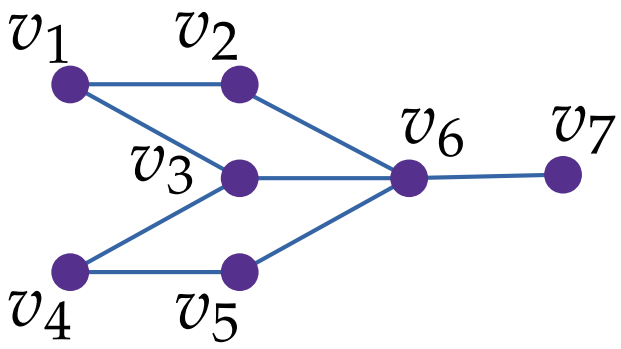
- **Potpuni** (complete) **graf** je graf u kojem je između svaka dva vrha točno jedan brid
  - Odnosi se samo na neusmjerene grafove
  - Potpuni graf *n*-tog reda (*n* vrhova) označava se s K_n
  - Broj bridova u potpunom grafu = broj dvočlanih podskupova skupa vrhova V

$$E(K_n) = \binom{V}{2} = \frac{V!}{(V-2)! \cdot 2!} = \frac{V(V-1)}{2} = O(V^2)$$

- Neka je $V' \subset V$ i $E' \subset E$, tada je $G' = (V', E')$ **podgraf** (subgraph) grafa $G = (V, E)$
  - Ako imamo $V' \subset V$, tada je $G[V']$ inducirani podgraf (induced subgraph) grafa G, koji sadrži sve bridove iz skupa E, a koji spajaju vrhove iz skupa V'



Graph connectivity – undirected graphs



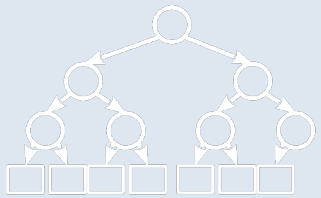
- We define a binary relation 'such that by traversing the graph connects the vertices and .If the vertices and connected is valid

  '

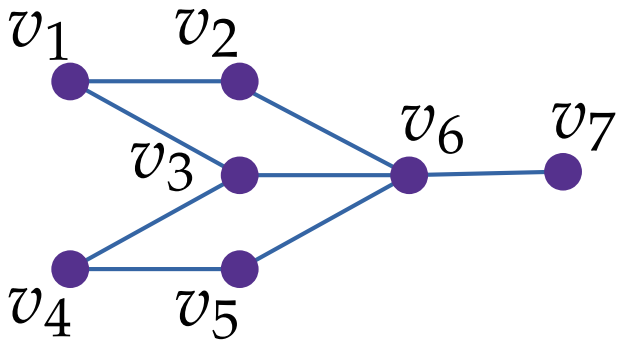
- **A connected undirected graph** = (,) -the undirected graph is connected(connected)only if it holds for all pairs of vertices

  $$\forall\ ,\ \in :\ '\ \wedge\ '$$

  - 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



Povezanost grafa – neusmjereni grafovi



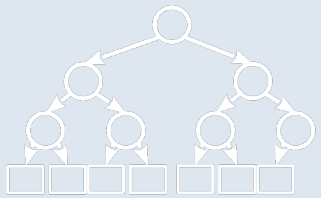
- Definiramo binarnu relaciju W^G takvu da obilaskom kroz graf G povezuje vrhove u i v. Ako su vrhovi u i v povezani vrijedi

$$uW^Gv$$

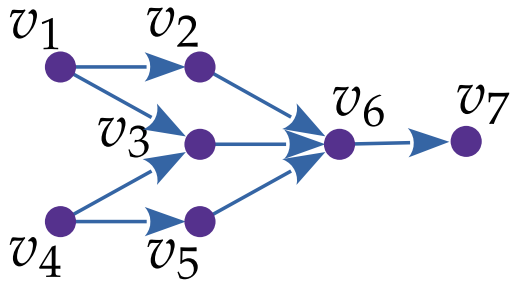
- **Povezani neusmjereni graf** $G = (V, E)$ - neusmjereni graf je povezan (connected) samo ako za sve parove vrhova vrijedi

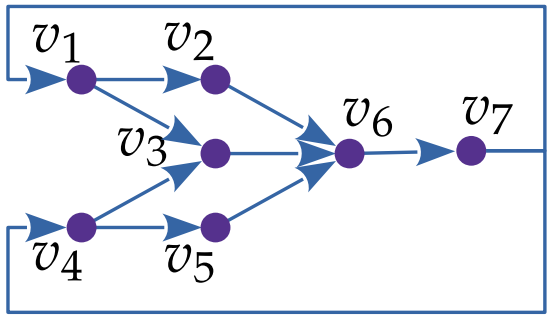
$$\forall u, v \in V : uW^Gv \wedge vW^Gu$$

  - što znači da je vrh v **dohvatljiv** (reachable) od vrha u
  - ali i obratno, što je u sukladnosti sa definicijom neusmjerenog grafa



Graph connectivity – directed graphs



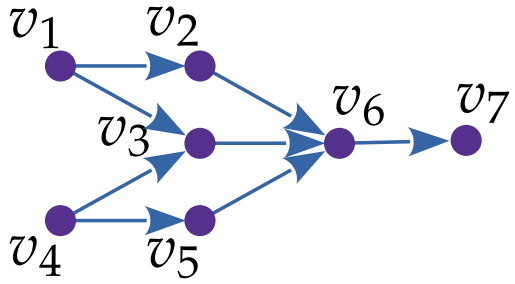


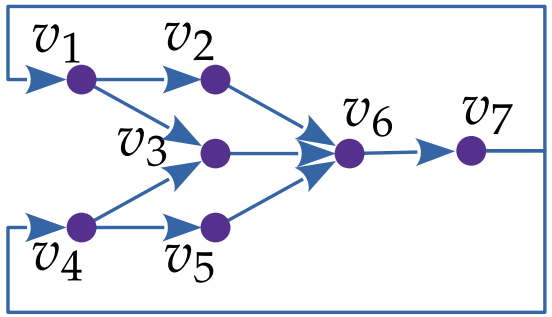
- **Weakly connected directed graph** (weakly connected)–is a directed graph whose undirected equivalent is connected according to the definition in the previous representation
- **A strongly connected directed graph G**(strongly connected)–is a directed graph where all pairs of vertices are mutually reachable

$$\forall , \in : ' \wedge '$$

  - 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

Povezanost grafa – usmjereni grafovi



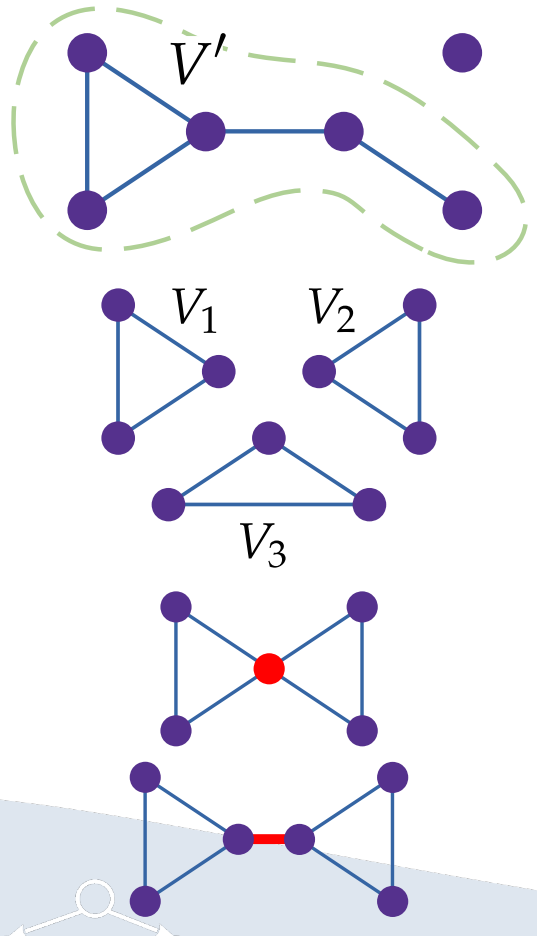


- **Slabo povezani usmjereni graf G (weakly connected)** – je usmjereni graf čiji je neusmjereni ekvivalent povezan prema definiciji na prethodnoj prikaznici
- **Snažno povezani usmjereni graf G (strongly connected)** – je usmjereni graf kod kojeg su svi parovi vrhova međusobno dohvatljivi

$$\forall u, v \in V : uW^G v \wedge vW^G u$$

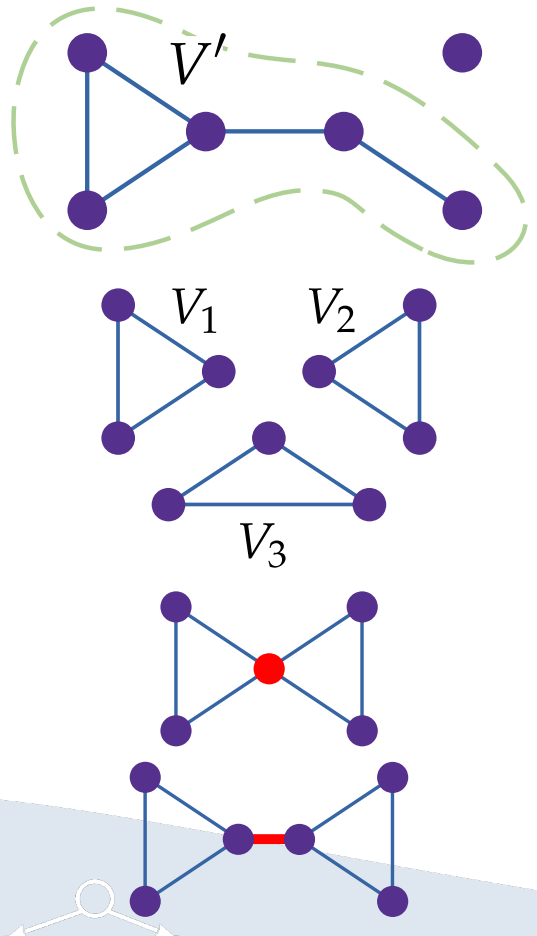
  - Primijetimo da je definicija za snažno povezani usmjereni graf identična definiciji za povezani neusmjereni graf
  - Razlika je i više nego očita s obzirom da se kod usmjerenog grafa stvaraju krugovi i ciklusi da bi on bio snažno povezan

Graph connectivity



- **A disconnected graph** **(disconnected graph)**
  - We define a subset of vertices ′ ⊂ such that is an induced subgraphG[$]connected and there is no tour between vertices in the set $and the rest of the peaks / $
    ∄ ∈ $∄ ∈ /$:) ∨)
  - 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.

Povezanost grafa

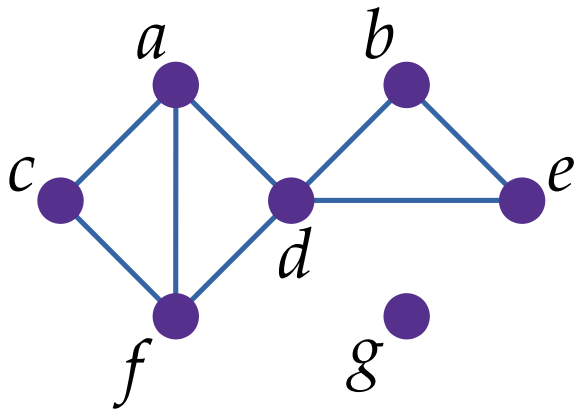


- **Nepovezani graf G (disconnected graph)**
  - Definiramo podskup vrhova $V' \subset V$ takav da je inducirani podgraf $G[V']$ povezan i ne postoji obilazak između vrhova iz skupa V' i ostatka vrhova $V \setminus V'$

$$\nexists u \in V' \nexists v \in V/V' : uW^G v \vee vW^G u$$

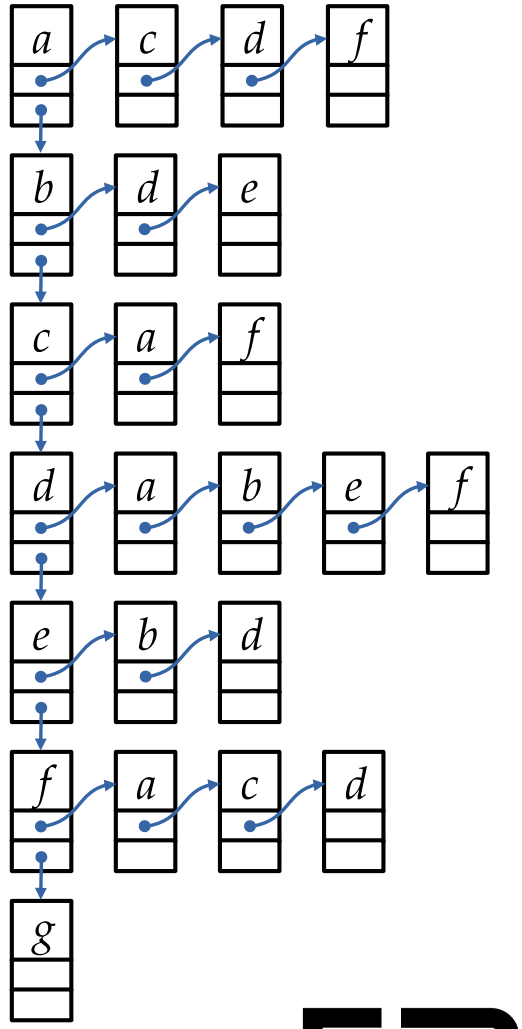
  - Takav se graf smatra nepovezanim
- **Prijelomna točka (articulation point)** – je vrh čijim uklanjanjem graf postaje nepovezan
- **Most (bridge)** – je brid čijim uklanjanjem graf postaje nepovezan.

Graph storage

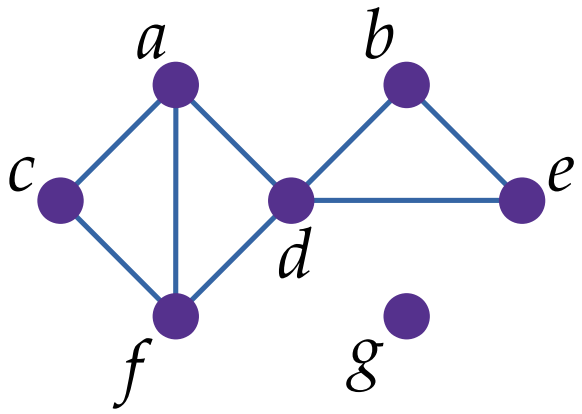


a	c	d	f	
b	d	e		
c	a	f		
d	a	b	e	f
e	b	d		
f	a	c	d	
g				

- 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 necessarydeg( )of steps in relation to | | 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)

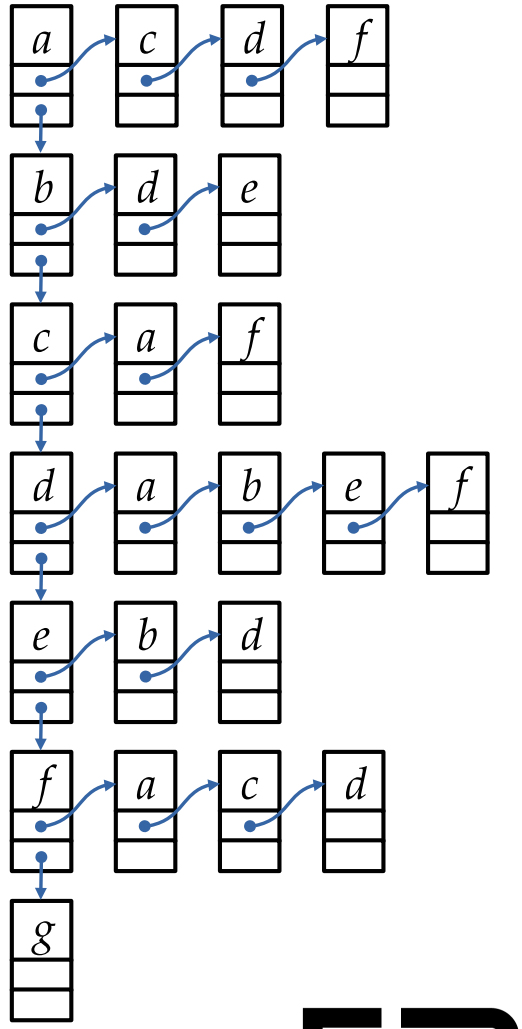


Pohrana grafa

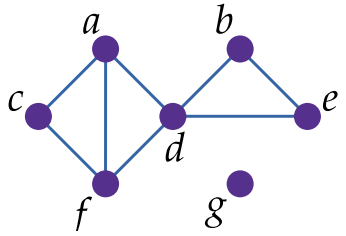


a	c	d	f	
b	d	e		
c	a	f		
d	a	b	e	f
e	b	d		
f	a	c	d	
g				

- Moguća je pohrana pomoću listi i pomoću matrica
- Prednost listi je u pristupu svim susjedima nekog vrha jer je potrebno $\deg(v)$ koraka u odnosu na $|V|$ za matrice
- Prednost matrica je u pojedinačnim intervencijama (dodavanje ili uklanjanje brida) zbog bržeg pristupa i složenosti $O(1)$ prilikom održavanja
- Primjer u obliku tablice (star representation)(lijevo) i u obliku dvodimenzionalne povezane liste (desno)



