Odabrani algoritmi nad grafovima

(*Graph Algorithms*)

AlgoritmiNadGrafovima

Praćenje unatrag (*Backtracking*)

- strategija po kojoj se slijed koraka tijekom rješavanja problema pamti pa se nakon neuspješnog pokušaja možemo vratiti na polazno stanje i nastaviti samo s još neisprobanim mogućnostima
- za probleme čije rješavanje iz koraka u korak nudi više izbora za sljedeći korak

Tipičan primjer:
problem kraljica (ponoviti ASP!)

- *backtracking* se ostvaruje `for` petljom, tj. povratkom u nju nakon nuspješnog pokušaja

```
Queen (row)
for every position col in the same row;
   if col is not attacked by already placed queens
   {   place the queen in position col;
       if row < 8
              Queen(row+1);
       else
              return success;
       remove the queen from position col;  }
return failure;
```

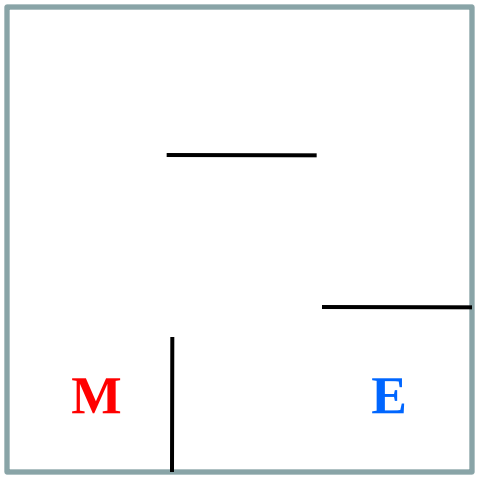
Glavna odlika i prednost *backtrackinga* je izbjegavanje moguće velikog broja (neisprobanih) mogućnosti ako se ustanovi da one sigurno ne vode ka rješenju.

- kraljice: ako se kraljica uopće ne može smjestiti u npr. peti redak (svugdje ju napadaju već postavljene), redci 6, 7 i 8 se ni ne ispituju, nego se traži novo mjesto (stupac) za kraljicu u četvrtom retku

Backtracking će nas sigurno dovesti do rješenja ako ono postoji, ali pomoći u izbjegavanju jalovih pokušaja može samo kada se konačni neuspjeh može predvidjeti prije iscrpljivanja svih mogućnosti koje pruža posljednja donesena odluka (detekcija pogreške!).

- kraljice: nakon neuspješnog pokušaja smještanja 5. kraljice, niti ne pokušavamo smjestiti neku sljedeću jer znamo da sigurno nećemo ostvariti konačni cilj; tražimo novo mjesto za 4. kraljicu

U npr. problemu izlaska iz labirinta, *backtracking* ne smanjuje broj mogućnosti koje moramo isprobati.



1	1	1	1	1	1	1
1	0	0	0	0	0	1
1	0	0	1	0	0	1
1	0	0	0	0	0	1
1	0	0	0	0	1	1
1	**M**	1	0	0	**E**	1
1	1	1	1	1	1	1

P = prolaz (staza)
Pmax = 3

Z = zid
Zmax = Pmax – 1 = 2

D = dim = Pmax + Zmax = 5

```
exitMaze (cell)                    //rekurzivno - tail rekurzija
if cell = exitCell
   return success;
//provjera: up, down, left, right dozvoljena polja?
if exitMaze(up) == success or exitMaze(down) == success
    or exitMaze(left) == success or exitMaze(right) == success
        return success;
return failure;
```

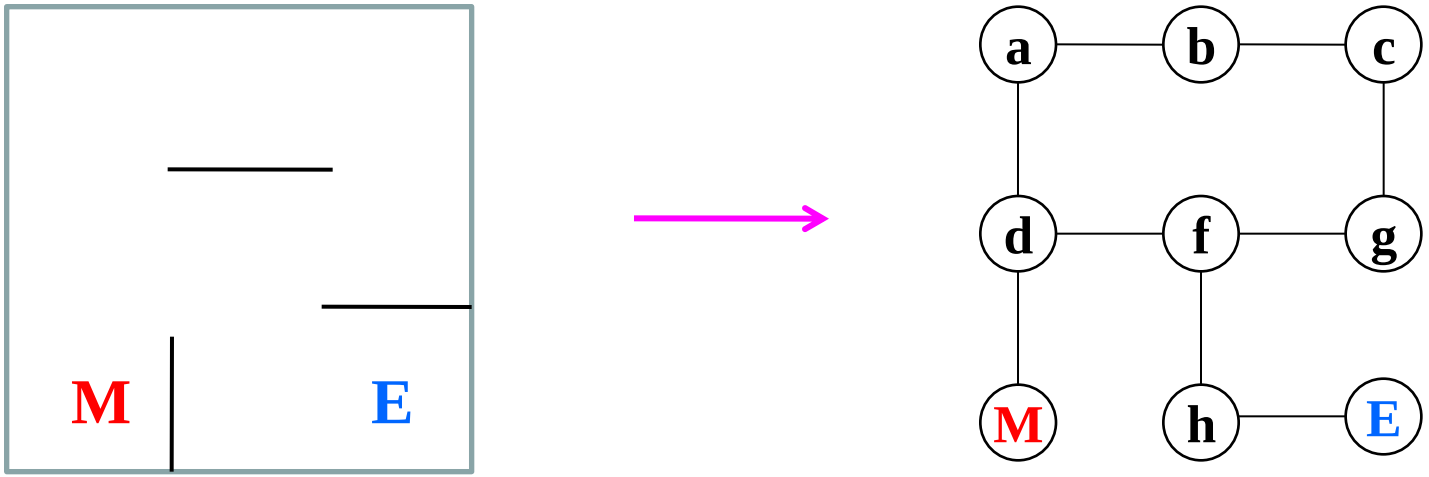
```
exitMaze ()                        //nerekurzivno
initialize stack, entryCell, exitCell, currCell = entryCell
while currCell is not exitCell
{  mark currCell cell as visited;
   push onto the stack the unvisited neighbours of currCell;
   if stack is empty
        return no way out;
   else
        currCell = a cell from the stack;    }
return success;
```

Maze

Koji je najkraći put iz labirinta?

Backtracking ne daje odgovor ...

Problem se može modelirati pretvorbom labirinta u graf.



Sada je pitanje koji je najkraći put od vrha M do E. Odgovor daje teorija grafova, odnosno algoritmi nad grafovima.

Osnovne definicije

Graf (*graph*) je uređeni par nepraznog konačnog skupa vrhova V (*vertex*) i skupa (moguće praznog) bridova E (*edge*); G=(V,E).

- graf se može shvatiti kao stablo (šuma; *forest*) u kojem nema hijerarhije među čvorovima

Brid (*edge*) je svaki dvočlani podskup skupa V.

- poveznica dva vrha

Red (*order*) **grafa** je broj vrhova u njemu.

- označava se s $|V|$; često pojednostavnjeno samo V

Veličina (*size*) **grafa** je broj bridova u njemu.

- označava se s |E|; pojednostavnjeno samo E
- brid između vrhova v_i i v_j označavamo s edge(v_i, v_j) ili kraće samo v_i v_j ili jednostavno samo e_i, pri čemu je $e_i = v_{i-1}v_i$

Petlja (loop) je brid koji počinje i završava u istom vrhu.

Supanj (*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

Razlikujemo:

1. **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 brid $v_i v_j$ u E
   - takav je brid prileže ći (*incident*) vrhovima v_i i v_j

2. **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 ako u E postoji brid $v_i v_j$ (ako se iz v_i može izravno u v_j)
   - vrh v_i se smatra susjedom vrhu v_j ako u E postoji brid $v_j v_i$
   - brid $v_i v_j$ se naziva izlaznim bridom (*outedge*) vrha v_i i ulaznim bridom (*inedge*) vrha v_j

Daljnje definicije i podjele:

Jednostavni graf (*simple graph*) je graf koji između svaka dva vrha ima najviše jedan brid i u kojem nema petlji (slike. a...d).

- taj se pojam primarno odnosi na neusmjerene grafove

Multigraf (*multigraph*) je graf koji može imati više od jednog brida između dva vrha (slika e).

Pseudograf (*pseudograph*) je multigraf u kojem mogu postojati povratne petlje (slika f).

Obilazak ili šetnja (*walk*) od vrha v_1 do vrha v_n je naizmjenični niz vrhova i bridova $v_1, e_2, v_2, e_3, \ldots, e_n, v_n$.

- kraće se označava $v_1,v_2,\ldots,v_{n-1}$ ili samo $v_1v_2\ldots v_n$

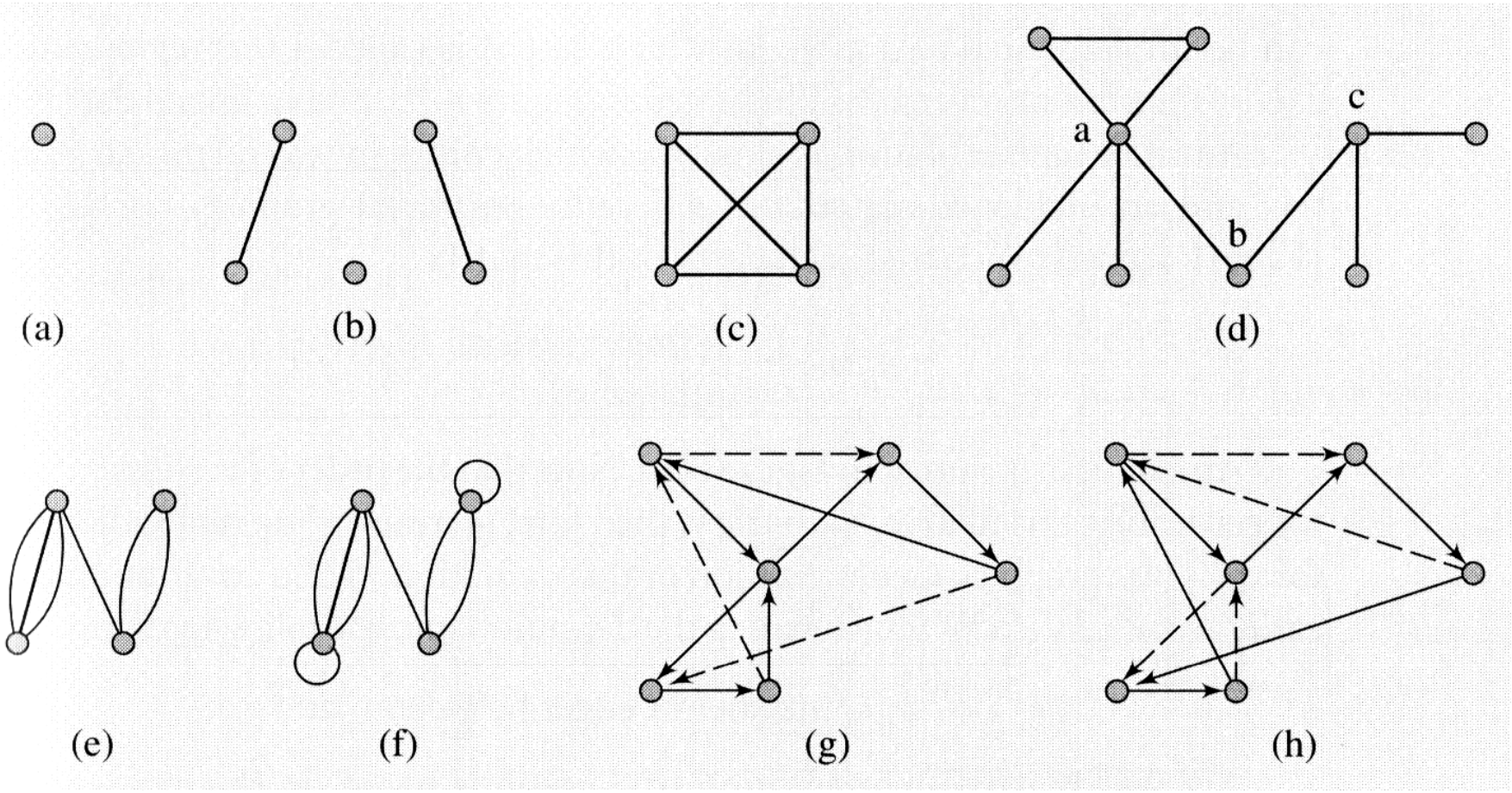
Putanja (*trail*) je obilazak u kojem su svi bridovi različiti (znači svakim bridom se prolazi samo jednom, ali vrhovi se mogu ponavljati).

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 $v_1 = v_n$ (slika g).

Ciklus (*cycle*) je staza (*path*) na kojoj je $v_1 = v_n$ (slika h).

Primjeri uz definicije pojmova iz teorije grafova:
slike a...d - jednostavni grafovi, e - multigraf,
f - pseudograf, g - krug, h - ciklus



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 Kn
- broj bridova u potpunom grafu = broj dvočlanih podskupova skupa vrhova V

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

Podgraf (*subgraph*) je skup G' = (V',E') koji je podskup grafa G = (V,E) takav da svaki brid iz E' pripada i E.