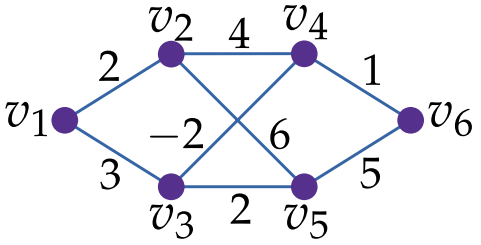
Graph storage



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

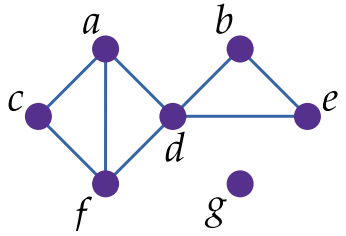
$$\mathbf{B} = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \\ f \\ g \end{array} \begin{array}{c} \begin{array}{cccccccc} ac & ad & af & bd & be & cf & de & df \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1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

- 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)



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

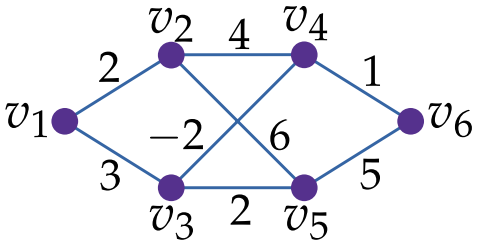
Pohrana grafa



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

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

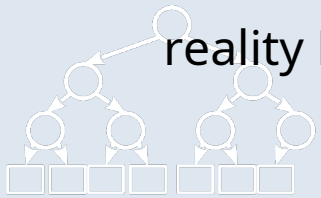
- Primjer u obliku matrice susjedstva (adjacency matrix)(lijevo-sredina)
  - Simetrična za neusmjerene grafove
- U obliku matrice incidencije (incidency matrix)(lijevo-dolje)
- U obliku težinske matrice susjedstva (weighted adjacency matrix)(desno)



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

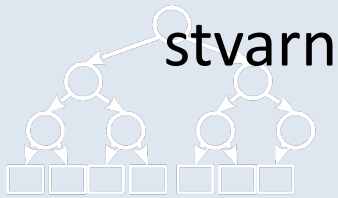
A tour of the graph(graph traversal)

- 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

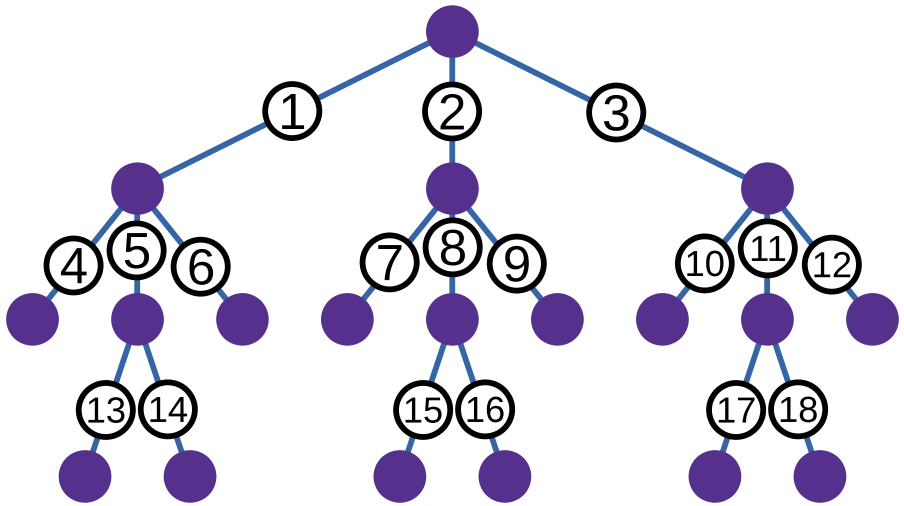


Obilazak grafa (graph traversal)

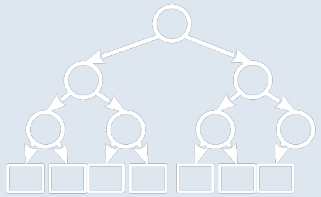
- Algoritmi za obilazak **stabla** nisu zadovoljavajući za općenite grafove jer:
  - Graf može imati cikluse pa bi se algoritam za stablo mogao naći u beskonačnoj petlji
  - Graf može imati odvojene i nepovezane vrhove, pa riskiramo da ne pronađemo sve povezane particije vrhova $P(V)$
- Dva najpoznatija algoritma za obilazak grafa su:
  - Obilazak (prvo) u širinu (*Breadth First Search;* BFS)
  - Obilazak (prvo) u dubinu (*Depth First Search*; DFS)
- Osim u jednostavnim primjenama (npr. obilazak grafa, detekcija ciklusa, provjera povezanosti pojedinih vrhova), DFS i BFS nisu međusobno zamjenjivi zbog bitne logičke različitosti
- Oba su temelj za brojne složenije algoritme i teorijski su podjednako brzi, ali u stvarnosti je DFS nešto sporiji jer je rekurzivan



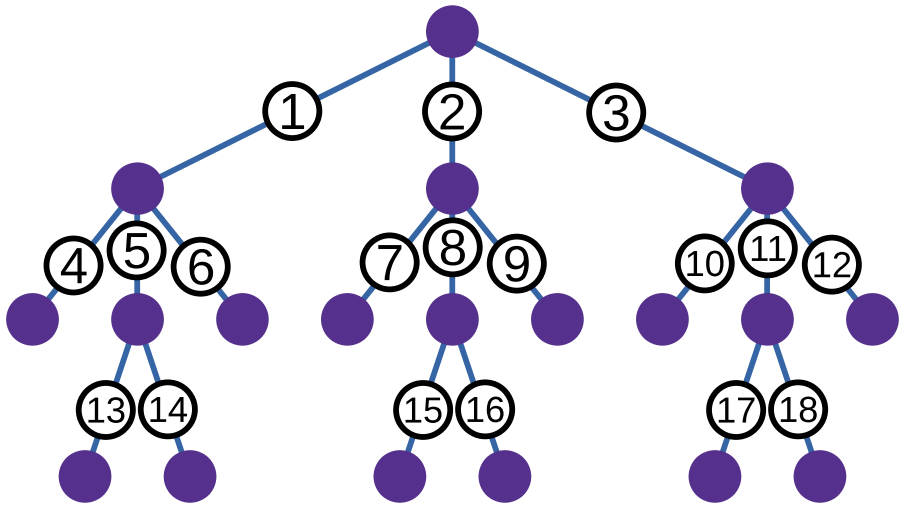
Traversing the graph (first) in width - BFS



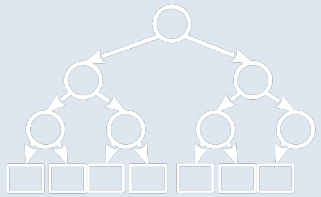
- The concept of the BFS algorithm
  - Initially, we have only the root vertex in the list - the list is a natural data structure for the BFS algorithm
  - We take the first vertex from the list as the current vertex
  - First we go through ALL neighboring vertices of the current vertex
  - As we go 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



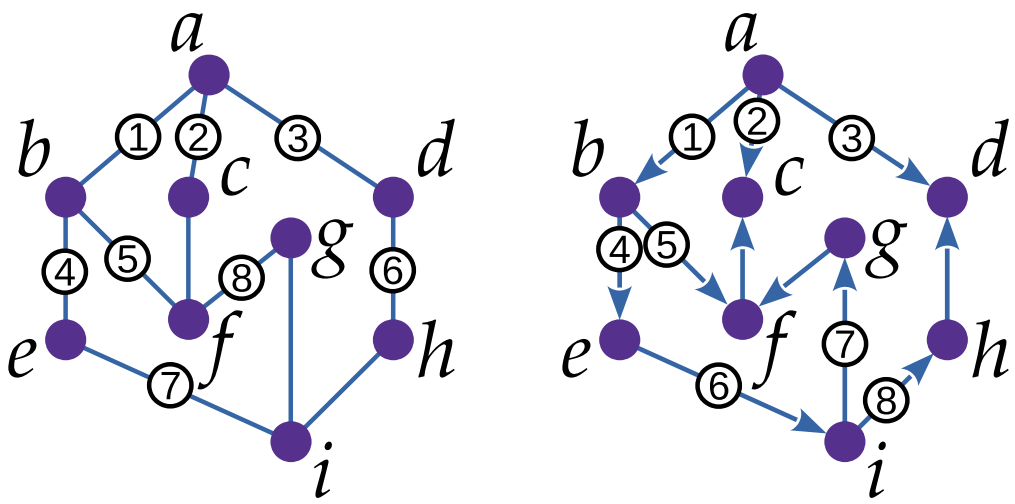
Obilazak grafa (prvo) u širinu - BFS



- Koncept BFS algoritma
  - Inicijalno u listi imamo samo korijenski vrh – lista je prirodna podatkovna struktura za BFS algoritam
  - Uzimamo prvi vrh iz liste kao trenutni vrh n
  - Prvo obilazimo SVE susjedne vrhove trenutnog vrha n
  - Kako obilazimo susjedne vrhove, stavljamo ih u listu
  - Nakon što smo obišli sve susjedne vrhove, uzimamo sljedeći vrh iz liste kao trenutni...
  - Taj postupak ponavljamo do dok nismo obišli sve vrhove u grafu
- Ukoliko imamo nepovezane podgrafove, postupak se ponavlja dok ima neobiđenih vrhova



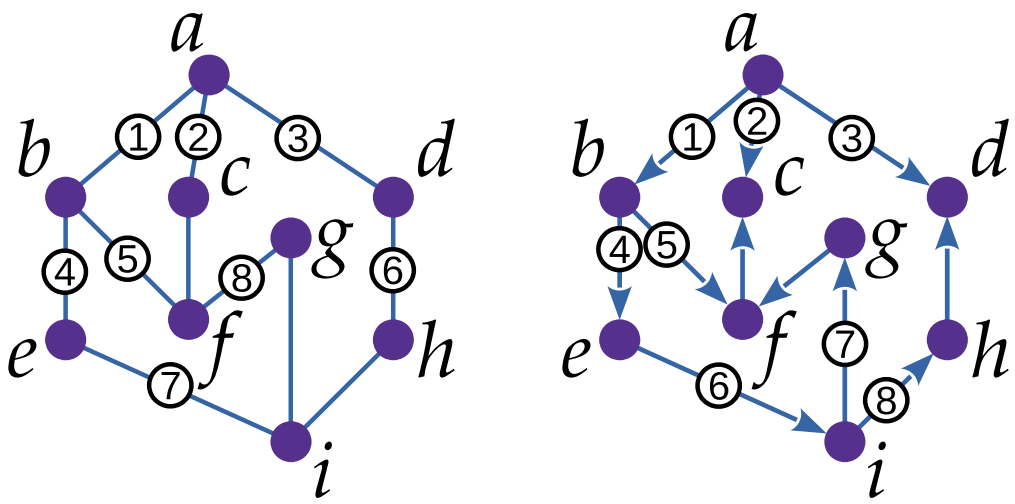
Traversing the graph (first) in width - BFS



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 ← empty queue
    while there is an unvisited vertex u_0 in G do
        mark u_0 as visited
        Q.enqueue(u_0)
        while Q is not empty do
            u ← Q.dequeue
            process u
            for v in adjacent vertices of u do
                if v is not visited then
                    mark v as visited
                    Q.enqueue(v)
```

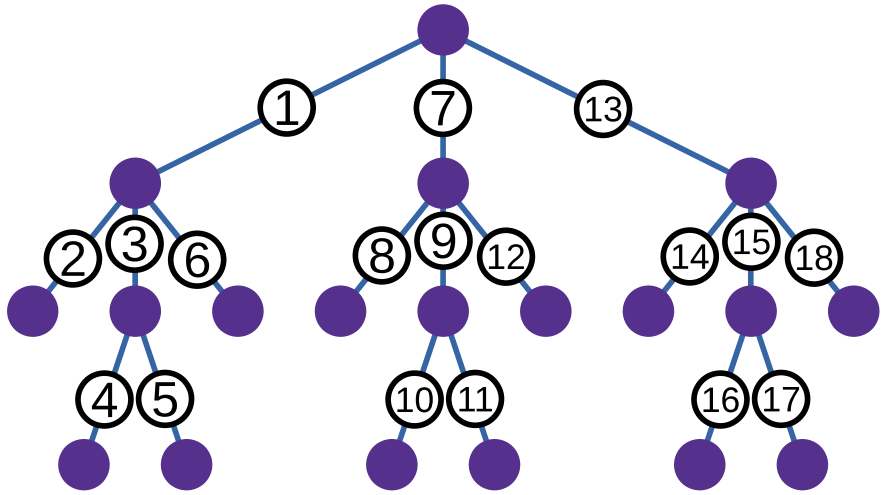
Obilazak grafa (prvo) u širinu - BFS



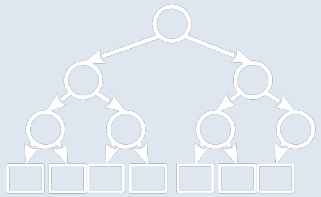
Step	u	queue Q
0		a
1	a	bcd
2	b	$cdef$
3	c	def
4	d	efh
5	e	fhi
6	f	hig
7	h	ig
8	i	g
9	g	

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

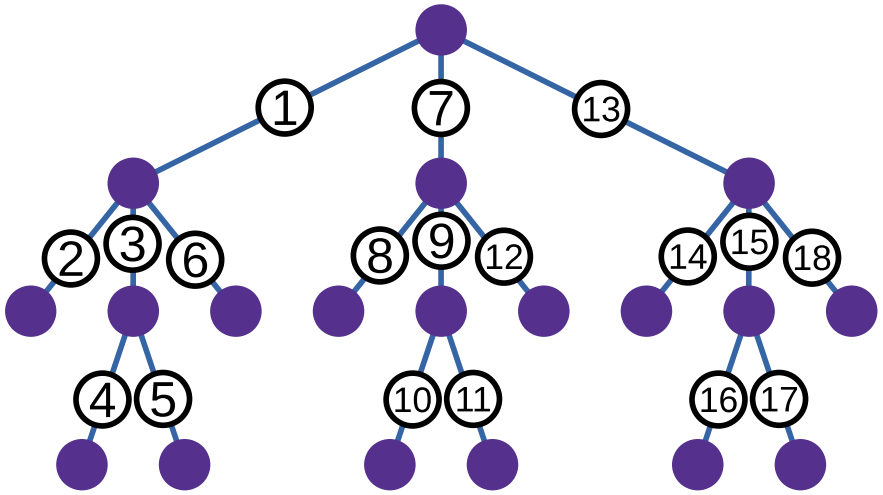
Touring the graph (first) in depth - DFS



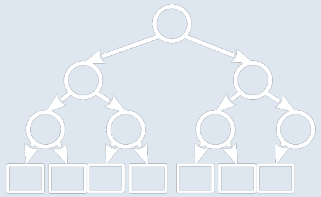
- Concept of DFS algorithm - recursive
  - We take the first unprecedented peak
  - We recursively descend to the first neighbor
  - The recursion is repeated, until we reach a leaf, then we go back to the first vertex where we still have unvisited neighbors, as soon as we start the recursive descent 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



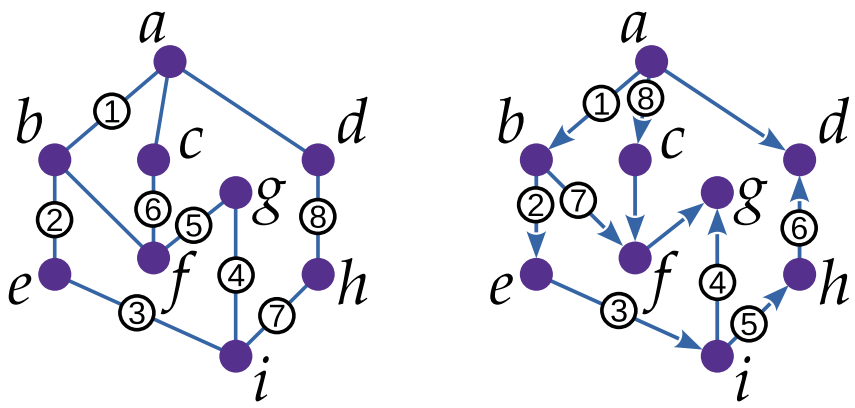
Obilazak grafa (prvo) u dubinu - DFS



- Koncept DFS algoritma - rekurzivno
  - Uzimamo prvi neobiđeni vrh
  - Rekurzivno se spuštamo na prvog susjeda
  - Rekurzija se ponavlja, sve do dok ne dođemo do lista, zatim se vraćamo natrag do prvog vrha na kojem imamo još neobiđenih susjeda, čim opet započinjemo rekurzivno spuštanje do listova
  - Stog je prirodna podatkovna struktura za DFS algoritam
- Ukoliko imamo nepovezane podgrafove, postupak se ponavlja dok ima neobiđenih vrhova



Touring the graph (first) in depth - DFS



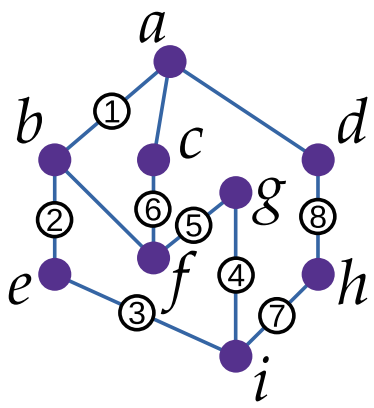
Step	u	visited	stack S
0			a
1	a	No	bcd
2	b	No	$efcd$
3	e	No	$ifcd$
4	i	No	$ghfcd$
5	g	No	$fhfdc$
6	f	No	$chfcd$
7	c	No	$hfcd$
8	h	No	$dfcd$
9	d	No	fcd
10	f	Yes	cd
11	c	Yes	d
12	d	Yes	

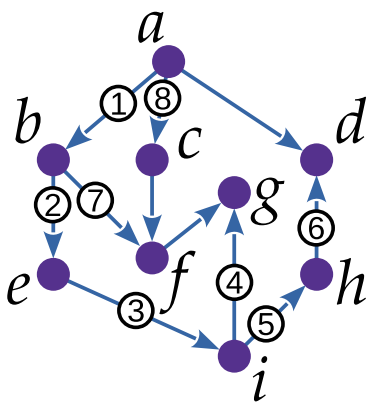
```
procedure DFS(G)
    initialize all vertices in G as not visited
    S ← empty stack
    while there is an unvisited vertex u_0 in G do
        S.push(u_0)
        while S is not empty do
            u ← S.pop
            if u is not visited then
                mark u as visited
                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)
```

Obilazak grafa (prvo) u dubinu - DFS





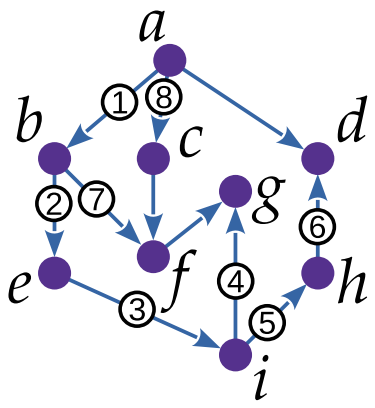
Step	u	visited	stack S
0			a
1	a	No	bcd
2	b	No	$efcd$
3	e	No	$ifcd$
4	i	No	$ghfcd$
5	g	No	$fhfdc$
6	f	No	$chfcd$
7	c	No	$hfcd$
8	h	No	$dfcd$
9	d	No	fcd
10	f	Yes	cd
11	c	Yes	d
12	d	Yes	

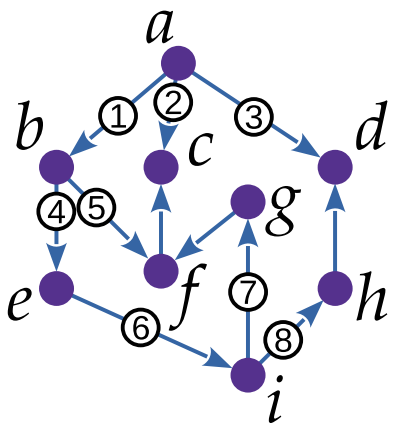
```
procedure DFS(G)
    initialize all vertices in G as not visited
    S ← empty stack
    while there is an unvisited vertex u_0 in G do
        S.push(u_0)
        while S is not empty do
            u ← S.pop
            if u is not visited then
                mark u as visited
               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