Prikaz (pohrana) grafa u računalu

1. Lista susjedstva (*adjacency list*)
   - svi vrhovi susjedni nekom vrhu
   a) u obliku tablice (*star representation*)
      - slika b
   b) kao dvodimenzionalno povezana lista
      - slika c
2. Matrični zapis
   a) Matrica susjedstva (*adjacency matrix*)
      - [vrhovi × vrhovi]: slika d
   b) Matrica povezanosti (*incidence matrix*)
      - [vrhovi × bridovi]: slika e

Odabir ovisi o situaciji:

a) pristup svim susjedima nekog vrha
   - bolje liste jer je potrebno deg(v) koraka, naspram |V| za matrice

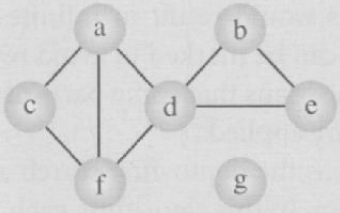
b) pojedinačne intervencije (dodati ili ukloniti vrh)
   - bolje matrice; brži pristup i O(1) održavanje

Prikazi grafa:

b,c - lista susjedstva

d - matrica susjedstva

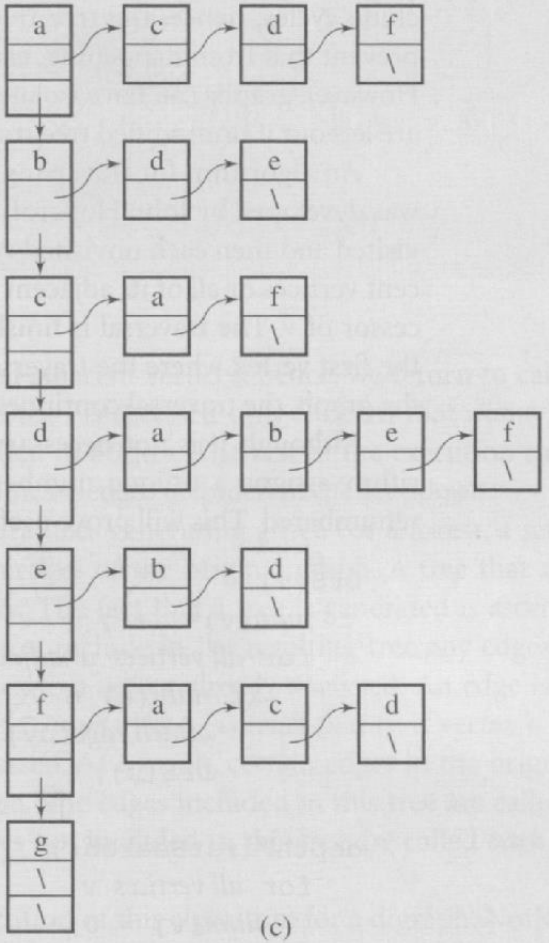
e - matrica povezanosti



(a)

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

(b)



(c)

	a	**b**	**c**	**d**	**e**	**f**	**g**
a	0	0	1	1	0	1	0
b	0	0	0	1	1	0	0
c	1	0	0	0	0	1	0
d	1	1	0	0	1	1	0
e	0	1	0	1	0	0	0
f	1	0	1	1	0	0	0
g	0	0	0	0	0	0	0

(d)

	ac	**ad**	**af**	**bd**	**be**	**cf**	**de**	**df**
a	1	1	1	0	0	0	0	0
b	0	0	0	1	1	0	0	0
c	1	0	0	0	0	1	0	0
d	0	1	0	1	0	0	1	1
e	0	0	0	0	1	0	1	0
f	0	0	1	0	0	1	0	1
g	0	0	0	0	0	0	0	0

(e)

Obilazak grafa (*Graph Traversal*)

Algoritmi za obilazak stabla ne zadovoljavaju jer:

1) graf može imati cikluse pa bi se algoritam za stablo mogao naći u beskonačnoj petlji
2) graf može imati odvojene i nepovezane vrhove.

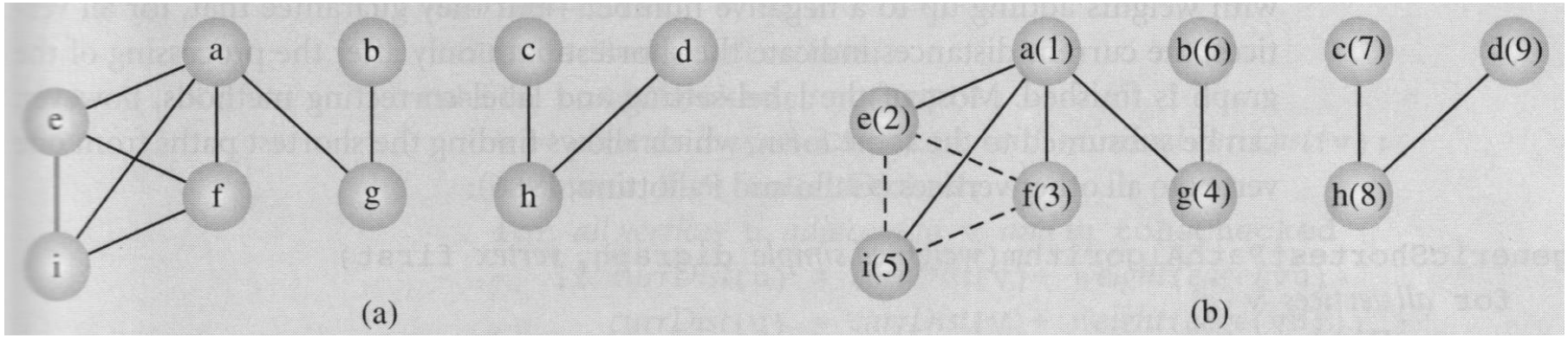
Dva najpoznatija algoritma za obilazak grafa su:

1) Obilazak (prvo) u širinu (*Breadth First Search;* BFS)
2) Obilazak (prvo) u dubinu (*Depth First Search*; DFS)

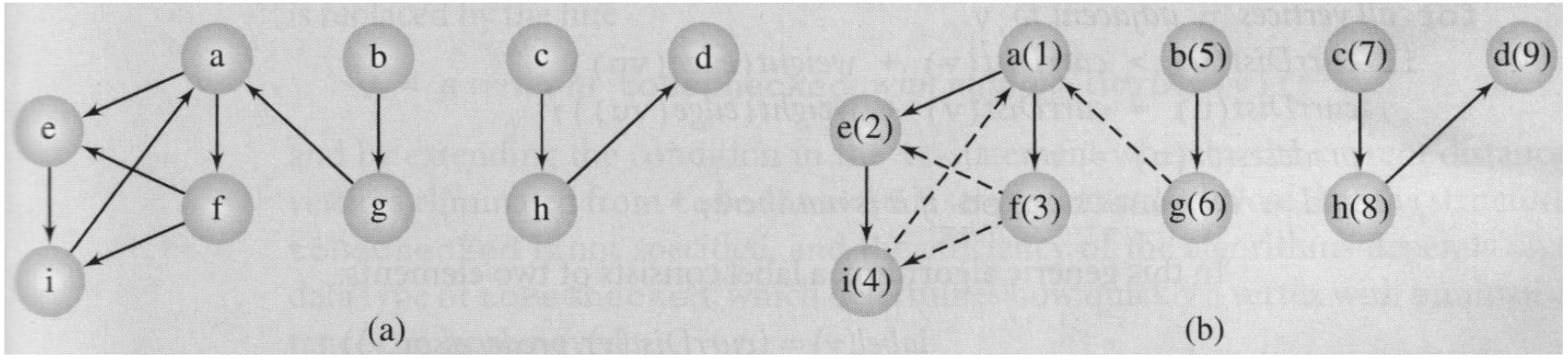
Osim u jednostavnim primjenama (npr. obilazak grafa, detekcija ciklusa, provjera povezanosti pojedinih vrhova), DFS i BFS nisu međusobno zamjenjivi zbog bitne logičke različitosti. Oba su temelj za brojne složenije algoritme i teorijski su podjednako brzi, ali u stvarnosti je DFS nešto sporiji jer je rekurzivan.

Obilazak (prvo) u širinu (*Breadth First Search*; BFS)

Obilazak neusmjerenog grafa:



Obilazak usmjerenog grafa:



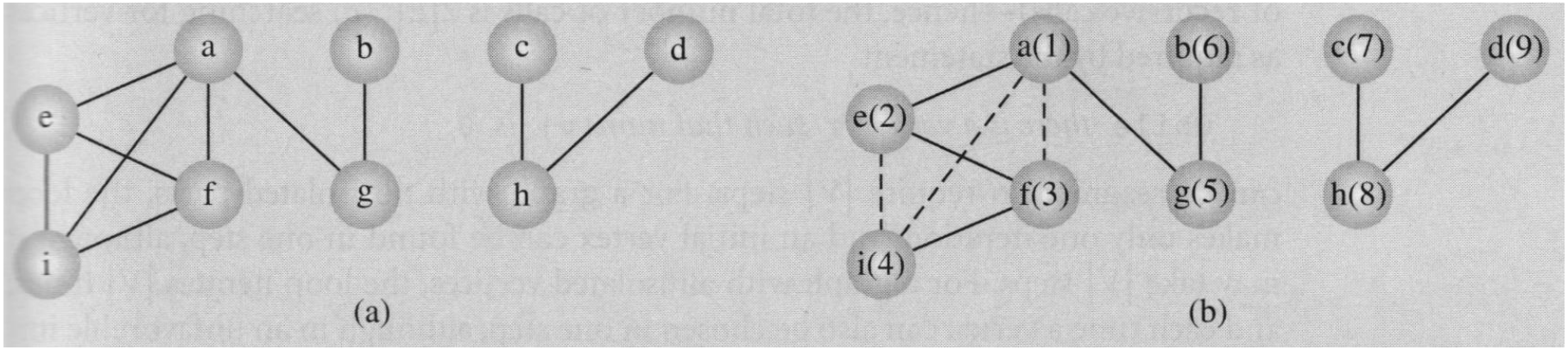
Obilazak (prvo) u širinu (*Breadth First Search* - BFS):

- nerekurzivan; "prirodna" pomoćna struktura mu je red (lista)
- složenost O(V+E)

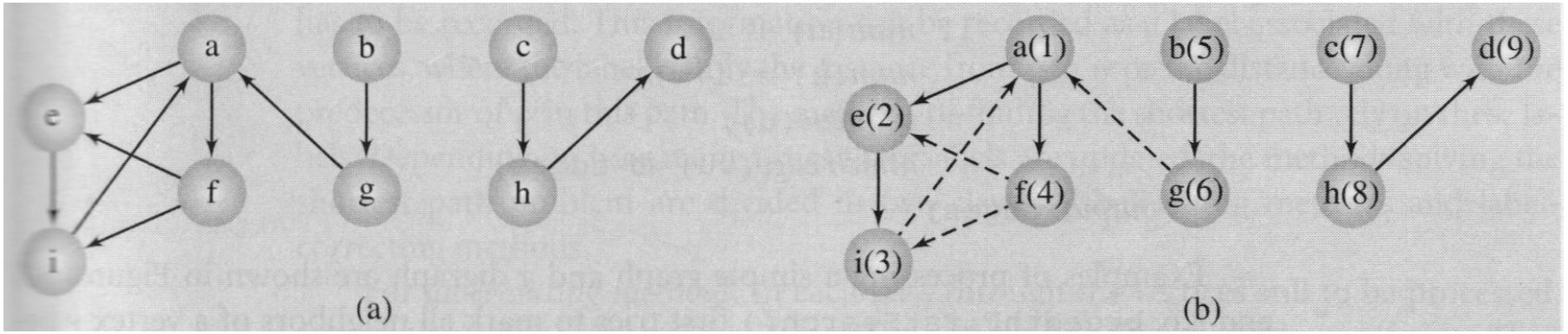
```
BreadthFirstSearch (G)
inicijalizacija: svim vrhovima num(v)=0
korak = 0; ustrojiti pomoćni red spremnik;
while u grafu G još ima neobiđenih vrhova v //num(v)==0
      num(v) = ++korak;
      dodati v u red spremnik (na kraj);
      while još ima vrhova u redu spremnik //nije prazan
            prvi = prvi u redu;
            for svi neobiđeni susjedi u vrha prvi
                  num(u) = ++korak;
                  u dodati u red spremnik;
                  zabilježiti brid (prvi,u); //ako treba
                                             //pamtiti put
```

Obilazak (prvo) u dubinu (*Depth First Search* - DFS)

Obilazak neusmjerenog grafa:



Obilazak usmjerenog grafa:



Obilazak (prvo) u dubinu (Depth *First Search* - DFS):

- rekurzivan; "prirodna" pomoćna struktura mu je stog
- složenost O(V+E)

```
DepththFirstSearch (G)
inicijalizacija: svim vrhovima num(v)=0
korak = 0;                    //je li potrebno ovisi o DFS(v)
while u grafu G još ima neobiđenih vrhova v  //num(v)==0
     DFS(v);
return edges;

DFS (v)
num(v) = ++korak;    //korak:statička,argument,globalna
for svi susjedi u vrha v    //Uvjeti u 'for' i 'if' zapravo
     djeluju zajedno i izdvajaju samo neobiđene susjede
     if num(u)==0
          prethodnik(u) = v;        //ako je potrebno
          zabilježiti brid (v,u) u edges;
          DFS(u);
```

Rezultantna putanja oba algoritma, DFS i BFS, čini stablo u kojem su svi vrhovi grafa. Takvo se stablo naziva **razapinjajuće stablo** (*Spanning Tree*).

Da je retultat stvarno stablo proizlazi iz bilježenja samo onih bridova koji završavaju u još neobiđenim vrhovima. Zbog toga razapinjajuće stablo ne mora sadržavati sve bridove grafa → dva važna pojma:

Poniručí (unaprijedni) bridovi (*forward edges*) – bridovi u razapinjajućem stablu (dakle, oni koji poniru, spuštaju se prema dubljim čvorovima)

Povratni bridovi (*back edges*) – bridovi koji završavaju u već obiđenom vrhu (nisu u razapinjajućem stablu; vraćaju se unazad).

- Na slikama su poniručí bridovi označeni punim, a povratni isprekidanim crtama. Povratni bridovi mogu poslužiti za detekciju ciklusa.