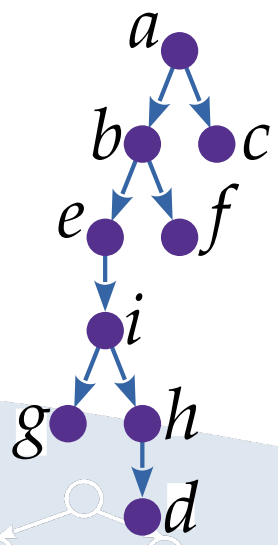
```
procedure DFS(G)
    initialize all vertices in G as not visited
    while there is an unvisited vertex u_0 in G do
        DFS_recursive(G, u_0)

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)
```

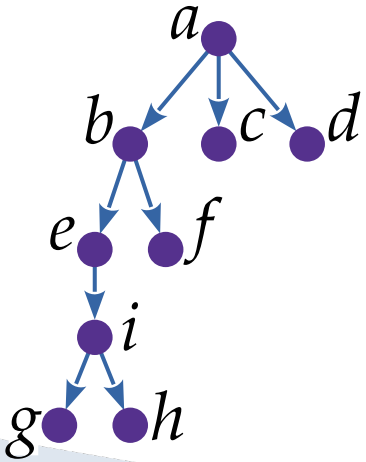
Crucifixion tree(spanning tree)







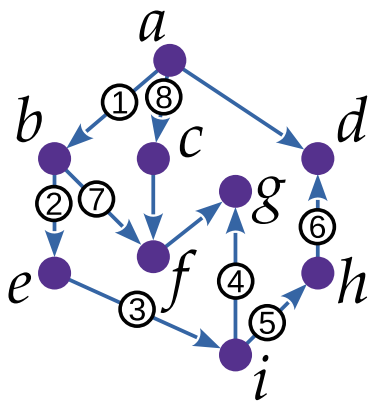
DFS

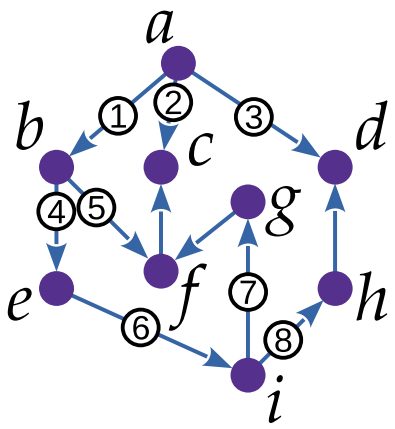


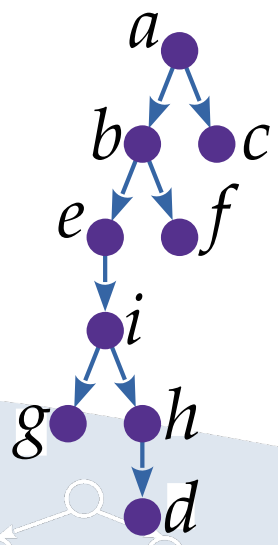
BFS

- 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)

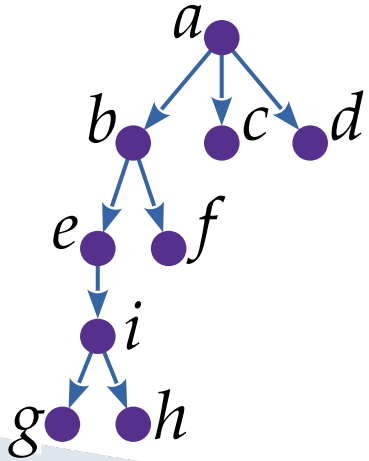
Razapinjujuće stablo (spanning tree)







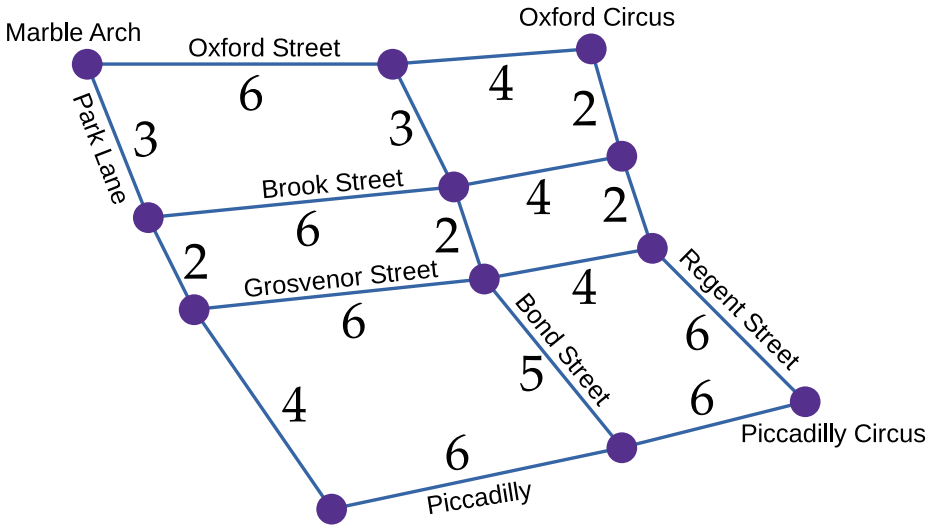
DFS



BFS

- Rezultanta putanja oba algoritma za obilazak grafa (BFS i DFS) čini stablo u kojem su svi vrhovi grafa. Takvo se stablo naziva **razapinjujuće stablo** (spanning tree).
  - Bilježimo samo bridove koji predstavljaju **napredovanje** BFS i DFS algoritama prema neobiđenim vrhovima – **unaprjedni bridovi** (forward edges)
  - Ne bilježimo bridove kojima se algoritmi vraćaju u vrhove koji su već obiđeni – **povratni bridovi** (back edges)

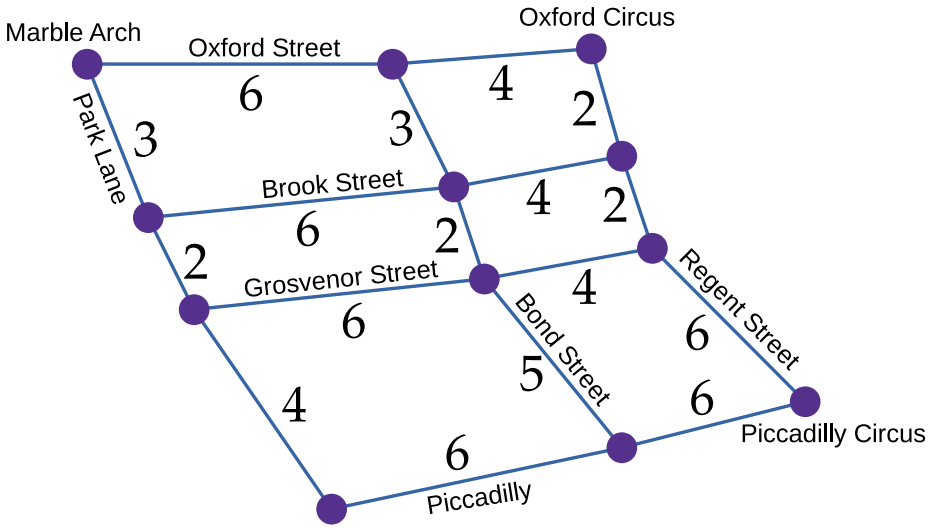
Shortest paths in the graph(shortest paths)



- 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)

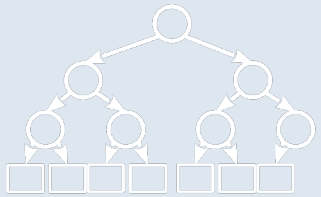
Najkraći putevi u grafu (shortest paths)



- Temeljni algoritmi za brojne primjene: transport, komunikacije, distribucijske energetske mreže, projektiranje integriranih elektroničkih sklopova i drugo...
- Bridovi, kao apstrakcija (model) poveznica između dvaju čimbenika nekog sustava, dobivaju "težine"; oznaka $w(u, v)$
- Važan je i pojam međuvrh; to je vrh na putu između neka dva vrha u grafu, dakle ni polazni ni završni
- Labela (label) = udaljenost vrha od nekog referentnog vrha (točke)
  - Najčešće se pohranjuje kao članska varijabla strukture 'vrh'
- Dvije osnovne grupe algoritama:
  - Algoritmi koji pronalaze najkraći put između dva određena vrha
  - Algoritmi koji pronalaze najkraće puteve između svih vrhova u grafu (all-to-all)

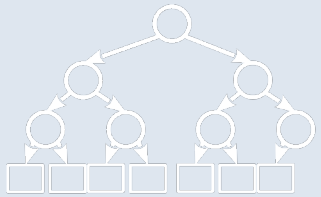
Shortest paths in the graph

- 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

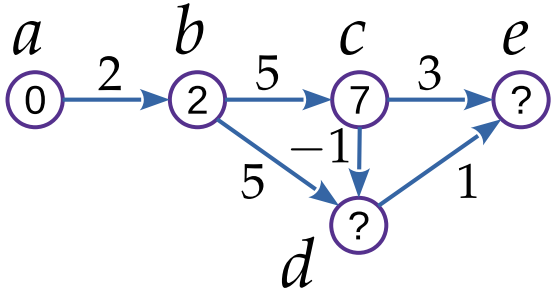


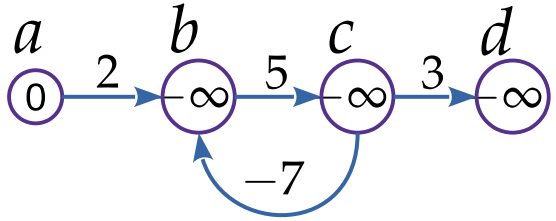
Najkraći putevi u grafu

- Odabir algoritma za traženje najkraćeg puta ovisi o težinama bridova u grafu; dvije su osnovne skupine tih algoritama:
  - **Label-setting** algoritmi – jednom upisana labela se više ne mijenja
    - Tako upisana labela se više ne provjerava
    - Funkcioniraju za grafove s isključivo pozitivnim težinama bridova – Dijkstrin algoritam
  - **Label-correcting** algoritmi – sve labele se mogu mijenjati sve do završetka postupka (konvergencije)
    - Funkcioniraju za grafove s bilo kakvim težinama bridova – Bellman-Ford algoritam



Shortest paths in the graph – negative weights

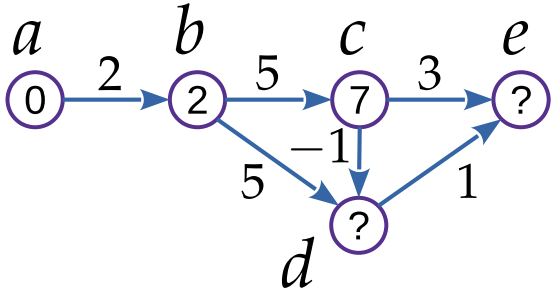


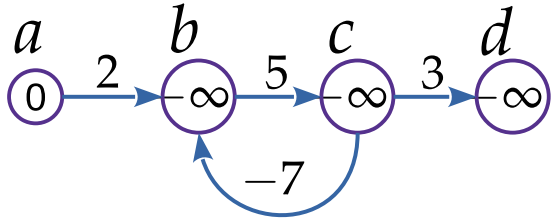


Iteration	$d(b)$	$d(c)$
1	2	7
2	0	5
3	-2	3
4	-4	1
5	-6	-1
...	...	...
∞	$-\infty$	$-\infty$

- 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

Najkraći putevi u grafu – negativne težine





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

- Negativna težina predstavlja problem za **label-setting** algoritme – uzmimo primjer na slici
  - Ako smo do vrha d došli obilaskom abd, tada je njegova labela $d(d) = 7$
  - Ako smo do vrha d došli obilaskom $abcd$, tada je njegova labela $d(d) = 6$
- Drugi problem predstavlja negativni ciklus, koji predstavlja problem i za **label-correcting** algoritme
  - Algoritam se uplete u beskonačnu petlju ažuriranja labela
  - U trenutku kada bi algoritam trebao konvergirati, on ne konvergira nego nastavlja ažurirati labele
  - Takav problem je nerješiv

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

$$= \; - /\ldots .$$

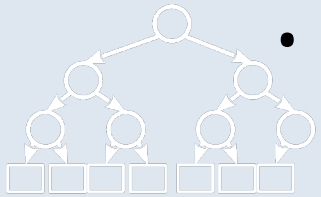
  - the concept of distance additivity can be applied

$$(\#, \; !) = \; (\#, \; \$) + \; (\$, \; !)$$

- 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



Dijkstrin algoritam

- Spada u algoritme koji računaju najkraću udaljenost između dva vrha v_1 i v_n
- Label-setting algoritam
  - Jednom ažurirana labela vrha se više ne mijenja, osim kroz drugu putanju
  - Nije u mogućnosti raditi s negativnim težinama bridova
- Za neku putanju

$$p = v_1 v_2 \dots v_n$$

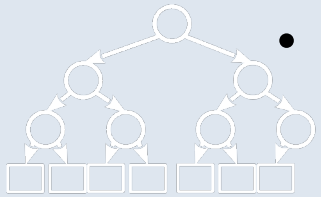
  - može se primijeniti koncept aditivnosti udaljenosti

$$d(v_1, v_n) = d(v_1, v_i) + d(v_i, v_n)$$

- Ako je putanja p ujedno i najkraća, tada vrijedi

$$d_{min}(v_1, v_n) = d_{min}(v_1, v_i) + d_{min}(v_i, v_n)$$

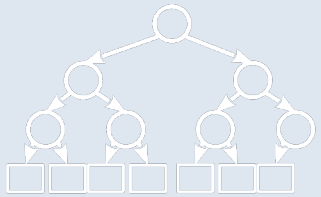
- Gledajući unatrag, ako svaki vrh na najkraćem putu "zna" (barem) svojeg neposrednog prethodnika, može se rekonstruirati cijeli put do polaznog vrha.
  - Ideja je računati $d(v_1, v_n) = d(v_1, v_{j-1}) + d(v_{j-1}, v_j)$, od od $j = n$ do $j = 2$