Najkraći putevi u grafu (*Shortest Paths*).

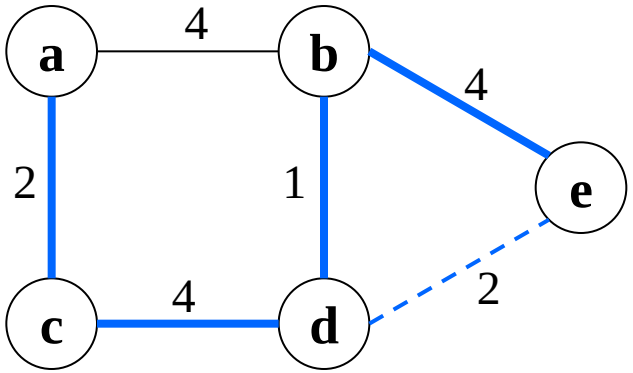
- temeljni algoritmi za brojne primjene (npr. transport, komunikacije, distribucijske energetske mreže, projektiranje integriranih elektroničkih sklopova, ...)
- bridovi, kao apstrakcija (model) poveznica između dvaju čimbenika nekog sustava, dobivaju "težine"; oznaka w(u,v)
- prilikom traženja najkraćeg puta između vrhova `u` i `v` potrebne su nam informacije o udaljenostima "međuvrhova" `k` (međuvrh = vrh na putu između neka dva vrha u grafu, dakle ne ni polazni ni završni)
  - ta se informacija može pohraniti u svakom vrhu kao *labela* (*label*)
- odabir algoritma za traženje najkraćeg puta ovisi o težinama bridova u grafu; dvije su osnovne skupine tih algoritama:

label-setting metoda: jednom upisana labela više se ne mijenja
- za grafove s težinama bridova ≥ 0; Dijkstra

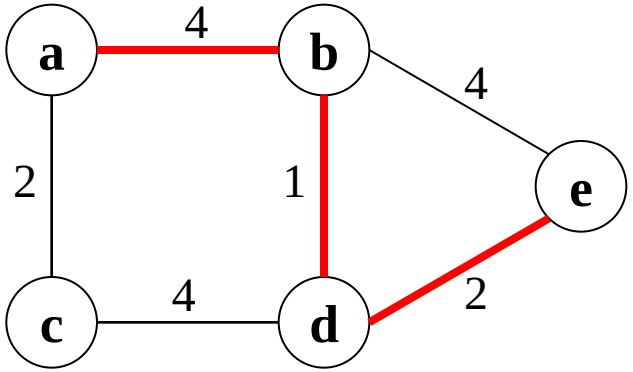
label-correcting metoda: sve labele mogu se mijenjati sve do završetka cijelog postupka
- za grafove s bilo kakvim težinama bridova; Bellman-Ford

Ilustracija problema:

Nije rješenje u svakom koraku prijeći u najbliži vrh!



a→e: Krenemo li iz **a**, najbliži je **c**.
Iz **c** možemo samo u **d** i potom do cilja **e**. Put je a→c→d→e, ukupne duljine 2+4+2=8.
Provođenjem načela "u najbliži" sve do kraja, put bi bio a→c→d →b→e, ukupne duljine 2+4+1+4=11.



a→e:
Najkraće: a→b→d→e, ukupno 7.

Ideja: promatrajmo najkraći put unatrag!

Dijkstrin algoritam

E. W. Dijkstra: *A note on two problems in connexion with graphs*,
Numerische Mathematik, 1 (1959), S. 269–271

Pretpostavimo da već znamo najkraći put od v_0 do v_n i označimo sve vrhove na tom putu s v_i, i=0,1,2,...,n. Također, označimo udaljenost pojedinog vrha v_i od polaznog vrha s $d(v_0,v_i)$. Tada je najkraći put (najmanja udaljenost) od v_0 do v_n sigurno jednak

$$d_{min}(v_0,v_n) = d_{min}(v_0,v_{n-1}) + d_{min}(v_{n-1},v_n) \ .$$

Općenito, za svaki vrh v_i na najkraćem putu vrijedi

$$d_{min}(v_0,v_n) = d_{min}(v_0,v_i) + d_{min}(v_i,v_n) \ .$$ (vidi Cormen...)

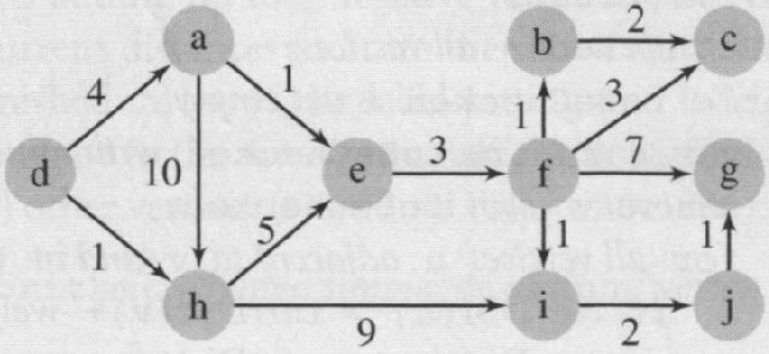
Dakle, gledajući unatrag, ako svaki vrh na najkraćem putu "zna" (barem) svojeg neposrednog prethodnika, može se rekonstruirati cijeli put do polaznog vrha.

– računati $d(v_0,v_j) = d(v_0,v_{j-1}) + d(v_{j-1},v_j)$, smanjujući j do j=n do j=1

Dijkstrin algoritam:

1. Svim vrhovima, osim polaznom, pridjelimo udaljenost ∞ i označimo ih kao neobrađene.
2. Promatramo sve neobrađene susjede v_i vrha u kojem se trenutačno nalazimo i izračunavamo im udaljenosti od polaznog vrha kada do njih dolazimo iz onog u kojem jesmo. Označimo tu udaljenost s $d^*(v_0,v_i)$. Ako je $d^*(v_0,v_i)$ manja od udaljenosti koja je do tada bila upisana u labelu vrha v_i, našli smo kraći put do vrha v_i i upisujemo $d^*(v_0,v_i)$ u labelu, a trenutačni vrh postaje prethodnik vrha v_i. Ako $d^*(v_0,v_i)$ nije kraća, samo nastavljamo s obilaskom susjeda.
3. Nakon osvježavanja labela neobrađenih vrhova (susjeda), za sljedeći vrh odabiremo onaj među neobrađenima (svima, ne samo susjedima prethodnog) koji je najbliži polaznom vrhu (najmanja udaljenost upisana u labelu) i nastavljamo ponavljajući postupak od točke 2 sve dok kao vrh najbliži polaznom ne bude izabran upravo odredišni vrh (cilj).

Primjer: detaljno praćenje rada Dijkstrinog algoritma



(a)

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

(b)

Dijkstrin algoritam: pseudokod

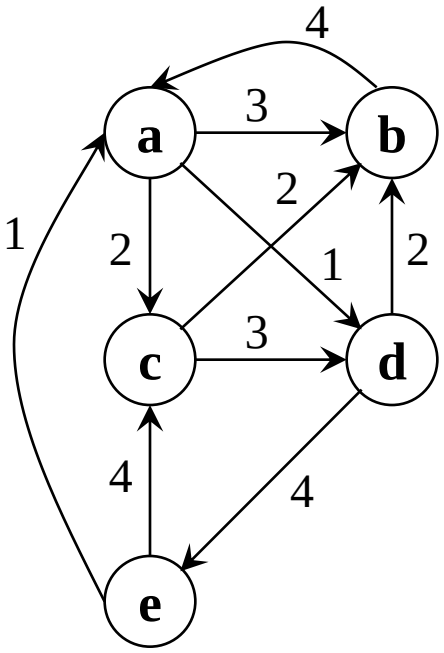
```
Dijkstra (source, dest)
initialization: for all vertices d(v)= inf   // d(v) = d(source,v)
d(source) = 0;
ToBeCh = all vertices;      //ToBeCh - popis svih neobrađenih
while there are vertices in ToBeCh      //... is not empty
      v = a vertex in ToBeCh with the least d(v);
      if v == dest    //Bez ovog uvjeta naći će najmanje
            return ... ;    //udaljenosti do svih.
      remove v from ToBeCh;
      for all vertices u adjacent to v and in ToBeCh
            dnew(u)=d(v)+edge(v,u); //moguća nova d(u)
            if dnew(u) < d(u)
                  d(u) = dnew(u);
                  predecessor(u) = v;      //prethodnik se
                  //najlakše čuva kao članska varijabla vrha
```

Složenost Dijkstrinog algoritma opisanog prethodnim pesudokodom je $O(V^2)$.

Uporabom gomile (*heap*) za ToBeCh algoritam se ubrzava do $O(E+V \cdot \log_2 V)$. (vidi Drozdek, Cormen...)

Primjer za vježbu: polazni vrh je **d**

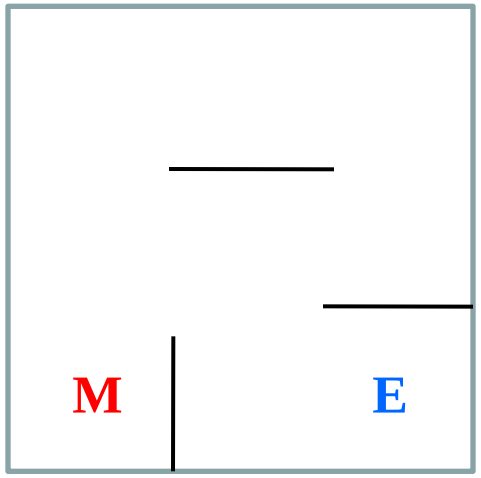
Dijkstra



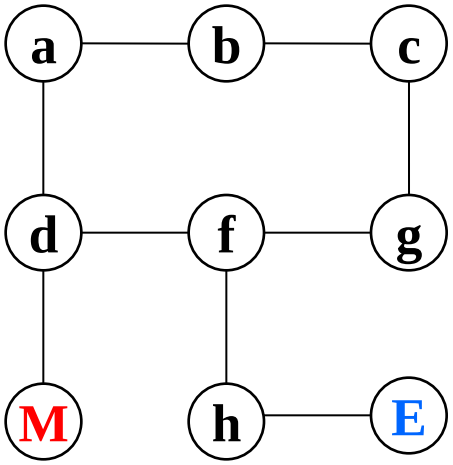
	a	b	c	d	e
a	0	3	2	1	∞
b	4	0	∞	∞	∞
c	∞	2	0	3	∞
d	∞	2	∞	0	4
e	1	∞	4	∞	0

vrh	d_{min}	put
a	5	d→e→a
b	2	d→b
c	7	d→e→a→c
d	0	
e	4	d→e

Primjer za vježbu: najkraći izlazak iz labirinta; pretvorba labirinta u graf



1	1	1	1	1	1	1			
1	0	0	0	0	0	1			
1	0	0	1	0	0	1			
1	0	0	0	0	0	1			
1	0	0	0	0	1	1			
1	M	1	0	0	E	1			
1	1	1	1	1	1	1			



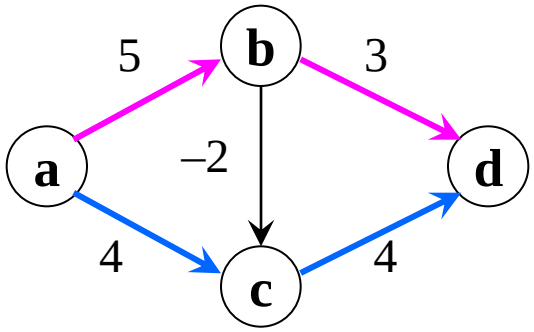
	a	b	c	d	E	f	g	h	M
a	0	1	∞	1	∞	∞	∞	∞	∞
b	1	0	1	∞	∞	∞	∞	∞	∞
c	∞	1	0	∞	∞	∞	1	∞	∞
d	1	∞	∞	0	∞	1	∞	∞	1
E	∞	∞	∞	∞	0	∞	∞	1	∞
f	∞	∞	∞	1	∞	0	1	1	∞
g	∞	∞	1	∞	∞	1	0	∞	∞
h	∞	∞	∞	∞	1	1	∞	0	∞
M	∞	∞	∞	1	∞	∞	∞	∞	0

Bellman-Fordov algoritam

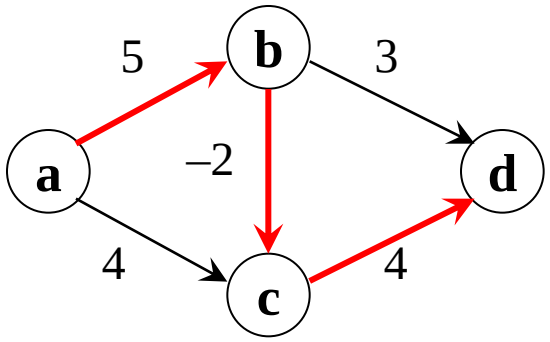
Richard Bellman: *On a Routing Problem*, Quarterly of Applied Mathematics, 16(1), pp.87-90, 1958.
Lestor R. Ford jr., D. R. Fulkerson: *Flows in Networks*, Princeton University Press, 1962.

- *label-correcting* metoda
- primjenjiv i kada u grafu ima bridova negativne težine
- ograničenje – u grafu ne smije biti negativnih ciklusa

Problem: Dijkstrin algoritam u grafu s negativnim težinama



a→d: Dijkstra
Prvo će pronaći a→c→d, a potom i a→b→d, ukupne duljine 8.



a→d:
Točno rješenje je a→b→c→d, ukupne duljine 7.

Rješenje?

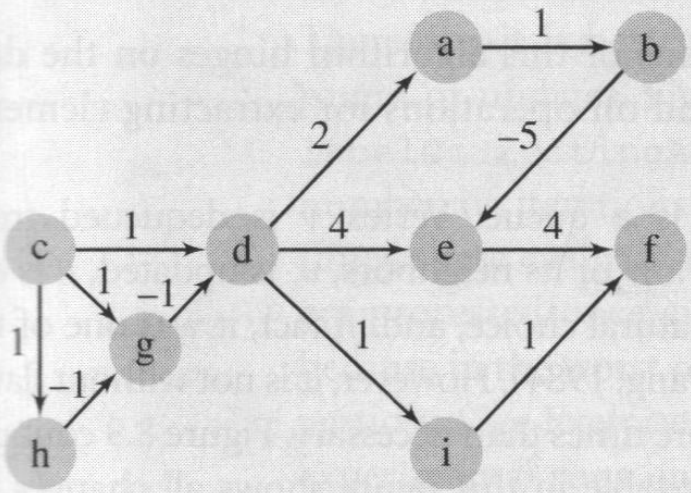
Uvijek provjeravati sve bridove, sve dok se makar jedna udaljenost među vrhovima mijenja (smanjuje) tijekom obilaska svih vrhova.

Zbog toga se najkraći put do nekog vrha može saznati tek nakon određivanja najkraćih udaljenosti do svih vrhova!

Bellman-Fordov algoritam: osnovna ideja

```
BellmanFord (graph)
initialisation: for all vertices d(v)= inf   // d(v) = (source,v)
d(source) = 0;
while there is an edge (u,v) such that
                              d(u)+edge(u,v) < current d(v)
      d(v) = d(u)+edge(u,v);
      predecessor(v) = u;
```

Primjer:



the order of edges: *ab be cd cg ch da de di ef gd hg if*

	iteration				
	init	**1**		**2**	**3**	**4**
a	∞	3		2	1	
b	∞			4	3	2
c	0					
d	∞	1	0	−1		
e	∞	5		−1	−2	−3
f	∞	9	3	2	1	
g	∞	1	0			
h	∞	1				
i	∞	2		1	0	

Složenost: O(V·E).

Točnije, može biti najviše (V–1)·E koraka jer najduži put (koji se ne vraća unatrag) u grafu s V vrhova može imati najviše V–1 bridova, a svakim obilaskom svih bridova rješavaju se putanje za jedan brid duže.

Bellman-Fordov algoritam: pseudokod brže inačice

- u ovoj inačici ne obilaze se svaki puta svi bridovi, nego se obilaze vrhovi i to samo oni kojima se u prethodnom obilasku promijenila udaljenost jer se, osim njima, mogla promijeniti samo još njihovim susjedima

```
BellmanFord (graph)

initialisation: for all vertices d(v)= inf    // d(v) = d(source,v)
d(source) = 0;
ToBeCh = source;          // all vertices
//ToBeCh = popis onih kojima se promijenila udaljenost, tj.
//onih čije susjede treba provjeriti.
while there are vertices in ToBeCh      //... is not empty
      v = a vertex in ToBeCh;           //with the least d(v);
      remove v from ToBeCh;
      for all vertices u adjacent to v          //and in ToBeCh
            dnew(u)=d(v)+edge(v,u);  //moguća nova d(u)
            if dnew(u) < d(u)
                  d(u) = dnew(u);
                  predecessor(u) = v;      //prethodnik
                  add u to ToBeCh if it is not already there
```

Ovaj posljednji korak ne postoji u Dijkstrinom algoritmu!

Najkraći put između svih parova vrhova (*All-to-All Shortest Path*)

- najjednostavnije rješenje: Dijkstrin ili Bellman-Fordov algoritam |V| puta (svaki vrh je jednom polazni)
- to je i najbolje rješenje ako je broj bridova relativno malen u odnosu na mogući broj bridova, tj. kada je graf rijedak
- nad gustim grafovima učinkovitiji su "univerzalni" algoritmi, npr. Warshall-Floyd-Ingermanov (složenost $O(V^3)$)

Warshall-Floyd-Ingermanov algoritam: teorijska pozadina

- postupno proširuje skup vrhova koji se koriste (mogu poslužiti) kao *međuvrhovi* (*intermediate vertices*)

Neka je zadan graf G čiji su vrhovi (njihovi redni brojevi) $V = \{1, 2, ..., n\}$ i neka je $\{1, 2, ..., k\}$ k-člani podskup od V, dakle $k \leq n$. Uzmimo sada bilo koja dva vrha v_i i v_j iz V te promatrajmo sve puteve između njih (od v_i do v_j) koji prolaze isključivo vrhovima iz skupa $\{1, 2, ..., k\}$ (međuvrhovi).

Označimo s *p* najkraći od tih puteva (to ne mora biti ujedno i najkraći put između tih vrhova u cijelom grafu!). Zapažamo dvije činjenice:

1. Ako *k* (vrh kojemu je to redni broj) nije međuvrh na putu *p*, onda su svi međuvrhovi na putu *p* u skupu {1, 2, ..., k – 1}.
   Iz tog zapažanja slijedi da je najkraći put od *i* do *j* kojemu su svi međuvrhovi iz skupa {1, 2, ..., k} jednak najkraćem putu kojemu su svi međuvrhovi iz skupa {1, 2, ..., k – 1} pa zaključujemo da za najmanju udaljenost v_i i v_j vrijedi $d_{ij}^{k} = d_{ij}^{k-1}$.

2. Ako *k* jest na putu *p*, onda put *p* možemo razdijeliti na dvije dionice: p1 = dionica *i* ... *k* te p2 = dionica *k* ... *j* (*k* je zajednički objema dionicama). No, sada je sigurno da je p1 najkraći put od *i* do *k* među svim putevima koji prolaze međuvrhovima iz {1, 2, ..., k – 1}, a isto tako je p2 najkraći put od *k* do *j* među svim takvim putevima. Slijedi da za najmanju udaljenost v_i i v_j putevima preko vrhova iz skupa {1, 2, ..., k} vrijedi $d_{ij}^{k} = d_{ik}^{k-1} + d_{kj}^{k-1}$.

Na temelju prethodnih zapažanja, najmanja udaljenost dvaju vrhova u grafu može se izračunati rekurzivno po formuli

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

iz koje slijedi Warshall-Floyd-Ingermanov algoritam.

```
WFIdistances (W) //W =adjacency matrix of the graph
initialisation of matrix D;      //matrica udaljenosti D⁰ = W
for k = 1 to |V|                    //skup međuvrhova
      for i = 1 to |V|                //polazni vrh
            for j = 1 to |V|             //završni vrh
                if D[i,k] + D[k,j] < D[i,j]
                      D[i,j] = D[i,k] + D[k,j];
```

Po završetku postupka, u matrici D će biti najmanje udaljenosti između svih parova vrhova. A putevi?

Putevi se (relativno nespretno) mogu odrediti iz D, ali bolje je paralelno s D konstruirati matricu puteva Π (veliko π) čiji je svaki element π_{ij} prethodnik vrha v_j na putu koji započinje u vrhu v_i. Matrica Π se mijenja tijekom cijele WFI procedure pa se stanje u k-tom koraku (skup međuvrhova {1, ... ,k}) označava s π_{ij}^{k}. Elementi Π se mijenjaju pod istim uvjetima kao i njihovi "parovi" u D i također računaju rekurzivno po formulama:

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

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

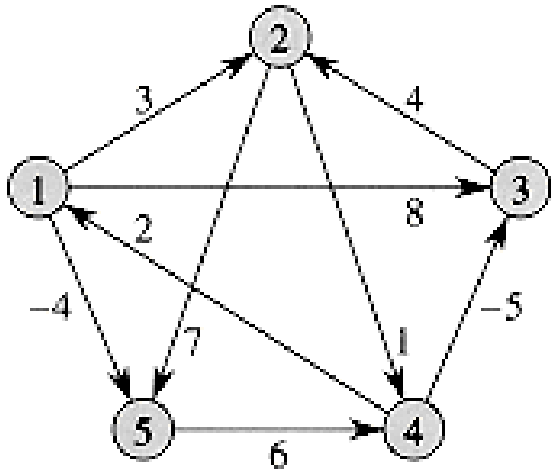
Time dolazimo do potpunog WFI algoritma.

```
WFI (W)                    //W=adjacency matrix of the graph
initialization of matrix D;      //matrica udaljenosti D^0 = W
initialization of matrix Path;     //matrica puteva π_ij^0
for k = 1 to |V|                        //skup međuvrhova
      for i = 1 to |V|                    //polazni vrh
            for j = 1 to |V|                  //završni vrh
                  if D[i,k] + D[k,j] < D[i,j]
                        D[i,j] = D[i,k] + D[k,j];
                        Path(i,j) = Path(k,j);
```

Da bi ovaj algoritam radio ispravno, petlja za promjenu broja vrhova u skupu međuvrhova mora obilaziti vrh po vrh korakom =1.

Svako kasnije rješenje se dobiva iskorištavanjem prethodnih, koja su do tada bila optimalna (najbolja moguća), što ovaj algoritam čini predstavnikom strategije koja se naziva dinamičko programiranje (*Dynamic programming*).

Primjer (iz Cormen...):



Npr. 3→5

Put: 3→2→4→1→5

Duljina: 3

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

Osim najkraćih udaljenosti (puteva), WFI algoritam može poslužiti i za dublju analizu grafa.

1. Ako se element d_{ij} matrice D, početno postavljen na $=\infty$, ne promijeni do kraja, znači da se iz polaznog vrha v_i uopće ne može doći u odredišni v_j. Time se dobiva osnovni uvid u povezanost vrhova u grafu, odnosno određuje tzv. *transitive closure* grafa (vidi Cormen ...).

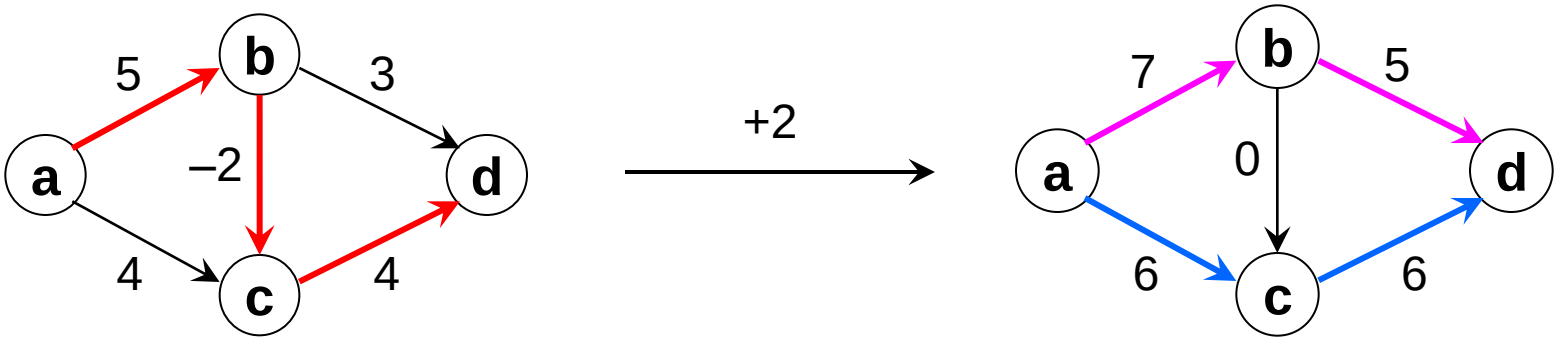
2. Iz prethodnog zaključka, uz $i=j$, slijedi da ako se dijagonalni elementi matrice D početno postave (inicijaliziraju) na $=\infty$ umjesto na =0, WFI algoritam otkriva postojanje ciklusa u grafu.
   - ako na kraju neki dijagonalni element više nije $=\infty$, nađen je obilazak (put) koji počinje i završava u istom vrhu, dakle ciklus
   - znamo samo duljinu ciklusa i vrh koji je u njemu, ne i cijeli ciklus

Rijetki grafovi: Dijkstrin ili Bellman-Fordov algoritam

- bolje Dijkstrin zbog brzine, ali problem su negativne težine

Transformacija - preslikavanje svih težina u pozitivne brojeve.

- jedini je uvjet da međusobni odnosi duljina puteva ostanu isti (oni koji su bili duži s izvornim težinama moraju ostati duži i s transformiranima, i obratno)
- jednostavna transformacija kojom bi se svim izvornim težinama dodao isti broj, dovoljan da sve postanu pozitivne, nije ispravna



Najkraći: a→b→c→d

Najkraći: a→b→d
a→c→d

Do ispravne transformacije se dolazi promatranjem najmanje moguće udaljenosti nekog vrha *v* od nekog drugog (referentnog, polaznog) vrha. Ona je sigurno manja ili jednaka zbroju najmanje moguće udaljenosti nekog susjeda *u* vrha *v* i duljine brida (*u*,*v*).

$$d(v) \le d(u) + w(u,v) \qquad / - d(v)$$

$$0 \le d(u) + w(u,v) - d(v) = w'(u,v)$$

Zaključujemo da su veličine $w'(v_i,v_j) = w(v_i,v_j) + d(v_i) - d(v_j)$ uvijek veće ili jednake nula (w' = transformirana težina).
Uzmimo sada bilo koja dva vrha, recimo v_1 i v_k, i neka je najkraća veza među njima niz $v_1, v_2, \ldots, v_k$ (indeksi su redni brojevi na putu, a ne u grafu). Zamijenimo li težine bridova na tom putu transformiranim težinama w', duljina puta L' od v_1 do v_k u transformiranom grafu bit će

$$L'(v_1,v_k) = \sum_{i=1}^{k-1} w'(v_i,v_{i+1}) = w(v_1,v_2) + d(v_1) - d(v_2)$$

$$+ w(v_2,v_3) + d(v_2) - d(v_3) + \ldots$$

odnosno,

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

gdje je $L(v_1,v_k)$ duljina najkraćeg puta između v_1 i v_k u izvornom grafu. Iz gornje relacije (izvoda) isčitavamo da je najkraći put u transformiranom grafu jednak najkraćem putu u izvornom grafu, s time da je duljina puta uvećana za konstantu koja NE ovisi o stazi, nego samo o polaznom i završnom vrhu. To znači da svi putevi ostaju u istom međusobnom odnosu!