Dijkstra's algorithm

```
procedure DIJKSTRA(G, source, destination)
    for each vertex v in V(G) do
        d(v) ← ∞
        predecessor(v) ← null
    d(source) ← 0
    work ← V(G)
    while work in not empty do
        u ← 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 ← d(u) + w(uv)
            if temp < d(v) then
                d(v) ← temp
                predecessor(v) ← u
```

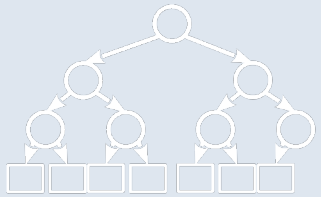
- We initialize the labels of all vertices except the initial one at∞,and predecessors on*null*.We initialize the initial peak to 0.
- We start from the initial vertex, and calculate and update the distances to all neighbors
- During the tour
  - If the neighbor's distance is smaller than the current 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



Dijkstrin algoritam

```
procedure DIJKSTRA(G, source, destination)
    for each vertex v in V(G) do
        d(v) ← ∞
        predecessor(v) ← null
    d(source) ← 0
    work ← V(G)
    while work in not empty do
        u ← 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 ← d(u) + w(uv)
            if temp < d(v) then
                d(v) ← temp
                predecessor(v) ← u
```

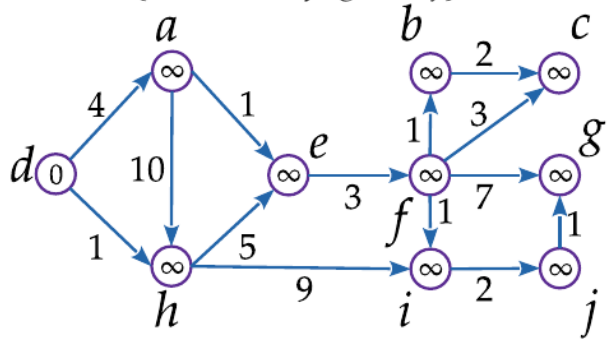
- Inicijaliziramo labele svih vrhova osim početnoga na ∞, te prethodnike na *null.* Početni vrh inicijaliziramo na 0.
- Krenemo od početnog vrha, te računamo i ažuriramo udaljenosti prema svim susjedima
- Prilikom obilaska
  - Ako je udaljenost susjeda manja od trenutne, tada ažuriramo labelu i tom susjedu stavljamo novog prethodnika
- Obilazak se temelji na BFS algoritmu koji je prioritiziran udaljenošću – manje udaljeni čvorovi idu prvi
- Završavamo u završnom vrhu



Dijkstra's algorithm - an example

Iteration 0: The input graph G and initialization

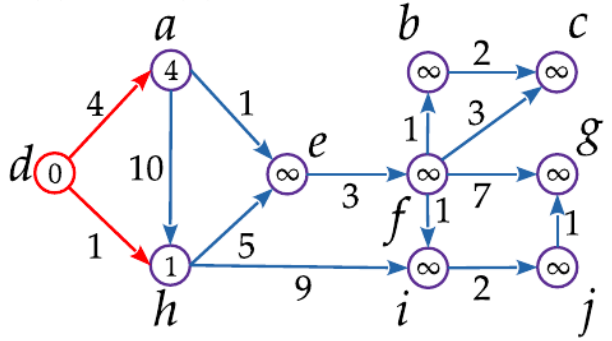
$work = \{a, b, c, d, e, f, g, h, i, j\}$



Iteration 1: $u = d$.

$work = \{a, b, c, e, f, g, h, i, j\}$

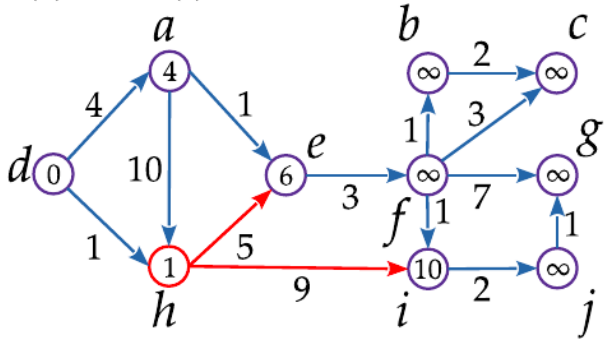
$d(a) = 4, d(h) = 1$



Iteration 2: $u = h$.

$work = \{a, b, c, e, f, g, i, j\}$

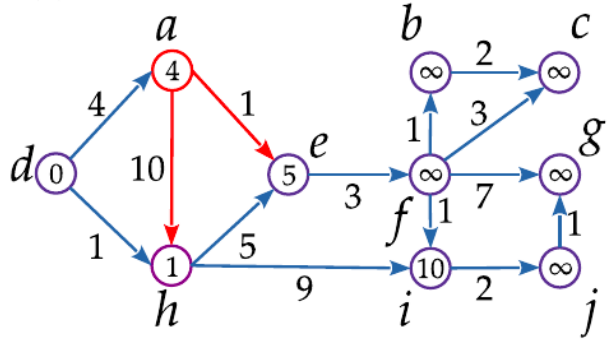
$d(e) = 6, d(i) = 10$



Iteration 3: $u = a$

$work = \{b, c, e, f, g, i, j\}$

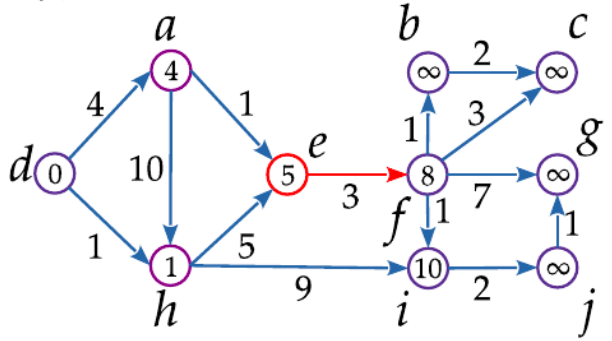
$d(e) = 5$



Iteration 4: $u = e$.

$work = \{b, c, f, g, i, j\}$

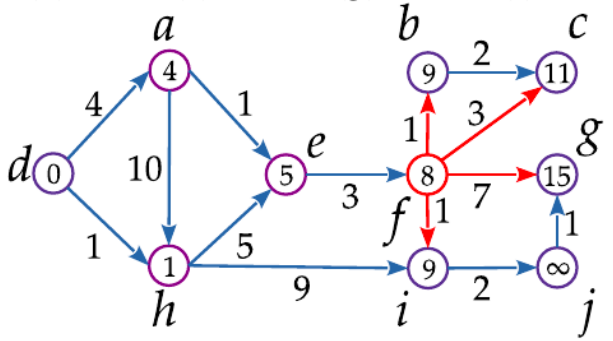
$d(f) = 8$



Iteration 5: $u = f$.

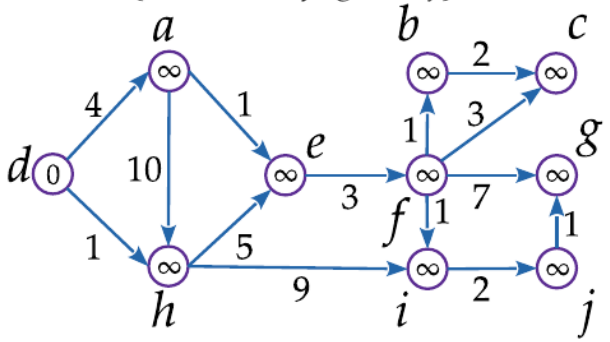
$work = \{b, c, g, i, j\}$

$d(b) = 9, d(c) = 11, d(g) = 15, d(i) = 9$

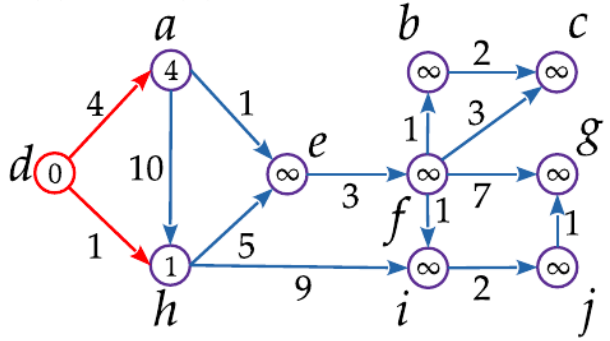


Dijkstrin algoritam - primjer

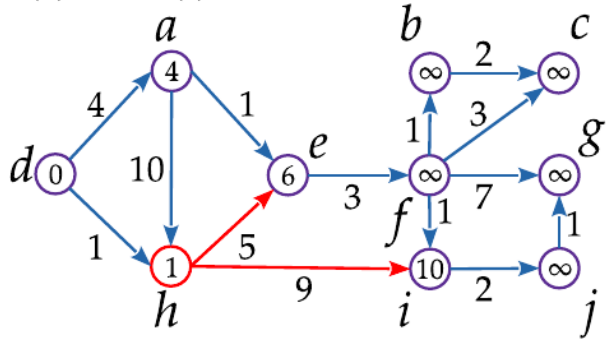
Iteration 0: The input graph G and initialization
$work = \{a, b, c, d, e, f, g, h, i, j\}$



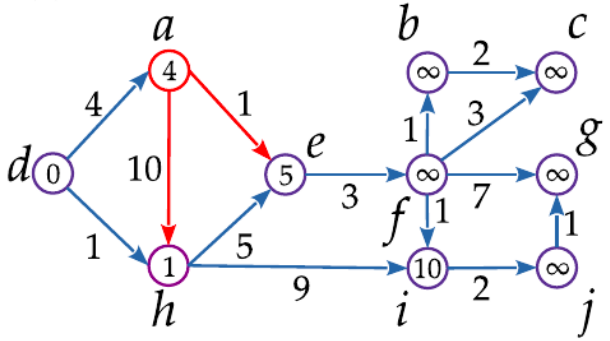
Iteration 1: $u = d$.
$work = \{a, b, c, e, f, g, h, i, j\}$
$d(a) = 4, d(h) = 1$



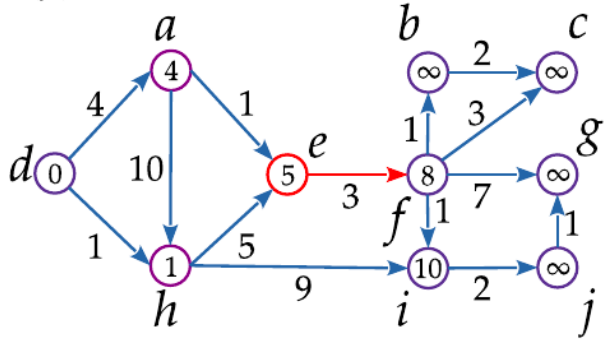
Iteration 2: $u = h$.
$work = \{a, b, c, e, f, g, i, j\}$
$d(e) = 6, d(i) = 10$



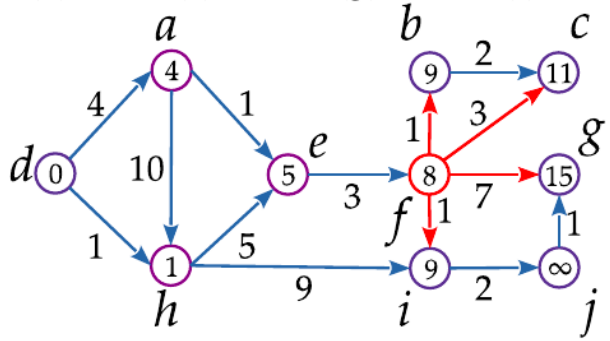
Iteration 3: $u = a$
$work = \{b, c, e, f, g, i, j\}$
$d(e) = 5$



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

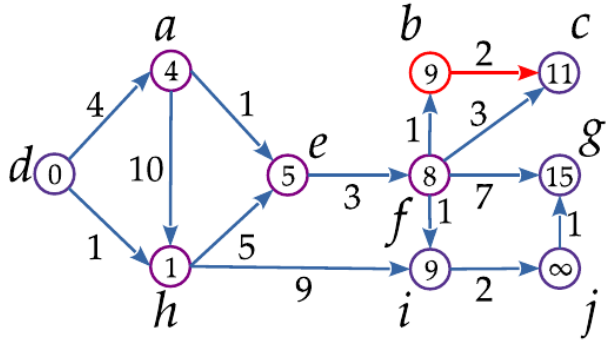


Iteration 5: $u = f$.
$work = \{b, c, g, i, j\}$
$d(b) = 9, d(c) = 11, d(g) = 15, d(i) = 9$

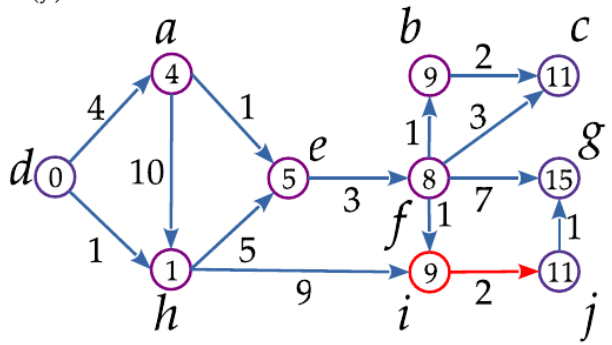


Dijkstra's algorithm - an example

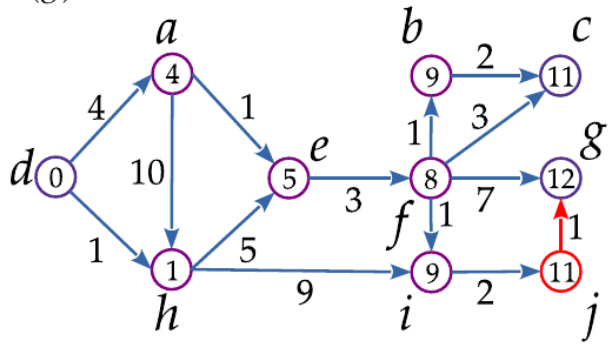
Iteration 6: $u = b$
$work = \{c, g, i, j\}$



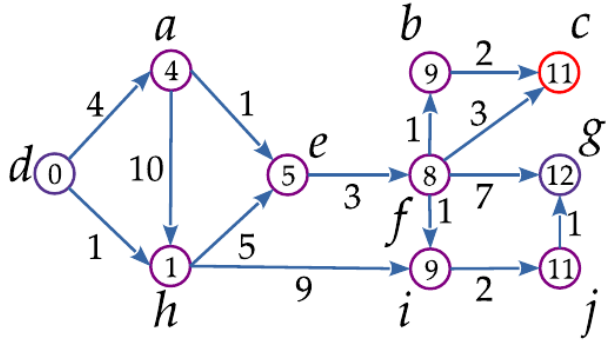
Iteration 7: $u = i$.
$work = \{c, g, j\}$
$d(j) = 11$



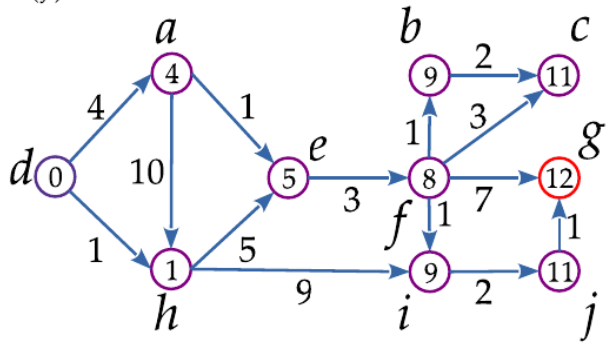
Iteration 8: $u = j$.
$work = \{c, g\}$
$d(g) = 12$



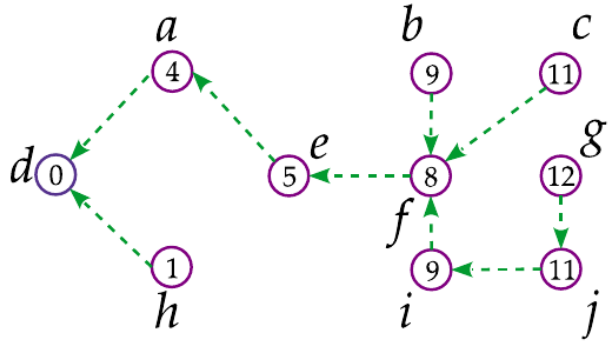
Iteration 9: $u = c$
$work = \{g\}$



Iteration 10: $u = g$.
$work = \{\}$
$d(j) = 11$

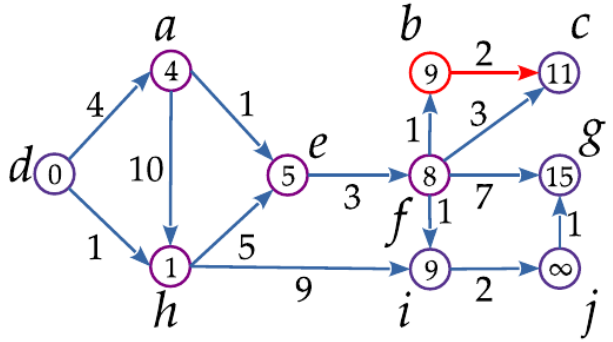


Final predecessors in the graph G.