Naime, svaka staza u transformiranom grafu ima svoj par u izvornom grafu. Kad bismo u transformiranom grafu otkrili neki novi najkraći put između v_1 i v_k, u izvornom grafu bi morao postojati njegov par. Budući da je transformacija preslikavanje “jedan u jedan”, novi najkraći put bi značio postojanje “novog” najkraćeg puta u izvornom grafu, a takvog nema.

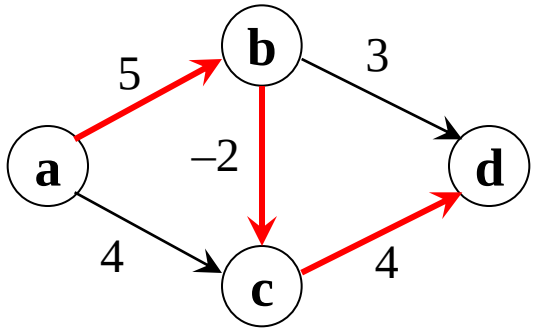
Transformacija u oba smjera:

$$L'_{ik} = L_{ik} + d_i - d_k \quad \Leftrightarrow \quad L_{ik} = L'_{ik} + d_k - d_i$$

No, da bi se transformacija mogla provesti, potrebno je znati udaljenosti svih vrhova od nekog (bilo kojeg) vrha, a to znači da prvo moramo primijeniti neku *label-correcting* metodu polazeći od vrha koji ima vezu sa svim drugim vrhovima grafa (referentni vrh). Prema tome, ukupni je slijed postupaka za pronalaženje najkraćih puteva u rijetkom grafu sljedeći:

1. Bellman-Fordovim algoritmom odrediti najkraće udaljenosti svih vrhova od nekog proizvoljnog (referentnog) vrha koji ima vezu sa svim drugim vrhovima grafa.
2. Transformirati graf $G \rightarrow G'$; $w'(v_i,v_j) = w(v_i,v_j) + d(v_i) - d(v_j)$.
3. $|V|$ puta primijeniti Dijkstrin algoritam, svaki puta mijenjajući polazni vrh tako da svi vrhovi jednom budu polazni.
4. Izvršiti obrnutu transformaciju da se dobiju izvorne udaljenosti: $L_{ik} = L'_{ik} + d_k - d_i$.

Primjer:



Bellman-Ford (polazni =a):

d(a) = 0, d(b) = 5

d(c) = 3, d(d) = 7

Transformacija: $w'(v_i,v_j) = w(v_i,v_j) + d(v_i) - d(v_j)$

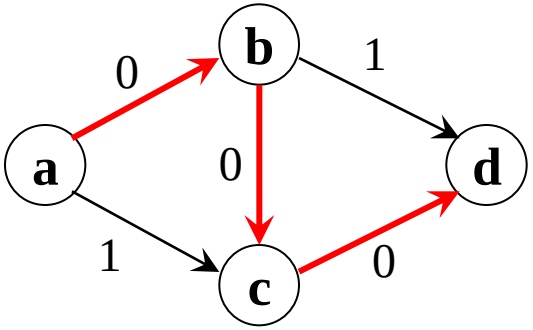
$w(a,b) = 5 \rightarrow w'(a,b) = 5 + 0 - 5 = 0$

$w(a,c) = 4 \rightarrow w'(a,c) = 4 + 0 - 3 = 1$

$w(b,c) = -2 \rightarrow w'(b,c) = -2 + 5 - 3 = 0$

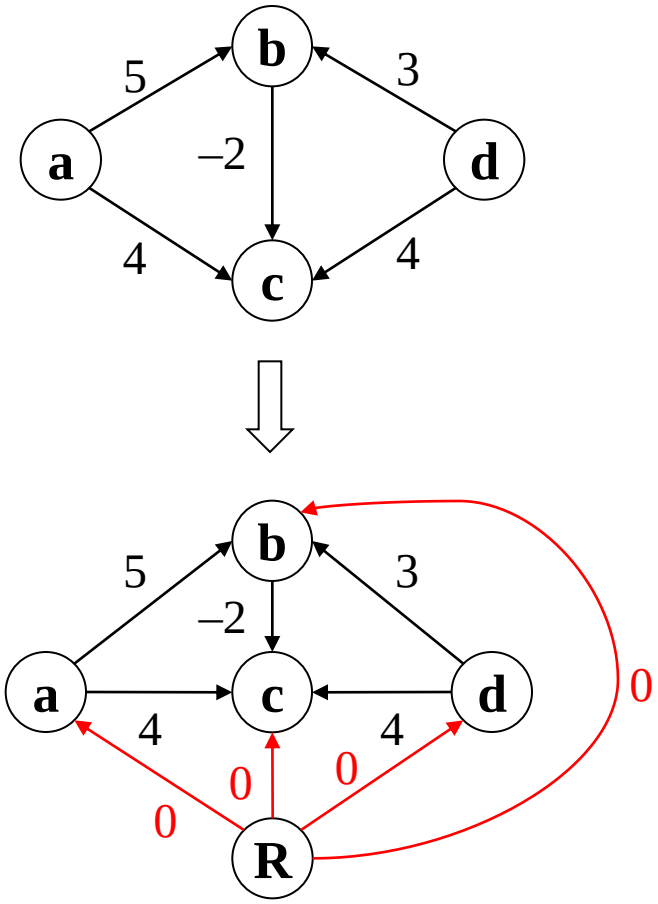
$w(b,d) = 3 \rightarrow w'(b,d) = 3 + 5 - 7 = 1$

$w(c,d) = 4 \rightarrow w'(c,d) = 4 + 3 - 7 = 0$



U transformiranom grafu najkraći putevi su isti kao u izvornom.
Na primjer, put iz a u d je opet a→b→c→d, samo mu je duljina u transformiranom grafu L' = 0.
Izvorna duljina dobiva se iz L = L' + d(d) – d(a) = 0 + 7 – 0 = 7.

Nema vrha koji ima vezu sa svim drugim vrhovima?



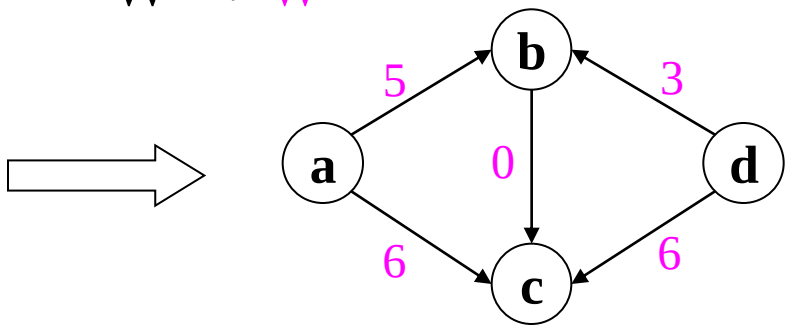
1. Dodati novi (“umjetni”) vrh u graf i povezati ga usmjerenim bridovima proizvoljne težine (najprikladnije =0) sa svim izvornim vrhovima.
2. Odrediti najkraće udaljenosti od novog do svih izvornih vrhova (Bellman-Ford) i nakon toga je transformacija moguća, pri čemu je novi vrh referentna točka.

Bellman-Ford (referentni =R):

d(a) = 0, d(b) = 0

d(c) = –2, d(d) = 0

W → W ’



Detekcija ciklusa (Cycle Detection)

- vrlo važna operacija u složenijim algoritmima nad grafovima (npr. najmanja razapinjajuća stabla, Eulerovi ciklusi itd.)
- najjednostavnije rješenje: DFS ili BFS s detekcijom povratnog brida

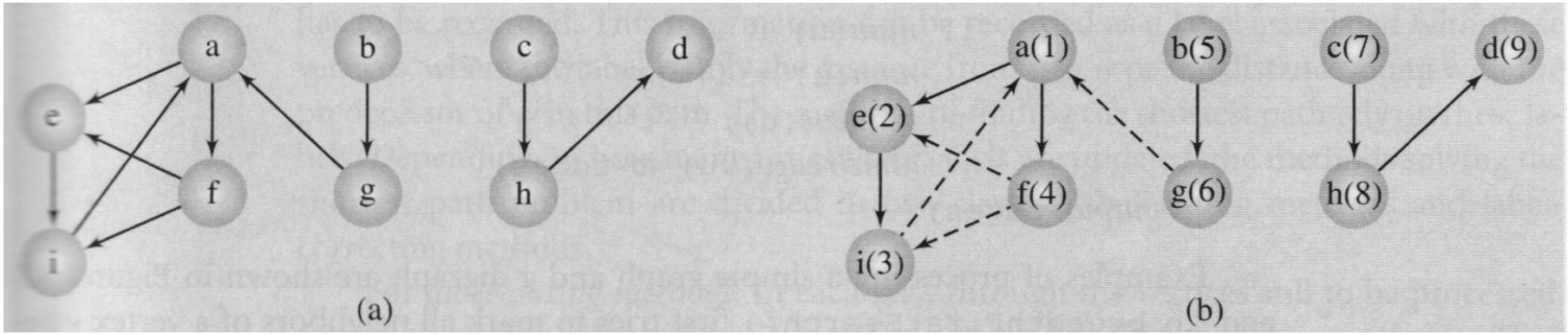
Npr. DFS kao osnova - neusmjereni grafovi

- proceduri `DFS(v)` samo treba dodati `else` na kraju

```
CycleDetectionDFS (v)    //neusmjereni grafovi
num(v) = ++korak;   //korak:statička,argument,globalna
for svi susjedi u vrha v
     if num(u)==0    //ako je to neobiđeni susjed
          prethodnik(u) = v;              //neobavezno
          CycleDetectionDFS(u);
     else
          detektiran ciklus;
```

U usmjerenim grafovima prethodna procedura može pogrešno detektirati ciklus zbog mogućih veza između različitih stabala u grafu (npr. brid (g,a) grafa u primjeru za DFS).

Budući da je ‘a’ prethodno već obiđen, brid g-a bi bio pogrešno detektiran kao dio ciklusa.



- rješenje je označiti sve vrhove istog stabla jer ciklus nastaje samo zbog povratnog brida u isto stablo
- svaki se vrh označi kad se prvi puta posjeti, a oznaka se briše tek kad obiđemo cijelo njegovo podstablo
- u ilustracijama se kao oznake često koriste boje pa se algoritam naziva i *coloured* DFS

```
CycleDetectionDigraphDFS (G)
inicijalizacija: svim vrhovima num(v)=0
korak = 0;                    //je li potrebno ovisi o DFS(v)
while u grafu G još ima neobiđenih vrhova v  //num(v)==0
    CycDetDigraphDFS(v);


CycDetDigraphDFS (v)                  //usmjereni
num(v) = ++korak;   //korak:statička,argument,globalna
flag = true;          //Označavanje vrha, slijedi
for svi susjedi u vrha v    //obrada njegovog podstabla.
    if num(u)==0      //ako je to neobiđeni susjed
        prethodnik(u) = v;        //ako želimo put
        CycDetDigraphDFS(u);
    else if  flag == true
        detektiran ciklus;
flag = false;                  //Važno - brisanje oznake!
```

Prethodna rješenja nisu prikladna za česte provjere je li neki određeni brid dio ciklusa. Kad bi se to, na primjer, htjelo učiniti za sve bridove u grafu zasebno, složenost postupka bi bila $O(|E|)\cdot O(DFS) = O(|E|)\cdot O(|V|+|E|)$, što za guste grafove može postati $O(|V|^4)$.

Pitanje je li neki brid dio ciklusa je zapravo pitanje pripadnosti tog brida, odnosno njegovih rubnih vrhova, nekom podskupu grafa. Kad su u pitanju grafovi, najbrže rješenje je rastavljanje grafa na blokove (neusmjereni grafovi), odnosno SCC (usmjereni grafovi), što se obrađuje malo dalje u tekstu. Nakon što se odrede blokovi (sastavnice) grafa, tj. vrhovi razvrstaju u podskupe, lako se i brzo može ustanoviti koji su bridovi u ciklusima.

No, posve općenito rješenje problema određivanja pripadnosti dvaju elemenata nekom skupu, odnosno razvrstavanja u skupine, je primjena **Union-Find** strukture.

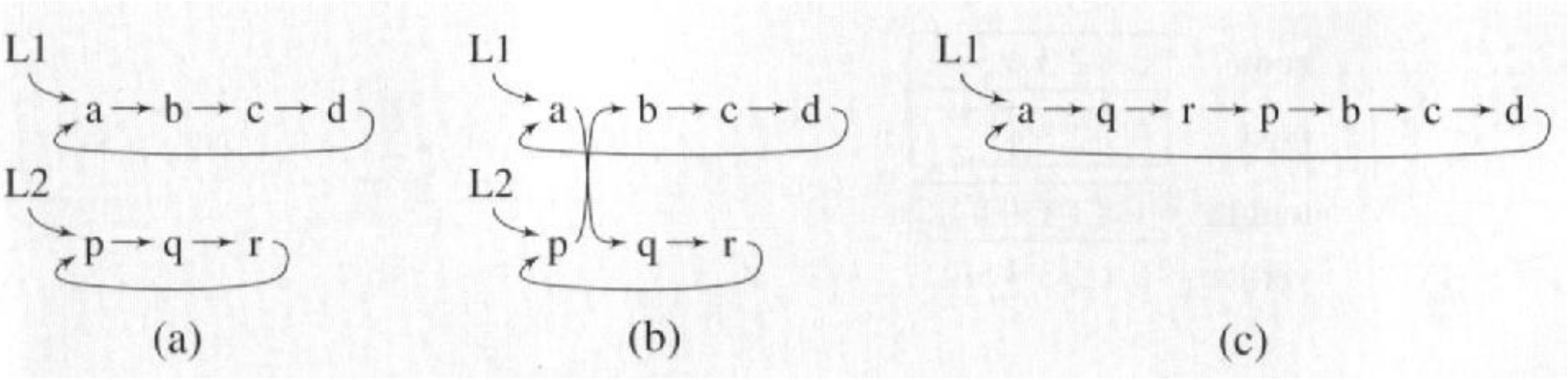
Union-find struktura: rješava pitanje pripadnosti dvaju elemenata istom skupu te, po potrebi, sjedinjenja njihovih skupova.

Treba riješiti dvije operacije:

a) Find – kojem skupu pripada neki element (to ujedno rješava i provjeru jesu li dva iz istog skupa)

b) Union – sjedinjenje dvaju skupova u jedan (*combine*, *merge*)

Skupovi kojima se rješava *union-find* problem logički trebaju odgovarati kružnim (*circular*) listama radi bržeg sjedinjenja.



Za ostvarenje *union-find* strukture, prikladno je uvesti tri polja. Indeksi odgovaraju rednim brojevima vrhova u grafu, a elementi su:

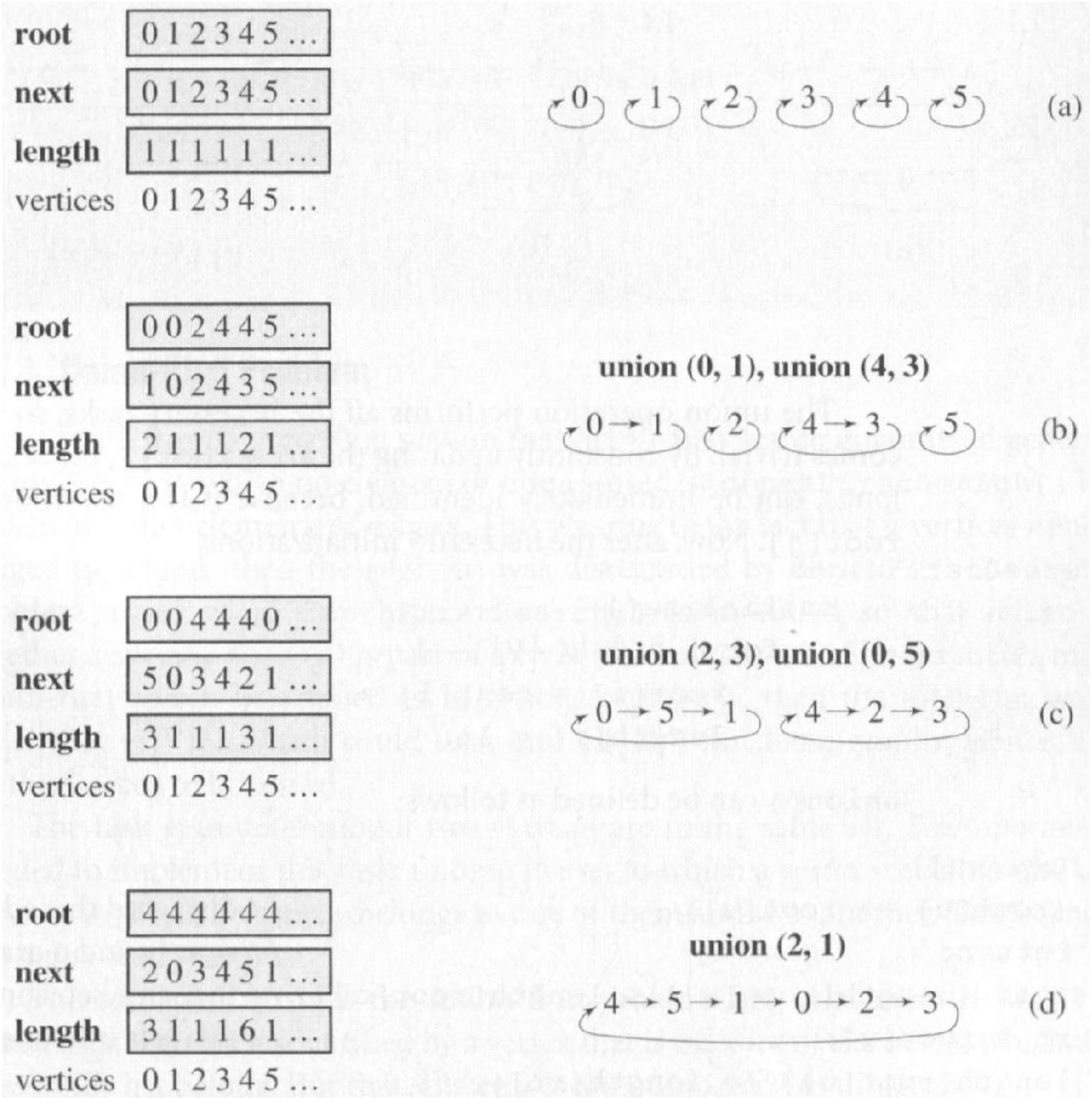
`1.root` – redni brojevi vodećih (identifikacijskih) vrhova skupova kojima pripadaju vrhovi zadanog rednog broja

2. `next` – redni brojevi sljedbenika (u istom skupu) vrha rednog broja jednakog indeksu u polju
   – upravo upisivanjem odgovarajućih sljedbenika se od "običnih" polja (lista) stvara kružne liste

`3.length` – duljine popisa (brojnost skupova) kojima pripada vrh rednog broja jednakog indeksu u polju
   – potrebno prilikom sjedinjavanja dva skupa da se zna koji je malobrojniji
   – dovoljno je osvježavati podatak za identifikacijski vrh

Prilikom sjedinjenja, manji se skup (kraći popis; zato je potrebno polje `length`) pridružuje duljem jer tako ima manje vrhova kojima treba osvježiti identifikacijski element skupa (u polju `root`) pa je cijela procedura kraća i brža.

Primjer:


root 0 1 2 3 4 5 ...
next 0 1 2 3 4 5 ...
length 1 1 1 1 1 1 ...
vertices 0 1 2 3 4 5 ...
0 1 2 3 4 5
(a)
root 0 0 2 4 4 5 ...
next 1 0 2 4 3 5 ...
length 2 1 1 1 2 1 ...
vertices 0 1 2 3 4 5 ...
union (0, 1), union (4, 3)
0 → 1 2 4 → 3 5
(b)
root 0 0 4 4 4 0 ...
next 5 0 3 4 2 1 ...
length 3 1 1 1 3 1 ...
vertices 0 1 2 3 4 5 ...
union (2, 3), union (0, 5)
0 → 5 → 1 4 → 2 → 3
(c)
root 4 4 4 4 4 4 ...
next 2 0 3 4 5 1 ...
length 3 1 1 1 6 1 ...
vertices 0 1 2 3 4 5 ...
union (2, 1)
4 → 5 → 1 → 0 → 2 → 3
(d)

```
initialize()
for i = 1 to |V|        //ili ... 0 to |V|−1
      root[i] = next[i] = i;
      length[i] = 1;

UnionFind (edge(u,v))
rt1 = root[u];  rt2 = root[v];
if (rt1 == rt2)        //Find dio - (ne)potrebno ovisno
      return;                    //o konkretnom programu
else if (length[rt1] < length[rt2])   //Union dio.
      length[rt2] += length[rt1];
      root[rt1] = rt2;    //novi korijen korijena rt1
      for (i=next[rt1]; i != rt1; i=next[i])
            root[i] = rt2;    //novi korijen ostalih
else                                  //u manjem skupu
      //isto kao u prethodnom slučaju,
      //ali sa zamijenjenim rt1 i rt2
swap (next[rt1], next[rt2]);    //sjedinjenje
```

Vrijeme izvršavanja UnionFind procedure ovisi o broju ponavljanja `for`-petlje, a ona se izvršava od jednom do najviše |V|/2 puta pa je složenost jednaka O(V).

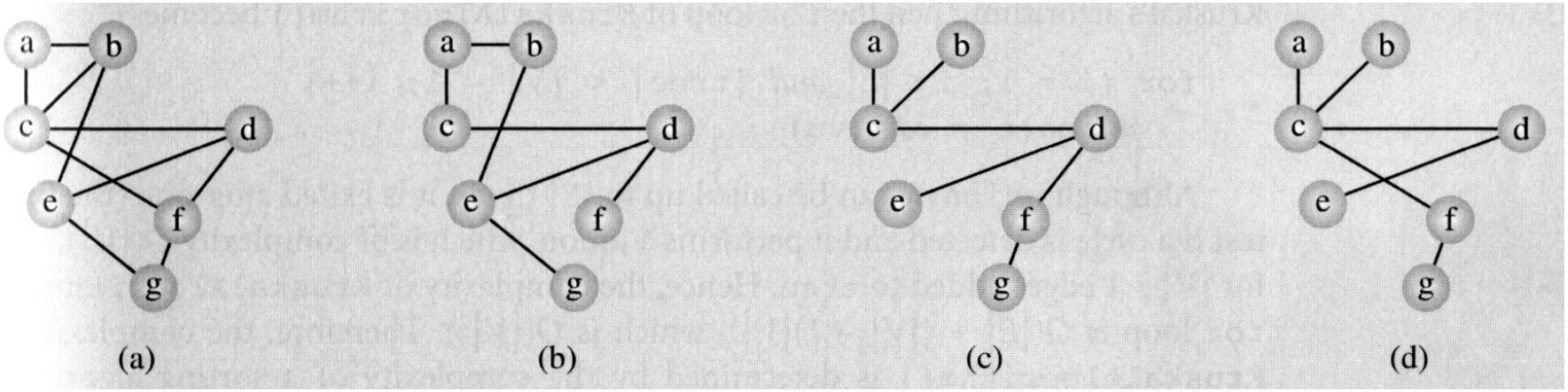
Prikladnost UnionFind strukture za detekciju pripadnosti nekog brida ciklusu ovisi o namjeni algoritma. UnionFind je ponajbolje rješenje kad se treba ustanoviti pripadnost jednom ciklusu (skupu), a bridovi dolaze na obradu jedan po jedan, kao npr. u Kruskalovom algoritmu (malo dalje u tekstu) jer tada ih se lako pridružuje postojećem skupu, nakon čega su i oni spremni za brzu provjeru sljedećeg brida. No, u grafovima su jedni te isti bridovi često u više različith ciklusa, a ako se treba ispitivati pripadnost različitim ciklusima (podskupima), bolje je graf unaprijed pripremiti (*preprocessing*) nekim od algoritama za određivanje povezanosti u grafu (rastavljanje grafa na blokove, odnosno SCC).

Najkraća razapinjajuća stabla (*Minimum Spanning Trees* - MST)

Koji su to bridovi, od svih u grafu, koje treba zadržati, a da svi vrhovi budu međusobno povezani?

- npr. koje su zrakoplovne linije potrebne, a da postoji veza među svim gradovima koje se promatra

Više je mogućih rješenja, a svako čini jedno razapinjajuće stablo (*Spanning Tree* = ST = stablo koje povezuje sve vrhove minimalnim brojem bridova):



Koje je rješenje najbolje, tj. za koje je ukupna duljina svih bridova razapinjajućeg stabla (zrakoplovnih linija) najmanja?
Problem = pronaći najkraće razapinjajuće stablo (MST).

Teorijski, razapinjajuće stablo u grafu s |V| vrhova imat će točno |V| – 1 bridova.

Najkraće razapinjajuće stablo (MST) - razapinjajuće stablo najmanjeg ukupnog zbroja duljina (težina) svih bridova.

- ne postavlja se uvjet da svaki vrh bude obiđen samo jednom, ali za bridove to vrijedi; svakim se bridom smije proći samo jednom

Razapinjajuće stablo je (usputni) rezultat DFS i BFS pa ako je graf bestežinski (svi bridovi jednake težine), svako rješenje DFS ili BFS je ujedno i MST.

Za težinske grafove pronađeno je više rješenja.
Šire primjenjivani su Kruskalov i Dijkstrin, kao "najbliži srodnici", te Primov algoritam.

Kruskalov algoritam

Joseph. B. Kruskal: *On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem*,

Proceedings of the American Mathematical Society, Vol 7, No. 1 (Feb, 1956), pp. 48–50

```
Kruskal (graph)
tree = null;                          //inicijalizacija MST
edges = all edges of graph sorted by weight;     //uzlazno
for (i=1; i≤|E| and |tree|<|V|-1; ++i)
    if e_i from edges does not form a cycle with edges in tree
            add e_i to tree;
```

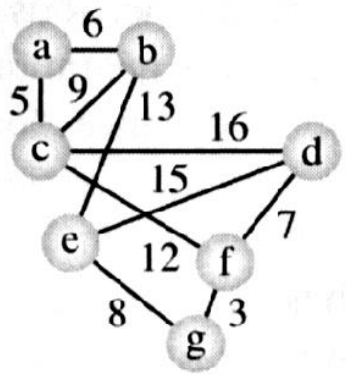
Složenost ovisi o sortiranju i metodi detekcije ciklusa.

Pretpostavka: brzo sortiranje = $O(E \cdot \log_2 E) = [E < V^2] = O(E \cdot 2 \cdot \log_2 V)$

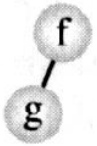
$= O(E \cdot \log_2 V) = O(V^2 \cdot \log_2 V)$.