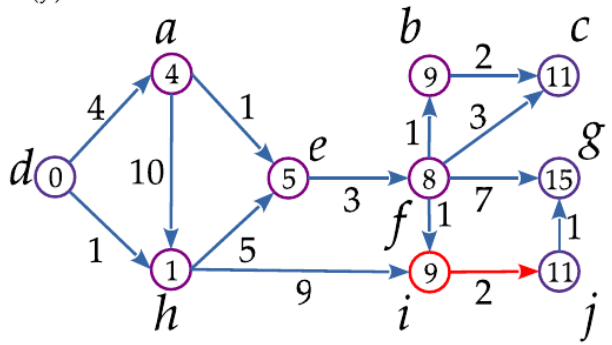


Dijkstrin algoritam - primjer

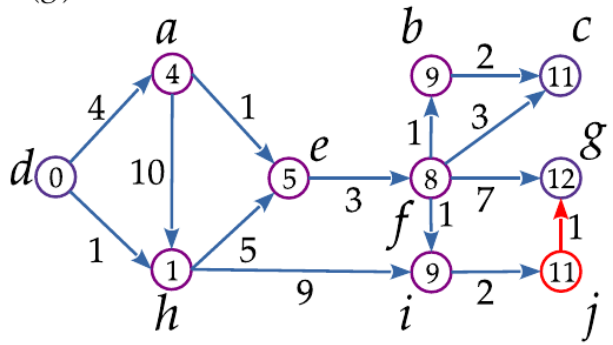
Iteration 6: $u = b$
$work = \{c, g, i, j\}$



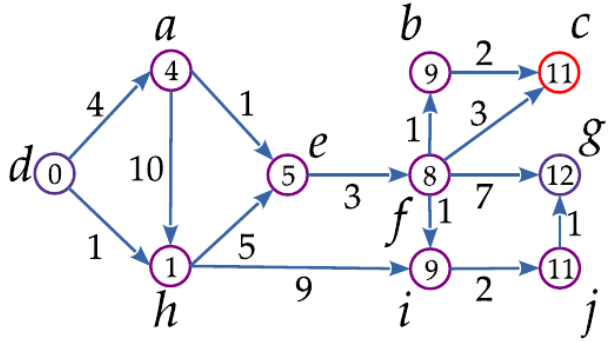
Iteration 7: $u = i$.
$work = \{c, g, j\}$
$d(j) = 11$



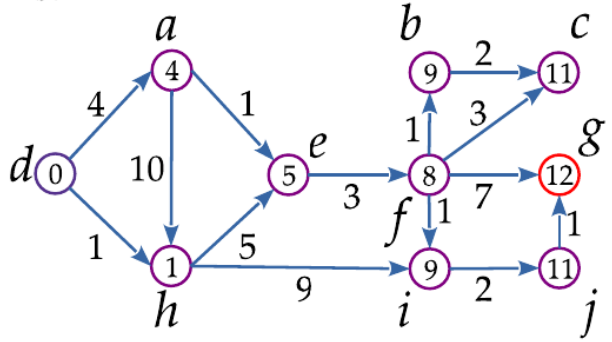
Iteration 8: $u = j$.
$work = \{c, g\}$
$d(g) = 12$



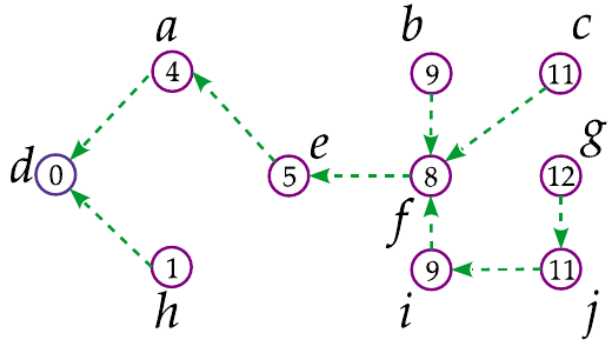
Iteration 9: $u = c$
$work = \{g\}$



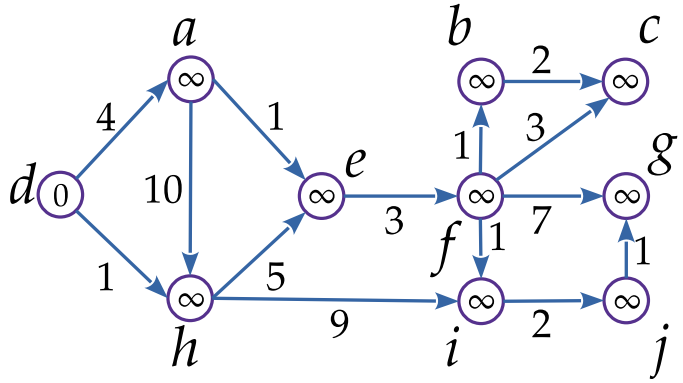
Iteration 10: $u = g$.
$work = \{\}$
$d(j) = 11$



Final predecessors in the graph G.



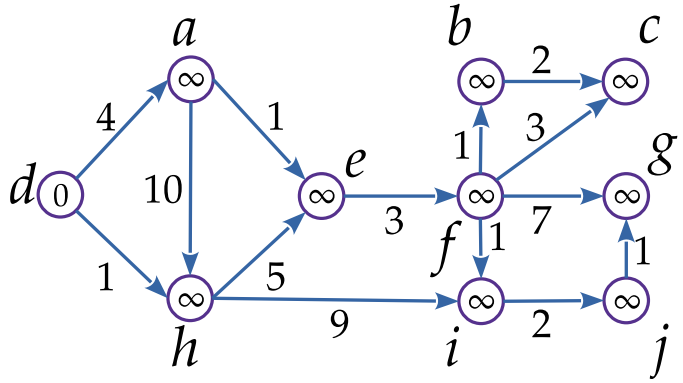
Dijkstra's algorithm - an example



→ → i → f → → →

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

Dijkstrin algoritam - primjer

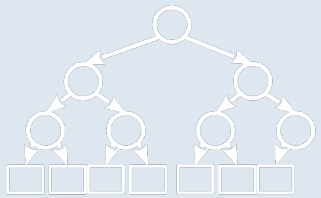


$$g \to j \to \mathrm{i} \to \mathrm{f} \to e \to a \to d$$

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

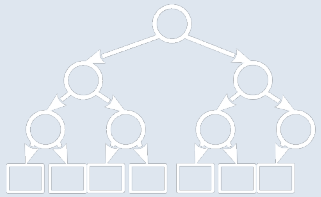
Dijkstra's algorithm - conclusion

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



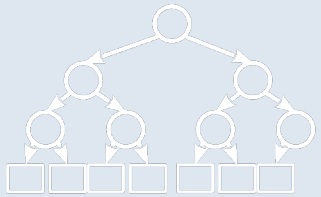
Dijkstrin algoritam - zaključak

- Kompleksnost algoritma ako se najbliži vrh iz *work* liste određuje sekvencijalno – $O(V^2)$
- Ukoliko se *work* lista implementira kao Fibonacci-eva gomila (Fibonacci heap), tada kompleksnost pada na $O(E + V \log_2 V)$



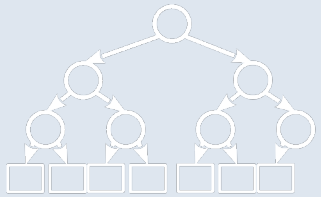
Bellman-Ford algorithm

- 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

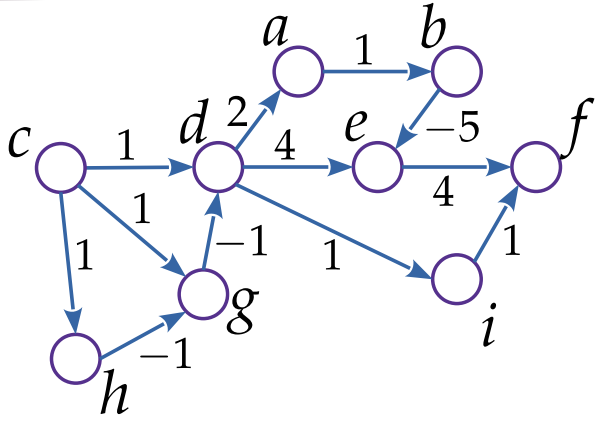


Bellman-Ford algoritam

- Spada u algoritme koji računaju najkraću udaljenost između početnog vrha i svih ostalih vrhova
- Label-correcting algoritam
  - Sve labele se ažuriraju do trenutka konvergencije – do dok više nema daljnjeg ažuriranja labela
  - Može raditi s negativnim težinama u grafu
  - Nije u mogućnosti raditi s negativnim ciklusima
- Sporiji od Dijkstrinovog algoritma
- Radi s bridovima
  - Provjerava sve bridove u grafu i po njima ažurira udaljenosti vrhova
  - Konvergencija je postavljena na:
    - Dok više nema ažuriranja labela na vrhovima
    - Programski limit je $|V| - 1$ iteracija kroz sve bridove grafa



Bellman-Ford algorithm

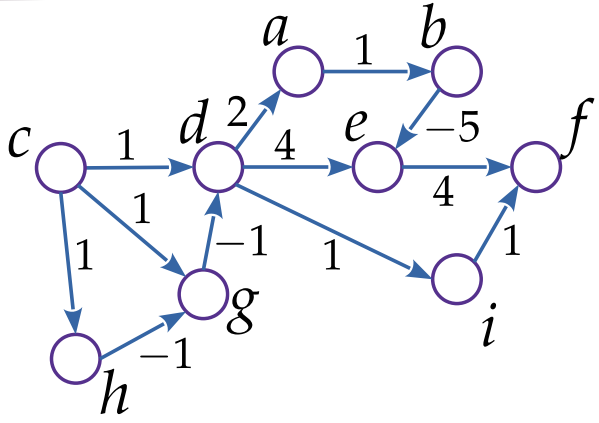



```
procedure BELLMAN-FORD(G, source)
    for each vertex v in V(G) do
        d(v) ← ∞
        predecessor(v) ← null
    d(source) ← 0
    loop |V(G)| − 1 times
        for each edge uv ∈ E(G) do
            if d(u) + w(uv) < d(v) then
                d(v) ← d(u) + w(uv)
                predecessor(v) ← u
    for each edge uv ∈ 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
  ( ) = { , , , , h, , , , , , h , }
- We pass through the edges in ( ).For the edge we update the vertex distance if we have

  ( ) < (( )
- We repeat this as much as possible − 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 algoritam



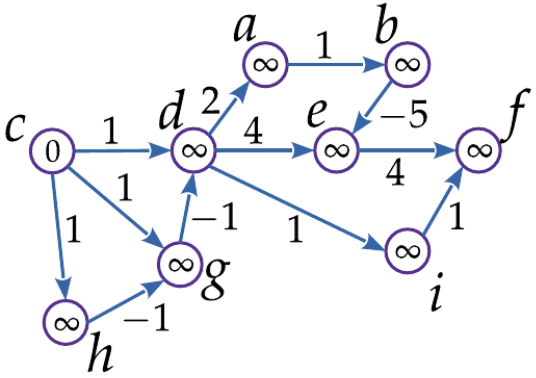

```
procedure BELLMAN-FORD(G, source)
    for each vertex v in V(G) do
        d(v) ← ∞
        predecessor(v) ← null
    d(source) ← 0
    loop |V(G)| − 1 times
        for each edge uv ∈ E(G) do
            if d(u) + w(uv) < d(v) then
                d(v) ← d(u) + w(uv)
                predecessor(v)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

- Sortiramo listu bridova grafa na neki način, na primjer
  $E(G) = \{ab, be, cd, cg, ch, da, de, di, ef, gd, hg, if\}$
- Prolazimo kroz bridove u $E(G)$. Za brid uv ažuriramo udaljenost vrha v ako imamo
  $$d(u) + w(uv) < d(v)$$
- To ponavljamo maksimalno $|V| - 1$ puta
- Na kraju prođemo još jednom kroz sve bridove, pa ako još uvijek imamo vrh čiju labelu možemo ažurirati – negativni ciklus

Bellman-Ford algorithm - example

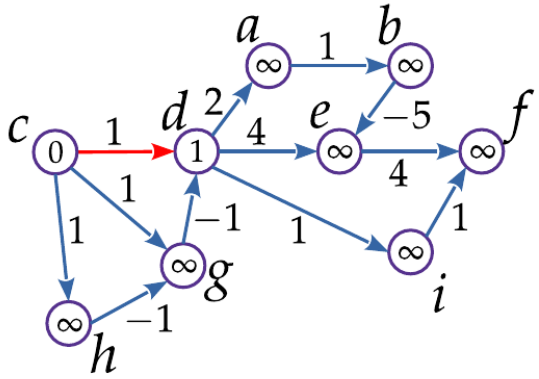
Outer loop iteration 1

Iteration 0: The input graph G and initialization

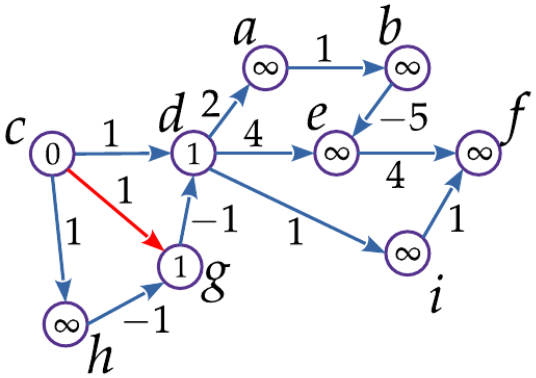


Iterations 1 and 2 are ineffective, since edges ab and be do not update any vertex distances.

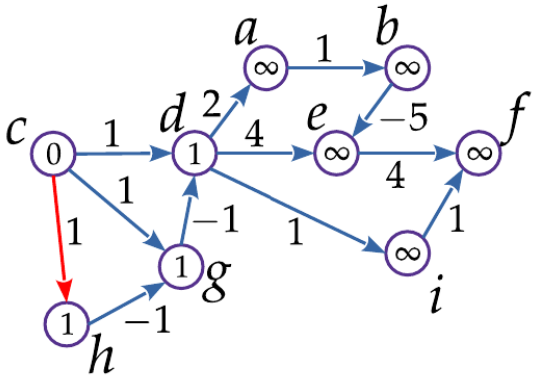
Iteration 3: The edge cd.