Najveći broj ponavljanja for-petlje Kruskalovog algoritma = O(E). Ako se detekcija ciklusa i dodavanje obavlja s UnionFind(), detekcija ciklusa će biti O(1), a dodavanje brida u MST, koje će se obaviti točno V–1 puta, će biti O(Union) = O(V). Dakle, petlja će obaviti $E \cdot O(1)+O((V–1) \cdot V)=O(E+V^2–V)=O(V^2+V^2–V)=O(V^2)$ operacija. Ukupno: $O(E \cdot \log_2 V) + O(V^2) = O(E \cdot \log_2 V)$.

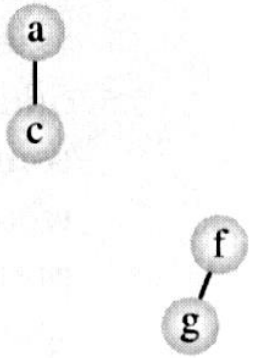
Primjer Kruskal:



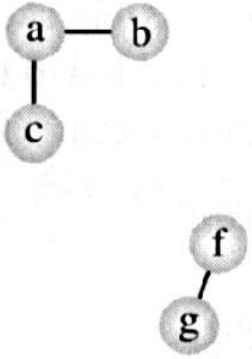
(a)



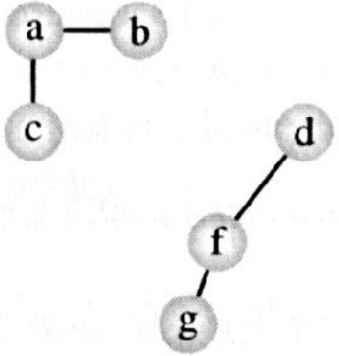
(ba)



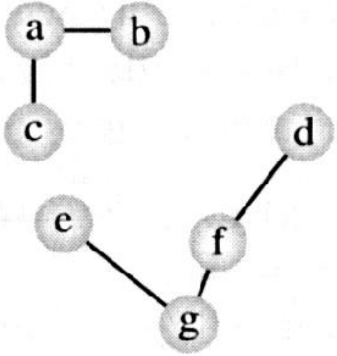
(bb)



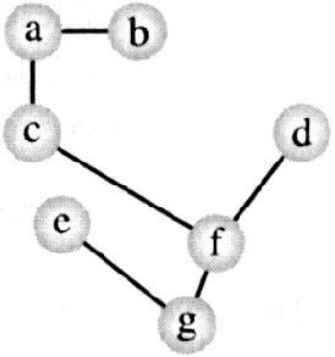
(bc)



(bd)



(be)



(bf)

Dijkstrin algoritam

E. W. Dijkstra: Some Theorems on Spanning Subtrees of a Graph,
Indagationes Mathematicae 28 (1960), 196–199

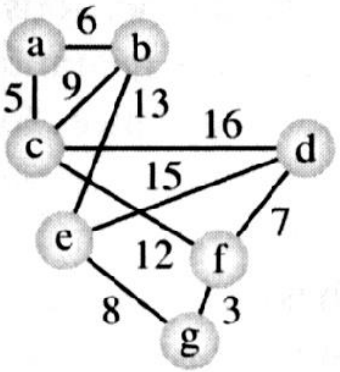
Nedostatak Kruskalovog – potreba za sortiranjem bridova.

Dijkstrin to ne zahtijeva.

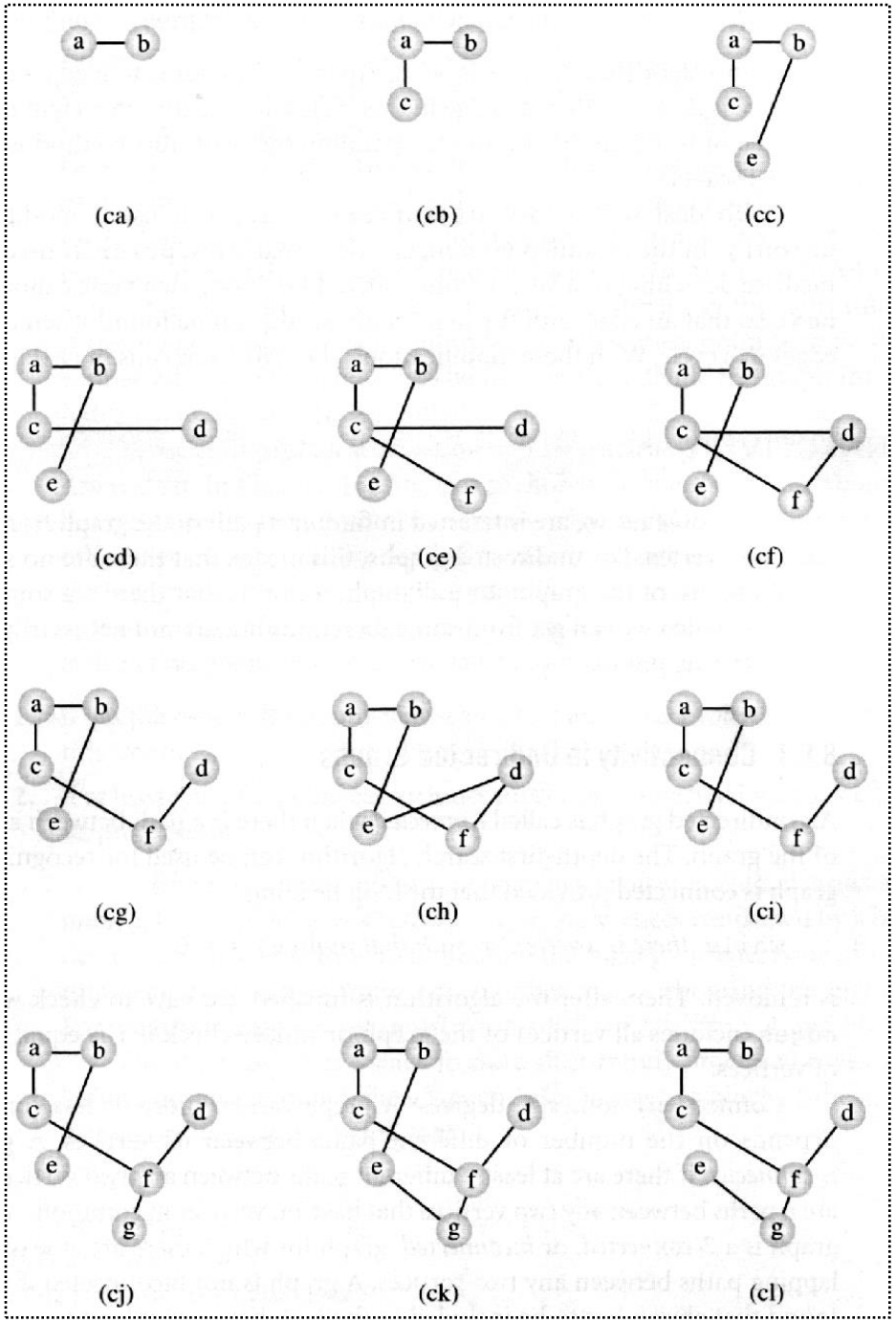
```
DijkstraMST (graph)
tree = null;                          //inicijalizacija MST
edges = all edges of graph;           //bilo koji poredak
for i=1 to |E|
      add e_i to tree;
      if there is a cycle in tree
            remove the maximum weight edge from the cycle;
```

Najbrža inačica: složenost $O(E \cdot V)$.

Primjer Dijkstra:



(a)



Primov algoritam

Izvorna ideja: V. Jarník: *O jistém problému minimálním*,
Práce Moravské Přírodovědecké Společnosti, 6, 1930, pp. 57-63. (in Czech)

Razrada i program: R. C. Prim: *Shortest connection networks and some generalisations*,
Bell System Technical Journal, 36 (1957), pp. 1389–1401

```
Prim (graph)
newG = arbitrary starting vertex;          //pomoćni skup vrhova
tree = null;                               //skup bridova MST
while (number of vertices in newG) < |V|
      choose a vertex v from graph (an edge(u,v)),
            which is closest to vertices in newG;
      add v to newG;
      add edge(u,v) to tree;               //'u' is the vertex
          //in newG to which v is the closest neighbour
```

Najbrža inačica: složenost $O(E + V \cdot \log_2 V)$.

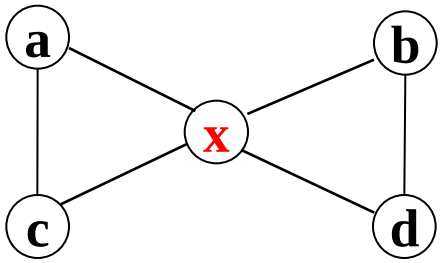
Povezanost grafa (*Connectivity*)

Povezani graf je graf u kojem postoji put između svaka dva vrha.
- različito se postupa s usmjerenim i neusmjerenim grafovima

Neusmjereni grafovi

Neusmjereni graf je povezan čim (ako) postoji razapinjajuće stablo koje obuhvaća sve vrhove grafa, što se lako može ustanoviti pomoću BFS ili DFS (npr. u DFS je dovoljno ukloniti while petlju i na kraju pogledati jesu li obiđeni svi vrhovi).

Razlikujemo stupnjeve povezanosti.
Graf je *n*-povezan ako postoji barem *n* različitih puteva između svaka dva vrha, s time da ti putevi nemaju zajedničkih vrhova.
Prijelomna točka (artikulacijska; *articulation point*) je vrh koji (čije uklanjanje) dijeli graf na nepovezane dijelove.



Vrh X je prijelomna točka.

Most (*bridge*, *cut-edge*) je brid koji spaja dva neovisna (odvojena) dijela grafa.

Blok ili dvostruko povezana sastavnica (*biconnected component*) je povezani (pod)graf u kojem nema ni prijelomnih točaka ni mostova.

Podjela grafa na blokove svodi se na traženje prijelomnih točaka, a njih nalazimo preinačenim DFS alogritmom.
Prijelomna točka je onaj vrh razapinjajućeg DFS stabla koji ima barem jedno podstablo koje ne stvara ciklus koji obuhvaća promatrani vrh (kandidat za prijelomnu točku). Preciznije, to je vrh koji ima barem jedno podstablo iz kojeg nema povratnih bridova do vrhova koji u razapinjajućem DFS stablu prethode promatranom vrhu (kandidatu za prijelomnu točku).
Prijelomne točke imaju dva svojstva koja omogućuju njihovo optičko prepoznavanje, tj. vrh je prijelomna točka ako:

1. je korijen DFS razapinjajućeg stabla i ima više od jednog djeteta
2. barem jedno njegovo podstablo nema nikakvu povratnu vezu s njegovim prethodnicima

Algoritamsko prepoznavanje prijelomnih točaka zahtijeva da svaki vrh "zna" koji je najdalji doseg unatrag svih njegovih podstabala. Da bi se to postiglo, tijekom poniranja u DFS rekurziji svakom se vrhu `v` upisuju redoslijed obilaska `num(v)` i najdalji doseg unatrag svakog njegovog podstabla `pred(v)`, pri čemu se taj doseg formalno definira kao
`pred(v)` = `min`(`num(v)`, `num(u1)`, `num(u2)`, ..., `num(uk)`),
gdje su `u1`, `u2`, ..., `uk` vrhovi na koje se potomci od `v` iz istog podstabla vraćaju putem povratnih bridova.

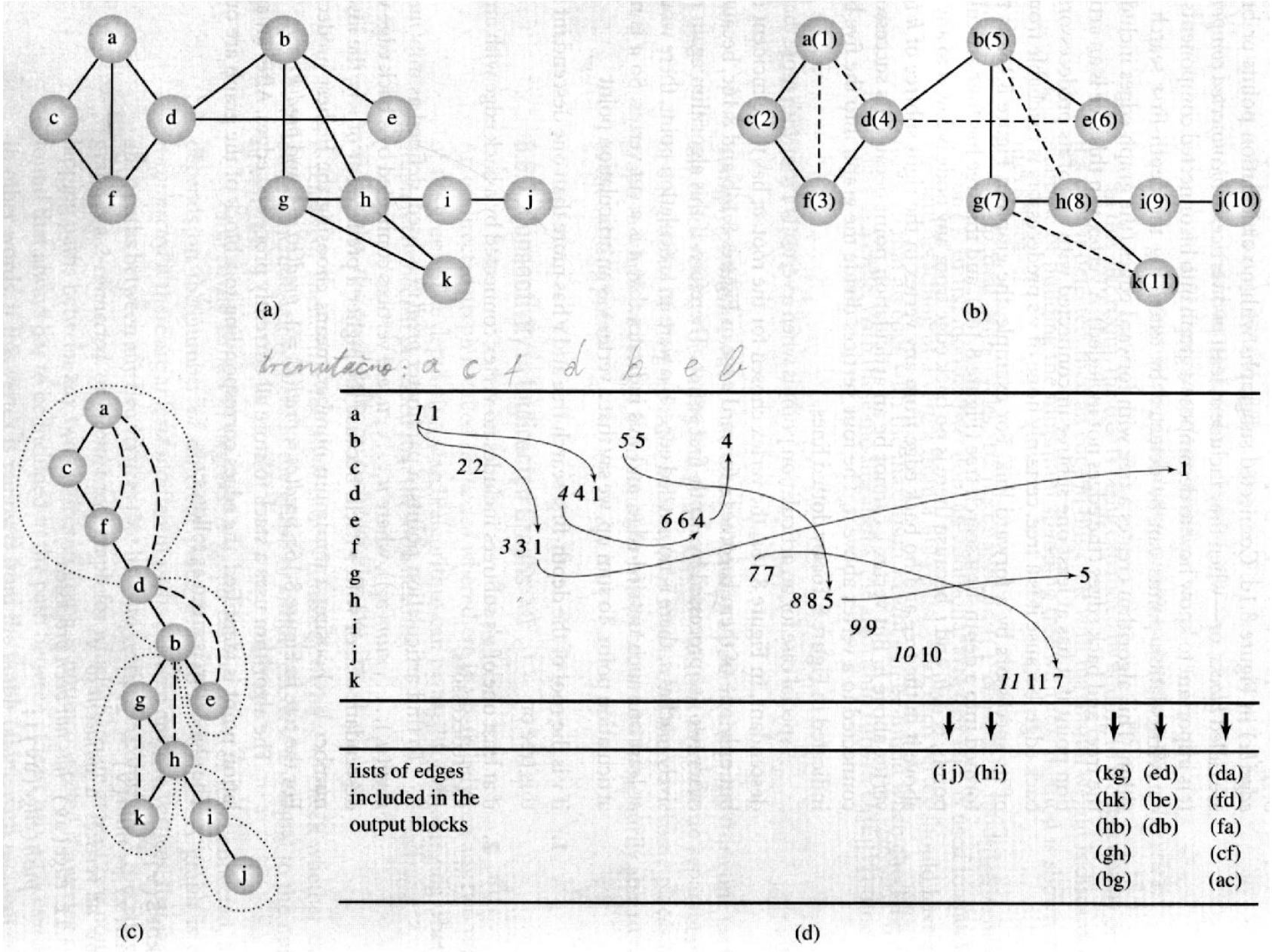
Ekvivalentna definicija je `pred(v)` =
`min`(`num(v)`, `pred(w1)`, `pred(w2)`, ..., `pred(wk)`),
gdje su `w1`, `w2`, ..., `wk` potomci od `v` iz istog podstabla.