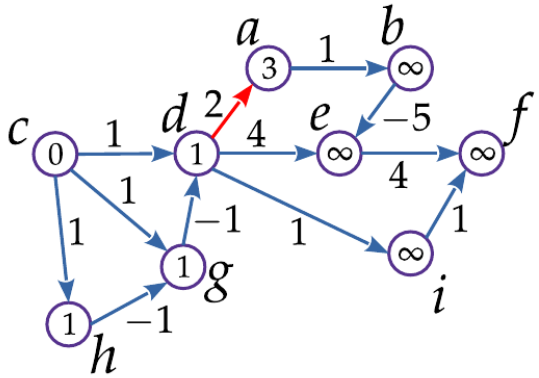
Iteration 4: The edge cg



Iteration 5: The edge ch

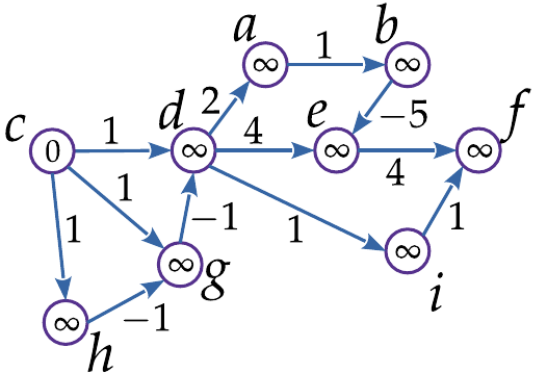
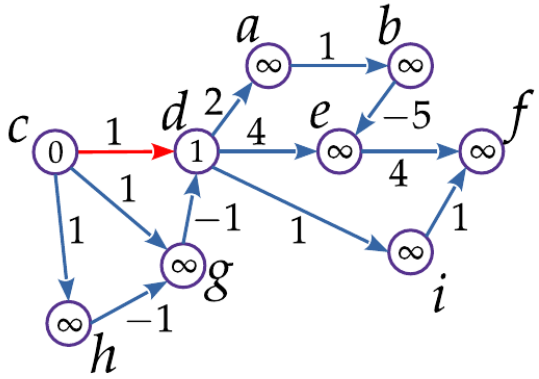
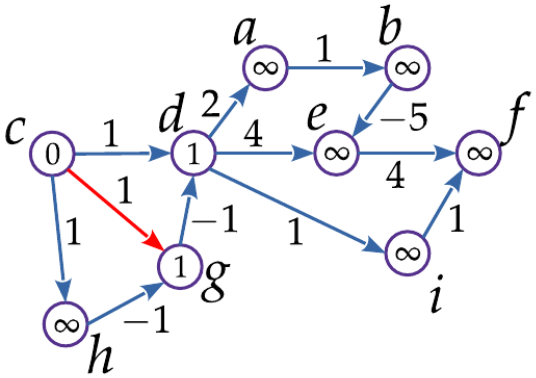
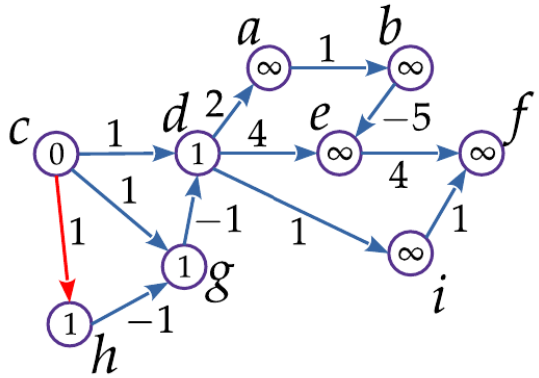
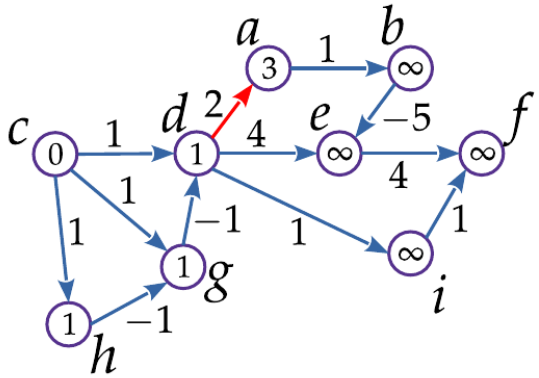


Iteration 6: The edge da.



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

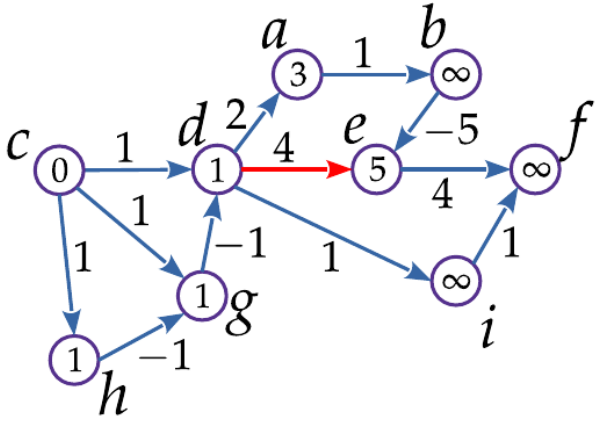
Bellman-Ford algoritam - primjer

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.
		

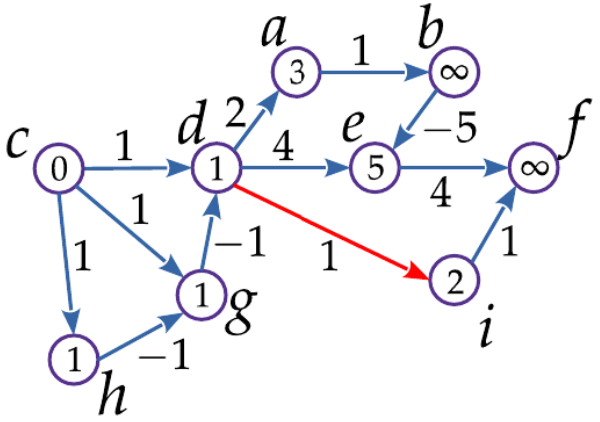
$E(G) = \{ab, be, cd, cg, ch, da, de, di, ef, gd, hg, if\}$

Bellman-Ford algorithm - example

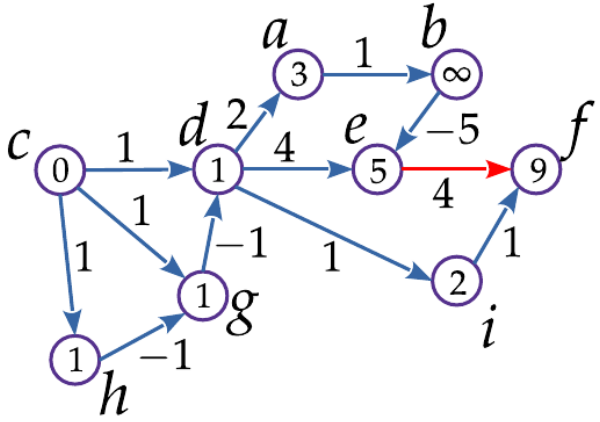
Iteration 7: The edge de



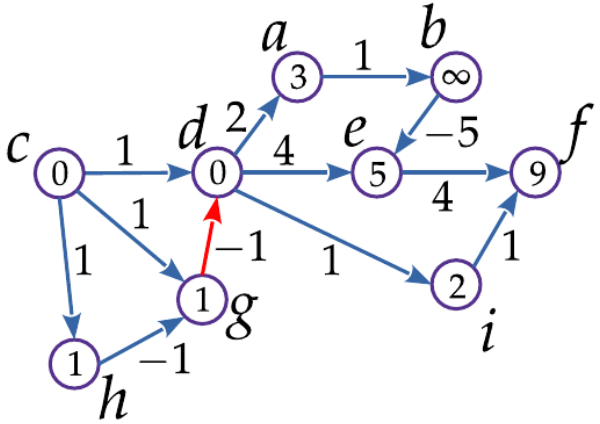
Iteration 8: The edge di



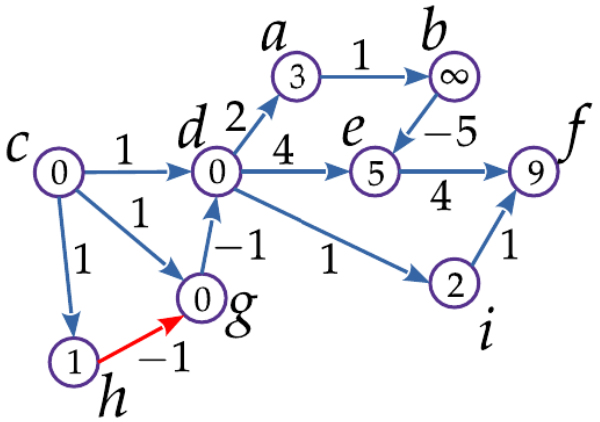
Iteration 9: The edge ef.



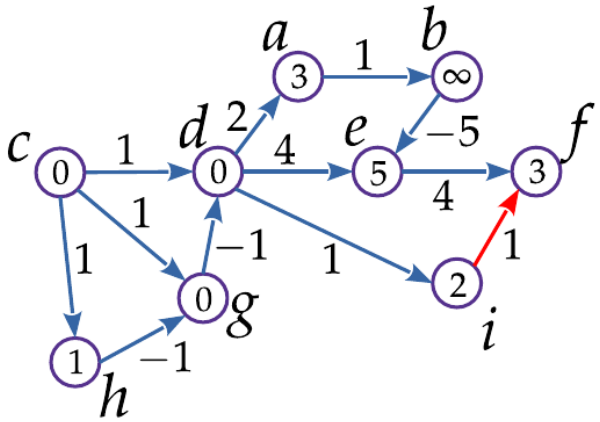
Iteration 10: The edge gd



Iteration 11: The edge hg

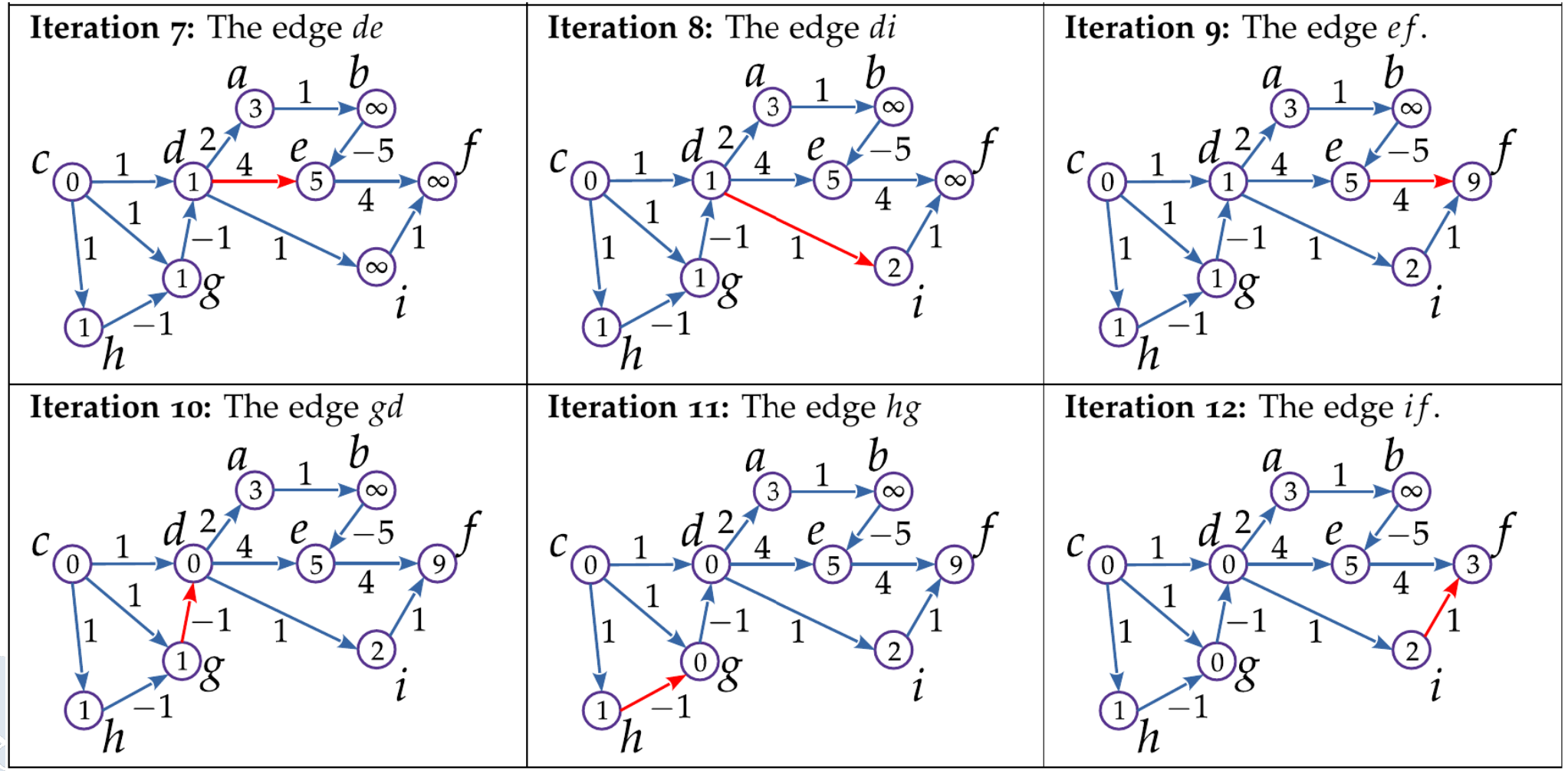


Iteration 12: The edge if.



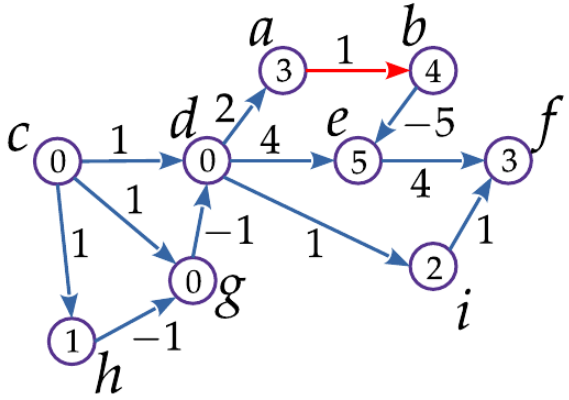
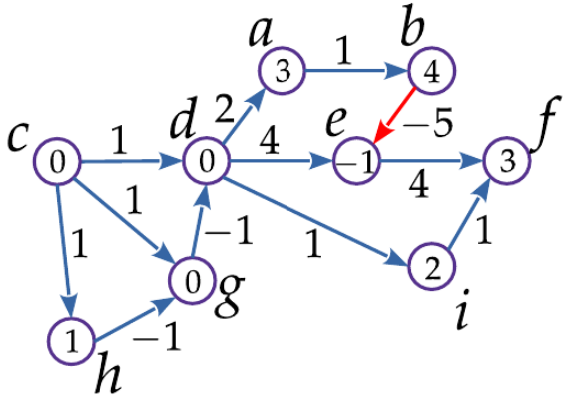
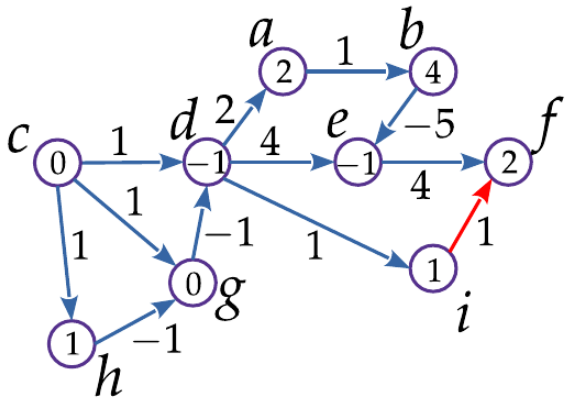
( ) = { , , , , h, , , , , , , h , }

Bellman-Ford algoritam - primjer



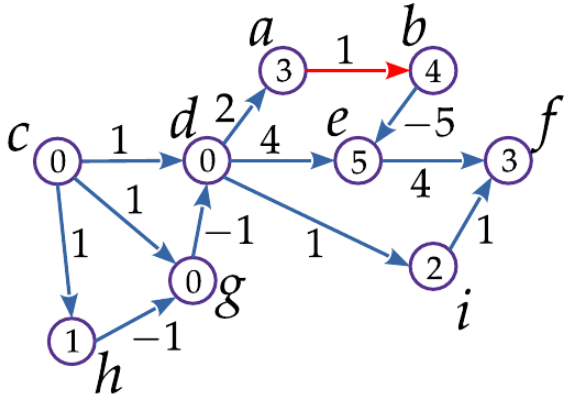
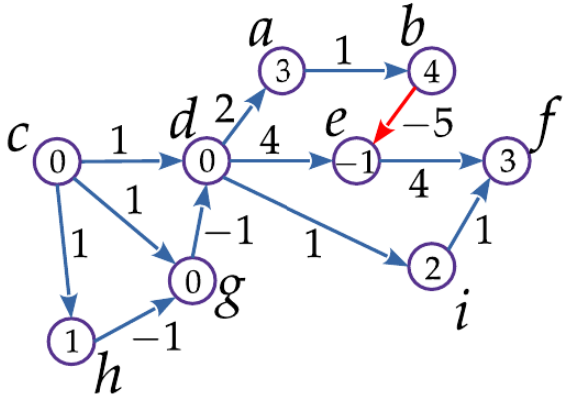
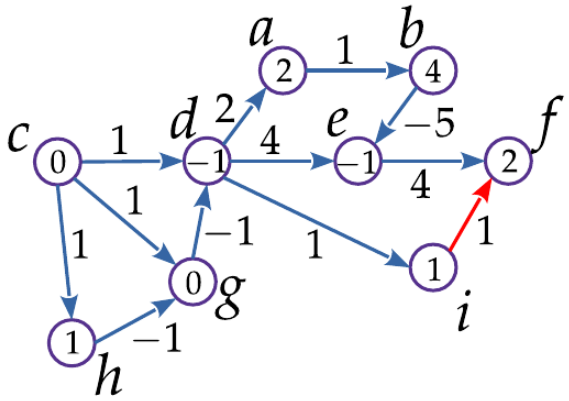
$E(G) = \{ab, be, cd, cg, ch, da, de, di, ef, gd, hg, if\}$

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* 		

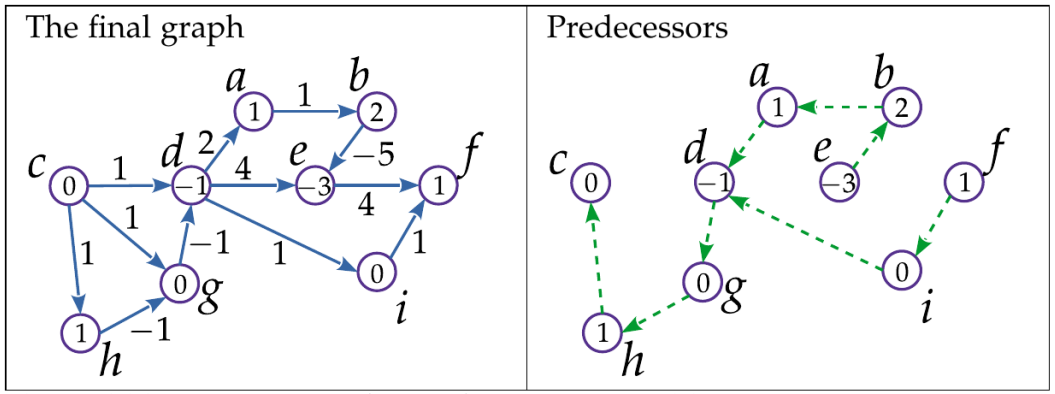
( ) = { , , , , h, , , , , , , h , }

Bellman-Ford algoritam - primjer

<table>
<tr><th colspan="3">Outer loop iteration 2</th></tr>
<tr>
<td>Iteration 1: The edge ab
  </td>
<td>Iteration 2: The edge be
  </td>
<td>We skip a number of the inner edge iterations here.</td>
</tr>
<tr>
<td>Iteration 12: The edge if
  </td>
<td></td>
<td></td>
</tr>
</table>

$E(G) = \{ab, be, cd, cg, ch, da, de, di, ef, gd, hg, if\}$

Bellman-Ford algorithm - example

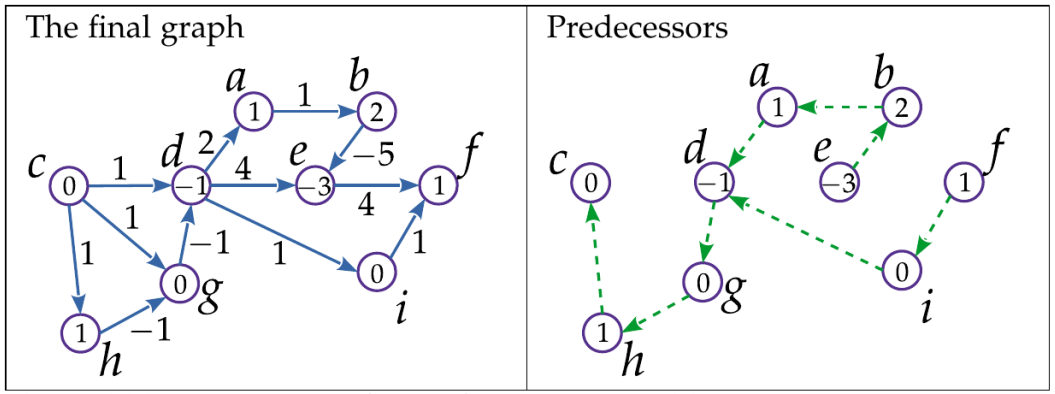


	Iteration					
	0	1		2	3	4
a	∞	3		2	1	
b	∞			4	3	2
c	0					
d	∞	1	0	-1	-1	
e	∞	5		-1	-2	-3
f	∞	9	3	2	1	
g	∞	1	0			
h	∞	1				
i	∞	2		1	0	

- 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 algoritam - primjer



	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	

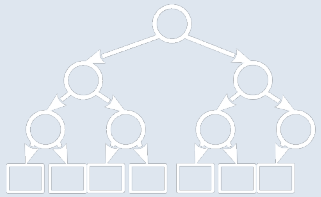
- Nakon $|V| - 1$ iteracija dobivamo konačni rezultat
  - Desni graf predstavlja prethodnike, a time i najkraće putanje od c do svih ostalih vrhova
- Primjer rješavanja kroz tablicu
  - Pratimo $E(G)$ i ažuriramo udaljenosti u koloni
- Kompleksnost Bellman-Ford algoritma je $O(E * V)$. Vanjska petlja iterira kroz vrhove, dok unutarnja iterira kroz bridove

$$E(G) = \{ab, be, cd, cg, ch, da, de, di, ef, gd, hg, if\}$$

Bellman-Ford algorithm – faster version

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

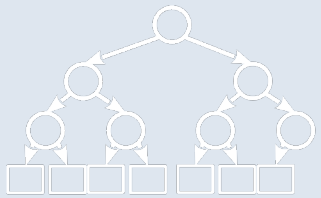
- The main difference is that we do not process all edges
- The initial vertex is placed in the list, and then only vertices (neighbors) whose label has changed
- Which means that only subgraphs are processed where there will potentially be some label change - it can happen if we have circles and cycles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



Bellman-Ford algoritam – brža inačica

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

- Osnovna razlika je u tome što ne obrađujemo sve bridove
- U listu se stavlja početni vrh, a zatim samo vrhovi (susjedi) čija se labela promijenila
- Što znači da se obrađuju samo podgrafovi gdje će potencijalno doći do neke promjene labele – može se desiti ako imamo krugove i cikluse u grafu
- Kompleksnost je i dalje $O(E * V)$

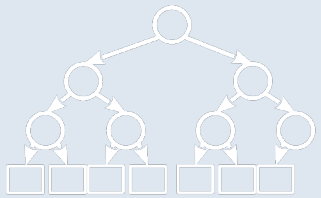


Warshall-Floyd-Ingerman algorithm

- 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

$$\forall \in : "= , 1 \leq \leq | |$$

$$)= , (= , *= , += , ,=$$

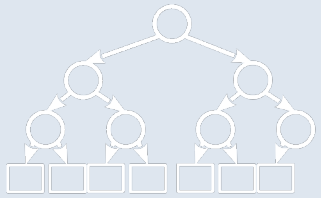


Warshall-Floyd-Ingerman algoritam

- Spada u algoritme koji računaju najkraću udaljenost između svih vrhova grafa (all-to-all)
- Label-correcting algoritam
  - Sve labele se ažuriraju do kraja rada algoritma
  - Može raditi s negativnim težinama u grafu
  - Nije u mogućnosti raditi s negativnim ciklusima
- Radi s matricama udaljenosti (distance matrix)
- Zamislimo neki skup vrhova $V = \{a, b, c, d, e\}$
- Mapiramo vrhove iz V tako da ih označimo rednim brojevima

$$\forall x \in V : v_i = x, 1 \leq i \leq |V|$$

$$v_1 = a, v_2 = b, v_3 = c, v_4 = d, v_5 = e$$

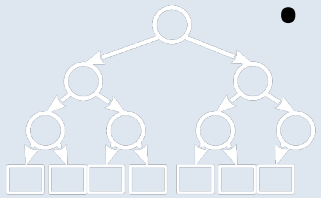


Warshall-Floyd-Ingerman algorithm

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

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

  - 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 ;"&(", #)

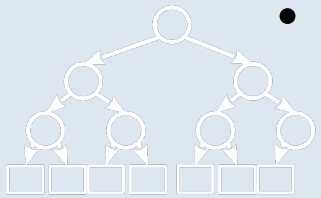


Warshall-Floyd-Ingerman algoritam

- Matrica udaljenosti za prethodni skup vrhova V izgleda kao

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

  - udaljenost između vrhova v_i i v_j označava se kao d_{ij}
  - za očekivati je da će se naći barem jedna putanja kroz graf između vrhova v_i i v_j - ovo nije nužno ako graf nije povezan, ali pretpostavimo da jest
  - treba odabrati onu putanju koja je najkraća ili $d_{min}(v_i, v_j)$



Warshall-Floyd-Ingerman algorithm

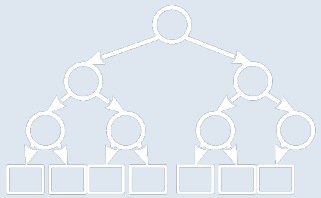
- If we have a new trajectory that passes through an intermediate vertex !

  =)… !… -

  - 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 !

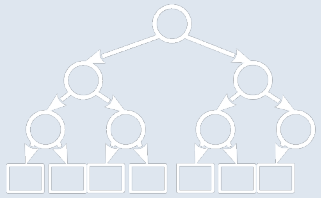


Warshall-Floyd-Ingerman algoritam

- Ako imamo novu putanju koja prolazi međuvrhom v_k

$$p = v_1 \dots v_k \dots v_n$$

  - smatra se da je putanja p kraća ako vrijedi $d(v_1, v_k) + d(v_k, v_n) < d(v_1, v_n)$
- WFI algoritam iterira po vrhovima grafa postavljajući ih kao međuvrh v_k
  - na taj se način testira da li je taj međuvrh v_k na minimalnoj putanji između v_i i v_j
  - s obzirom da WFI algoritam izračunava udaljenosti između svih vrhova, imamo tri petlje s kojima iteriramo po vrhovima grafa, odabirući u svakoj od njih v_i, v_j i v_k



Warshall-Floyd-Ingerman algorithm

- Distances are calculated as

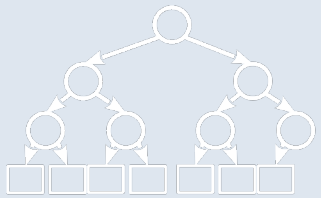
$$201 = \begin{cases} 01 & , = 0 \\ 23- & (\ 01\ ,\ 02^{23-} + 23-21\ )\ , \geq 1 \end{cases}$$

  - when we look at the intermediate peak (, the distance is the minimum between the distances $&calculated for the intermediate peak ('#and the distance of the path passing through the intermediate peak (

- Paths (predecessors) are saved in the path matrix as

$$201 = C_{01}\begin{cases} 23- & , 23-01 \leq 02^{23-} + 23-21 \\ 23-21 & , 01^{23-} > 02^{23-} + 21^{23-} \end{cases}$$

  - which means that if we updated the distance $&then we must also put that the predecessor of the vertex &intermediate peak (



Warshall-Floyd-Ingerman algoritam

- Udaljenosti se računaju kao

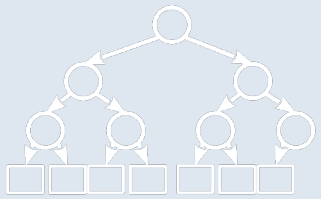
$$d_{ij}^{k} = \begin{cases} w_{ij} & , k = 0 \\ min\left(d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1}\right) & , k \geq 1 \end{cases}$$

  - kada gledamo međuvrh v_k, udaljenost je minimum između udaljenosti d_{ij} izračunate za međuvrh v_{k-1} i udaljenosti putanje koja prolazi kroz međuvrh v_k

- Putanje (prethodnici) spremaju se u matricu putanja kao

$$\pi_{ij}^{k} = \begin{cases} \pi_{ij}^{k-1} & , d_{ij}^{k-1} \leq d_{ik}^{k-1} + d_{kj}^{k-1} \\ \pi_{kj}^{k-1} & , d_{ij}^{k-1} > d_{ik}^{k-1} + d_{kj}^{k-1} \end{cases}$$

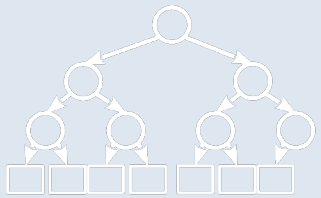
  - što znači da ako smo ažurirali udaljenost d_{ij} tada moramo staviti i da je prethodnik vrha v_j međuvrh v_k



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

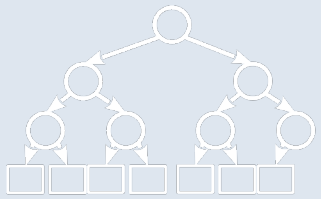
$$.\text{"\#} = \begin{cases} , = = 0_{\#} \\ , \neq \text{"\#} \neq 0 \end{cases}$$



Warshall-Floyd-Ingerman algoritam

- Na početku imamo inicijalnu matricu udaljenosti, koja je $D^0 = W$, te sadrži udaljenosti samo direktno povezanih vrhova, izračun ostalih udaljenosti je stvar WFI algoritma
- Inicijalna matrica putanja definira se iz težinske matrice susjedstva W na način

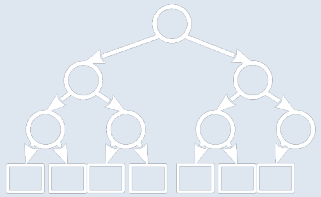
$$\pi_{ij}^0 = \begin{cases} null & , i = j \; or \; w_{ij} = 0 \\ i & , i \neq j \; and \; w_{ij} \neq 0 \end{cases}$$



Warshall-Floyd-Ingerman algorithm

```
procedure WFI(W)
    Create initial distance matrix D = D^0 from W
    Create initial path matrix Π = Π^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]
                    Π[i,j] = Π[k,j]
```

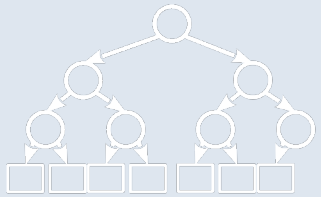
- The algorithm is simple, it has three loops that we iterate through peaks ", #and !
- Thus, the complexity of the algorithm is somewhat high (*)



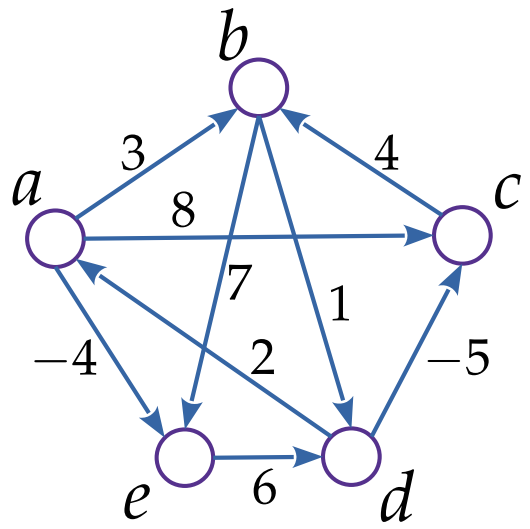
Warshall-Floyd-Ingerman algoritam

```
procedure WFI(W)
    Create initial distance matrix D = D^0 from W
    Create initial path matrix Π = Π^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]
                    Π[i,j] = Π[k,j]
```

- Algoritam je jednostavan, ima tri petlje kojima iteriramo po vrhovima v_i, v_j i v_k
- Time je i kompleksnost algoritma nešto visoka $O(V^3)$



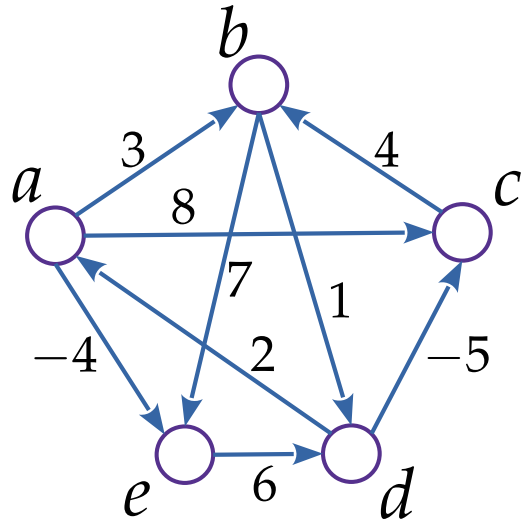
WFI algorithm - example



$$\mathbf{D^0} = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \end{array} \begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{bmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix} \end{array}, \mathbf{\Pi^0} = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \end{array} \begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{bmatrix} null & 1 & 1 & null & 1 \\ null & null & null & 2 & 2 \\ null & 3 & null & null & null \\ 4 & null & 4 & null & null \\ null & null & null & 5 & null \end{bmatrix} \end{array}$$

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

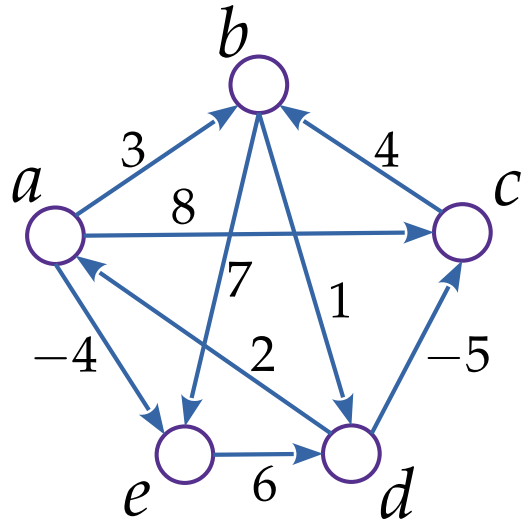
WFI algoritam - primjer



$$\mathbf{D^0} = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \end{array} \begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{bmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix} \end{array}, \mathbf{\Pi^0} = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \end{array} \begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{bmatrix} null & 1 & 1 & null & 1 \\ null & null & null & 2 & 2 \\ null & 3 & null & null & null \\ 4 & null & 4 & null & null \\ null & null & null & 5 & null \end{bmatrix} \end{array}$$

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

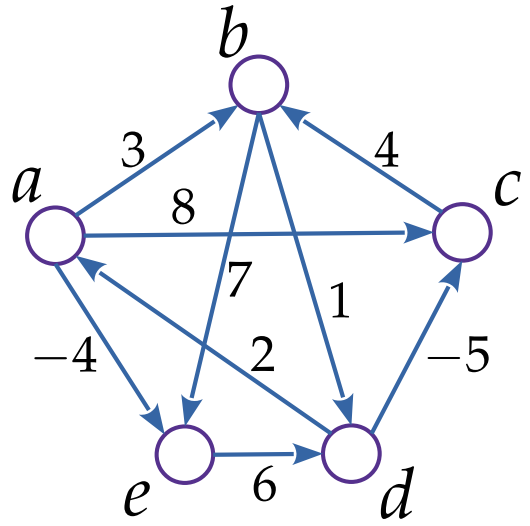
WFI algorithm - example



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

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

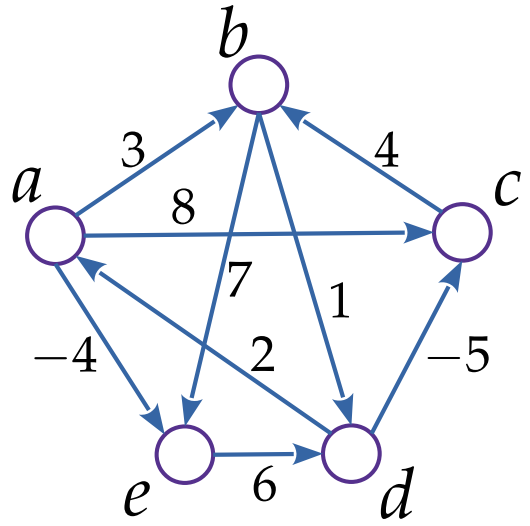
WFI algoritam - primjer



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

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

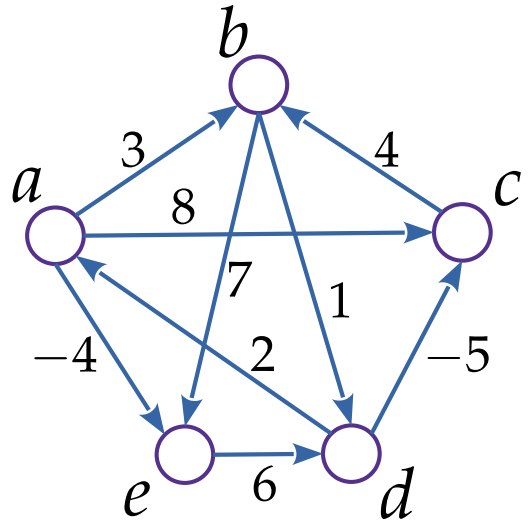
WFI algorithm - example



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

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

WFI algoritam - primjer



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

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

WFI algorithm - example

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

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

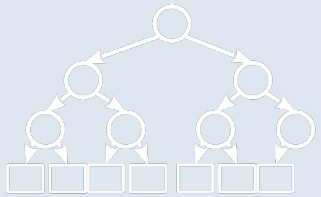
- We look at the distance and the shortest path betweeni = 3 = andj = 5 =
- We read the distance directly from $, that is $ %$= 3
- Reading the path matrix is interpreted as
  - If it is !initial peak, a "final, then we have a trajectory ! #$%... #$& # "
  - In the matrix Π'we have ' !"= ,and then '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

WFI algoritam - primjer

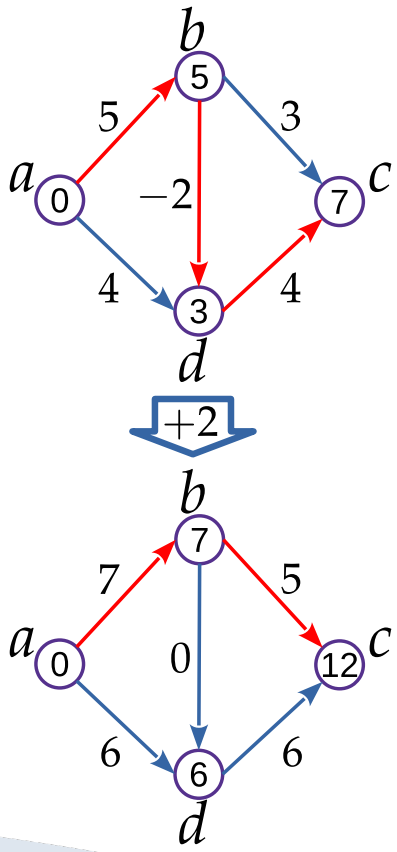
k	vertex
$\pi^5_{35} = 1$	$v_5 = e$
$\pi^5_{31} = 4$	$v_1 = a$
$\pi^5_{34} = 2$	$v_4 = d$
$\pi^5_{32} = 3$	$v_2 = b$
$\pi^5_{33} = null$	$v_3 = c$

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

- Gledamo udaljenost i najkraću putanju između i $= 3 = c$ i j $= 5 = e$
- Udaljenost očitamo direktno iz D^5, to jest $d^5_{35} = 3$
- Čitanje matrice putanja se interpretira kao
  - Ako je v_i početni vrh, a v_j završni, tada imamo putanju $v_i v_{k-n} \dots v_{k-1} v_k v_j$
  - U matrici Π^5 imamo $\pi^5_{ij} = k$, pa zatim $\pi^5_{ik} = k - 1$, pa $\pi^5_{i(k-1)} = k - 1$, sve do $\pi^5_{i(k-n)} = i$
  - π^5_{ii} će po definiciji biti $null$ i tu završavamo
- Za putanju idemo unatrag kroz matricu od π^5_{35}, gdje je $i = 3$, a $j = 5$, pa je tako
  - $j = 5$ i to je vrh e
  - $k = \pi^5_{ij} = \pi^5_{35} = 1$ i to je vrh a
  - $k - 1 = \pi^5_{ik} = \pi^5_{31} = 4$ i to je vrh d
  - $k - 2 = \pi^5_{i(k-1)} = \pi^5_{34} = 2$ i to je vrh b
  - $k - 3 = \pi^5_{i(k-2)} = \pi^5_{32} = 3 = i$ i to je vrh c
  - $\pi^5_{ii} = \pi^5_{33} = null$ i tu završavamo.

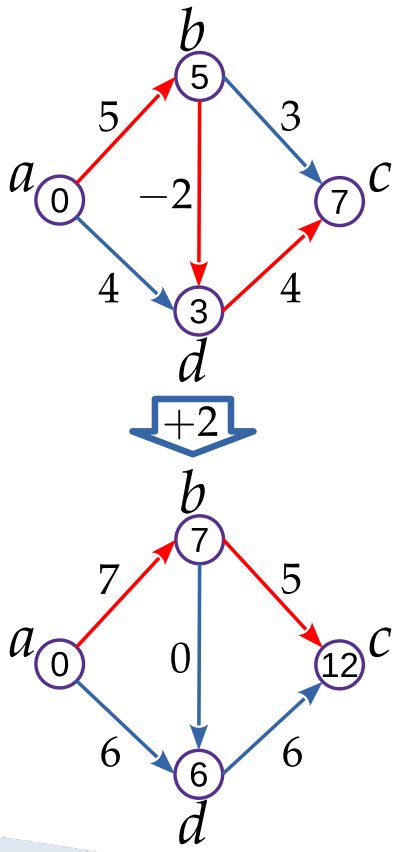


Weight transformation



- 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 to 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

Transformacija težina



- Žarko želimo koristiti Dijkstrin algoritam, ali nam smetaju negativne težine na bridovima
- Da li je moguće nekako transformirati graf da transformacijom uklonimo negativne težine?
- Naivni pokušaj bio bi identičnom translacijom prema najnegativnijoj težini brida
- Najkraći put od a do c u originalnom grafu je $abdc$ s udaljenošću 7
- Dodavanjem težine 2 na sve bridove to se remeti i sada je najkraća putanja abc s udaljenošću 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

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

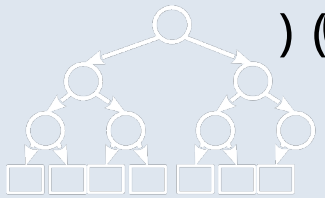
- If we use the previous expression on the path =) (... !we get the distance

$$!=)$$

$$),(!);= T ′(" ">)) =$$

$$"<)$$

$$) ((+)−) (+ ·(·+) !=) !+(!=)− (!) ( ) ( )$$



Transformacija težine

- Ono što znamo sa prethodnih prikaznica je

$$d(v) \leq d(u) + w(uv)$$

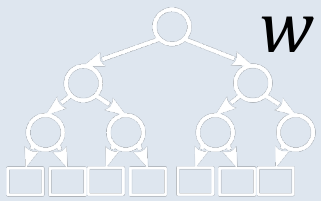
  - znači udaljenost vrha v nije veća od udaljenosti vrha u uvećana za težinu brida $w(uv)$
  - zapišemo li to drukčije dobivamo da je

$$0 \leq d(u) + w(uv) - d(v) = w'(uv)$$

- Ako prethodni izraz upotrijebimo na putanji $p = v_1 v_2 \ldots v_k$ dobivamo udaljenost

$$L(v_1, v_k)' = \sum_{i=1}^{k-1} w'(v_i v_{i+1}) =$$

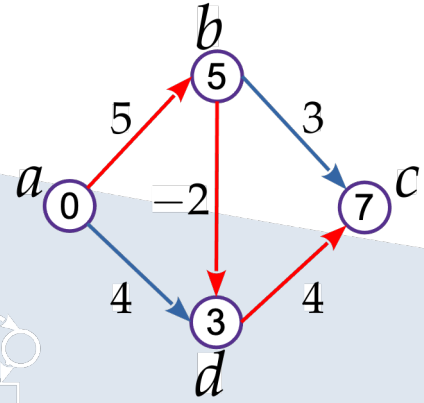
$$w(v_1 v_2) + d(v_1) - d(v_2) + \cdots + w(v_{k-1} v_k) + d(v_{k-1}) - d(v_k)$$



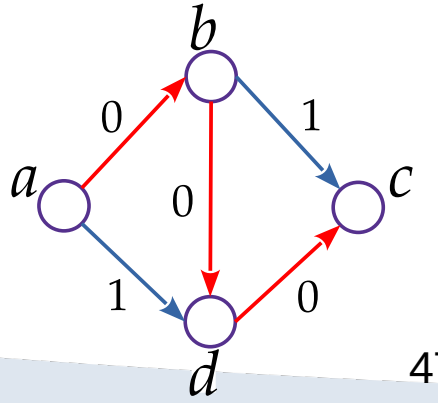
Weight transformation

- Also valid

- The transformation is bijective, so i is valid



v_i	v_j	$w(v_iv_j)$	$d(v_i)$	$d(v_j)$	$w'(v_iv_j)$
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



Transformacija težine

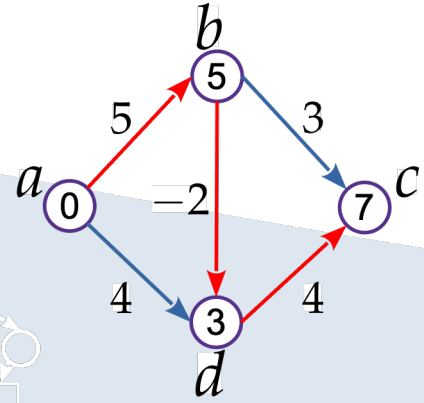
- Također vrijedi

$$L'(v_1, v_k) = \left(\sum_{i=1}^{k-1} w(v_i v_{i+1}) \right) + d(v_1) - d(v_k) =$$

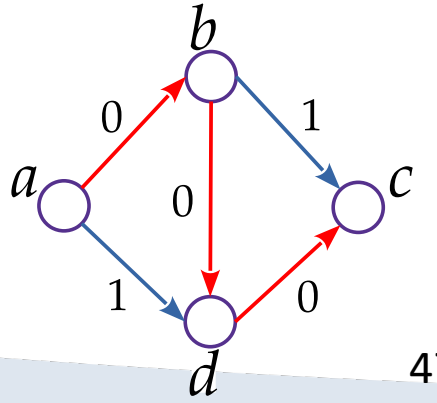
$$L(v_1 v_k) + d(v_1) - d(v_k)$$

- Transformacija je bijektivna pa vrijedi i

$$L(v_1 v_k) = L'(v_1 v_k) + d(v_k) - d(v_1)$$



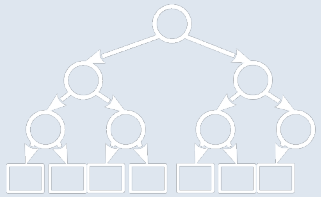
v_i	v_j	$w(v_i v_j)$	$d(v_i)$	$d(v_j)$	$w'(v_i v_j)$
a	b	5	0	5	5+0-5=0
a	d	4	0	3	4+0-3=1
b	d	-2	5	3	-2+5-3=0
b	c	3	5	7	3+5-7=1
d	c	4	3	7	4+3-7=0



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 $L(v_i v_k) = L'(v_i v_k) + d(v_k) - d(v_i)$.

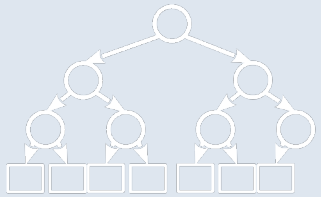
- In order to perform the weight transformation, we first need to determine the Bellman-Ford algorithm the distance of all vertices from one particular vertex
- Then we do the transformation
- After that, using Dijkstra's algorithm, we determine the distance between all pairs of vertices in the graph,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



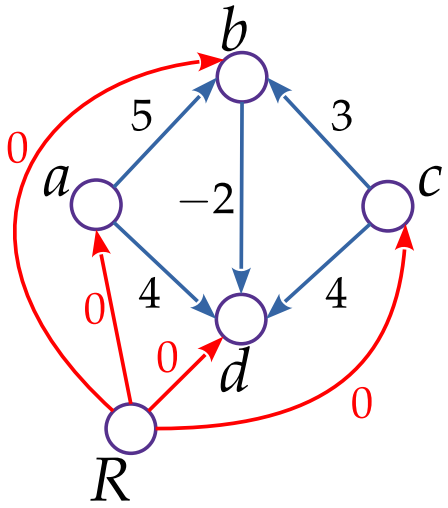
Transformacija težine

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 $L(v_i v_k) = L'(v_i v_k) + d(v_k) - d(v_i)$.

- Da bismo odradili transformaciju težine, prvo trebamo Bellman-Ford algoritmom odrediti udaljenost svih vrhova od jednog određenog vrha
- Zatim odradimo transformaciju
- Nakon toga korištenjem Dijkstrinog algoritma odredimo udaljenost između svih parova vrhova u grafu, čime dobivamo *all-to-all* udaljenosti



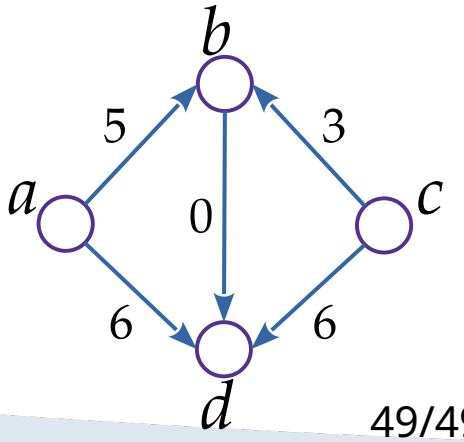
Weight transformation



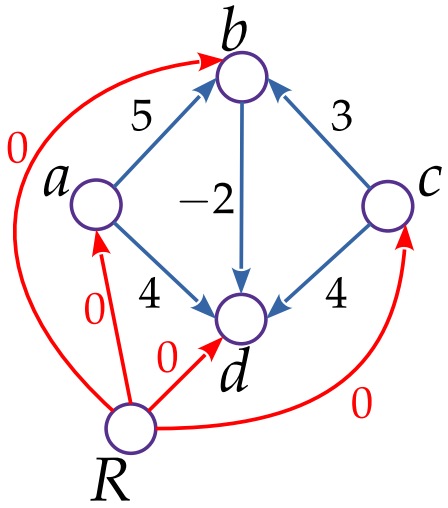
- If we have a disconnected graph, then it is impossible to calculate the distance of all vertices from one reference vertex
- Then we add an artificial vertex (let's say R) which we connect with all other vertices of the graph by edges
  - The weights of these artificially added edges are 0
- Now we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

v	$d(v)$
a	0
b	0
c	0
d	-2

v_i	v_j	$w(v_iv_j)$	$d(v_i)$	$d(v_j)$	$w'(v_iv_j)$
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



Transformacija težine



- Ukoliko imamo nepovezani graf, tada je nemoguće izračunati udaljenost svih vrhova od jednog referentnog vrha
- Tada dodajemo umjetni vrh (recimo R) koji bridovima povezujemo sa svim ostalim vrhovima grafa
  - Težine tih umjetno dodanih bridova su 0
- Sad možemo odrediti udaljenost svih vrhova od umjetnog vrha R i na temelju toga napraviti transformaciju

v	$d(v)$
a	0
b	0
c	0
d	-2

v_i	v_j	$w(v_iv_j)$	$d(v_i)$	$d(v_j)$	$w'(v_iv_j)$
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