Prema tome, ako se nakon DFS obrade, s pridjeljivanjem `pred(v)`, podstabla vrha `v` koje započinje djetetom `u` ustanovi da je `pred(u)` ≥ `num(v)`, znači da se iz cijelog podstabla nije moglo vratiti nigdje ispred `v`, a to znači da je `v` prijelomna točka. Na toj se činjenici temelji BlockDFS algoritam.

```
BlockSearch()
initialization: all vertices num(v)=0;
korak = 0;
while there is a vertex v such that num(v)==0
        BlockDFS(v);
```

```
BlockDFS (v)
pred(v) = num(v) = ++korak;
for all vertices u adjacent to v
        if edge(v,u) is not on the stack //bilo kakav popis
                push(edge(v,u));
        if num(u)==0                    //još uvijek poniremo ...
                BlockDFS(u);
                if pred(u) >= num(v)    //prijelomna točka
                        pop all the egdes until        //svi do edge(v,u)
                                edge(v,u) is popped off; //čine blok
                else //iz podstabla možemo u prethodnika; osvježiti pred(v)
                        pred(v) = min(pred(u), pred(v));
        else if u is not the parent of v
                pred(v) = min(pred(v),num(u));
```

Primjer: pronalaženje prijelomnih točaka i blokova

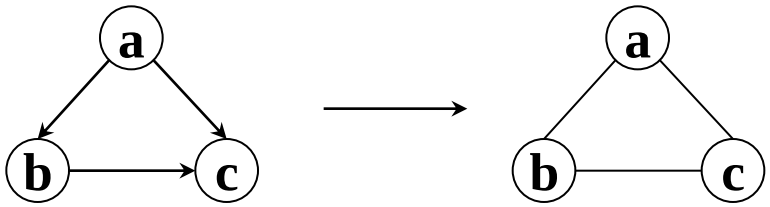


Usmjereni grafovi

Slabo (*weakly*) **povezani** graf je graf kojemu je njegov neusmjereni ekvivalent, dakle graf s istim vrhovima i bridovima, samo neusmjerenima, povezan.

Čvrsto (*strongly*) **povezani** graf je graf kojemu su svaka dva vrha povezana (postoji put između njih) u oba smjera.

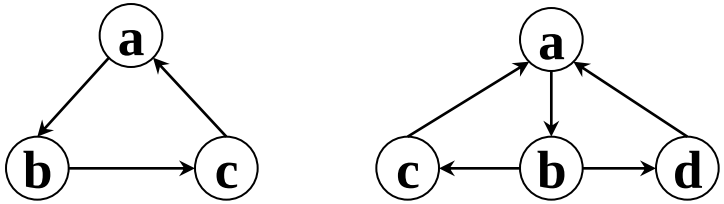
Slaba povezanost



a→b, ali ne i b→a

svi↔svi

Čvrsta povezanost



kružna lista

svi↔svi

Usmjereni grafovi se sastoje od (i mogu rastaviti na) čvrsto povezanih sastavnica (*Strongly Connected Components* - SCC). Ako se obavi takvo rastavljanje i nađe više od jedne SCC, znamo da graf nije u cijelosti čvrsto povezan.

Rastavljanje usmjerenog grafa na SCC slično je rastavljanju neusmjerenog grafa na blokove, ali ima i zamjetnih razlika.

Dok se neusmjerenim grafovima traže prijelomne točke, usmjerenim grafovima traže se korijeni SCC sastavnica.

Budući da u SCC postoje staze između svaka dva vrha u oba smjera, najdalji doseg unatrag u cijelom SCC bloku uvijek je točno do polaznog vrha (iz kojeg se krenulo u DFS rekurziju). Za taj polazni vrh (korijen) sigurno vrijedi `pred(v)=num(v)` i to je obilježje koje služi za detekciju korijena SCC sastavnice.

Problem koji treba riješiti je nalaženje korijena svih SCC sastavnica. Parametar `pred(v)` treba biti doseg unatrag, ali samo do vrhova unutar iste SCC sastavnice.

- doseg do vrha izvan trenutačne SCC sastavnice ne znači ništa jer se iz takvog vrha više ne bismo mogli vratiti u trenutačni vrh pa on sigurno ne može biti korijen SCC sastavnice koja bi sadržavala vrh u koji bismo otišli

Iz ovog razmatranja slijedi da bismo prilikom traženja korijena SCC sastavnice već morali znati koji vrhovi pripadaju pojedinoj SCC, što je, prividno, nerješivo!

Izlaz iz tog zatvorenog kruga je uporaba stoga za privremeni smještaj vrhova tijekom DFS rekurzije, tj. popisivanje vrhova tijekom izgradnje pojedine SCC sastavnice.

Na vrhu stoga uvijek će biti vrhovi SCC sastavnice koja se trenutačno analizira (iako još ne znamo koji su sve vrhovi u toj SCC) pa koji god vrh v grafa trenutačno promatrali kao potencijalni korijen SCC, sigurno je da su iznad njega na stogu samo vrhovi koji bi morali biti dio one SCC kojoj bi v bio korijen (dakle, one SCC koja se trenutačno "gradi").

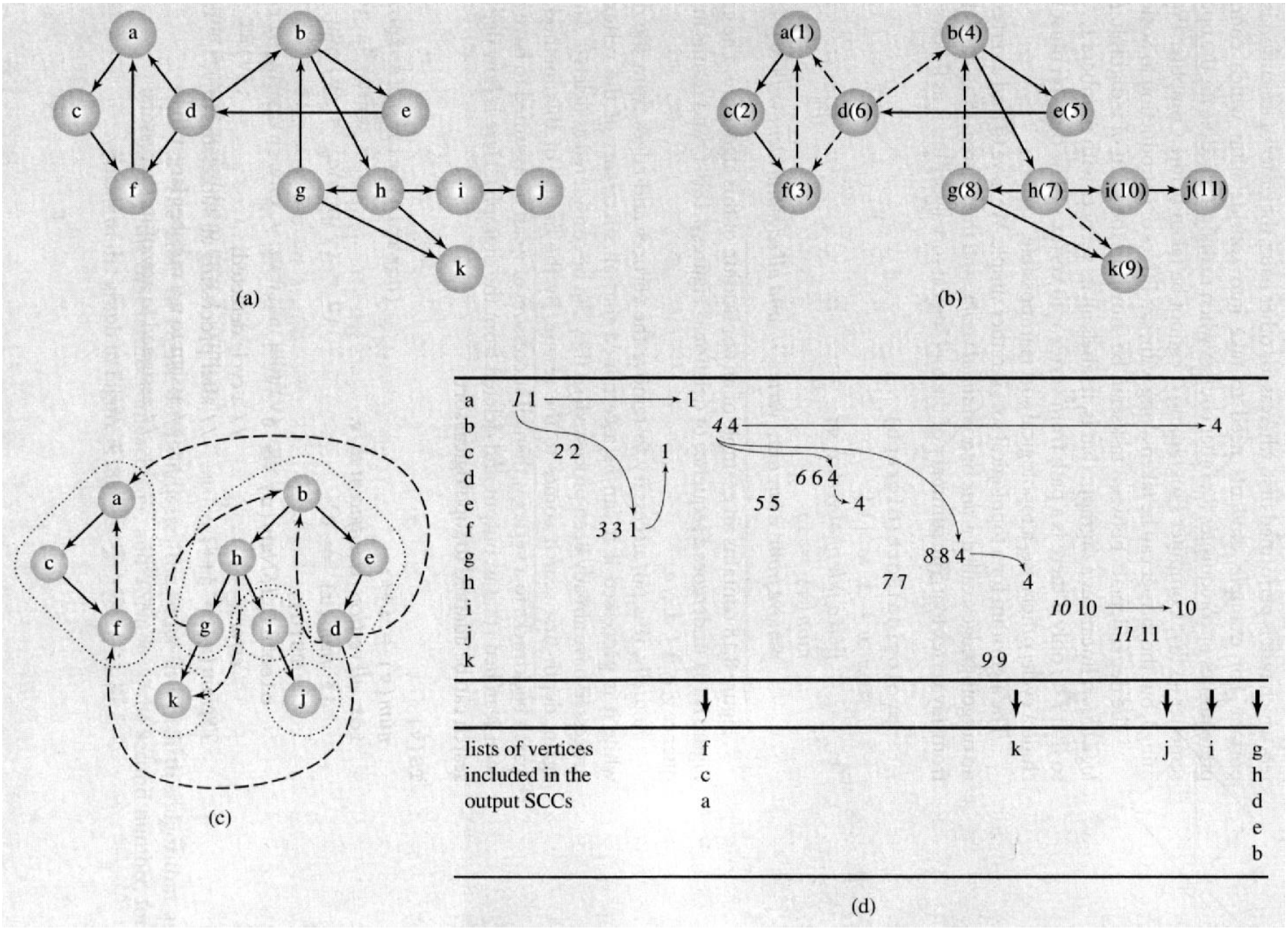
Napomena:
U `else if` naredbi (u pseudo-kodu algoritma na sljedećem slajdu) uvjet *'and* `u` *is on the stack'* spriječava "skok" u drugo već obrađeno stablo ili drugu SCC sastavnicu jer ako vrh `u` nije na stogu, znači da je već uklonjen s njega prilikom pronalaska korijena neke druge SCC sastavnice.

Tarjanov algoritam

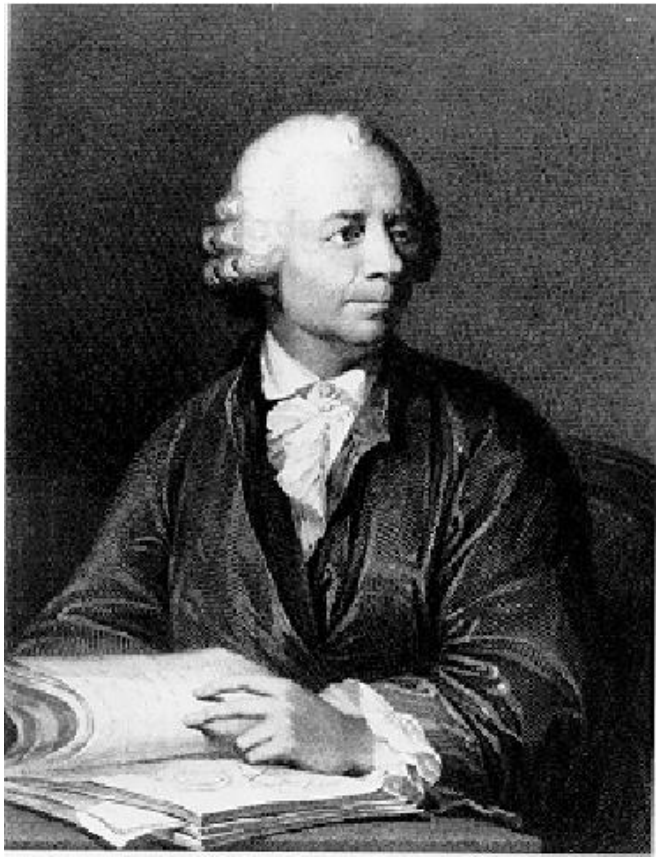
```
SCCSearch()
initialization: all vertices num(v)=0;
korak = 0;
while there is a vertex v such that num(v)==0
      SCCDFS(v);

SCCDFS (v)
pred(v) = num(v) = ++korak;
push(v);      //svaki vrh na koji naiđemo ide na stog
for all vertices u adjacent to v
      if num(u)==0                //još uvijek poniremo ...
            SCCDFS(u);
            pred(v) = min(pred(u), pred(v));
      else if num(u)<num(v) and u is on the stack
            pred(v) = min(pred(v),num(u));
if  pred(v) == num(v)      //'v' je korijen SCC
   pop all vertices off the stack until v is popped off;      //ispis SCC
```

Primjer: pronalaženje SCC sastavnica usmjerenog grafa



Eulerovi grafovi (*Eulerian Graphs*)

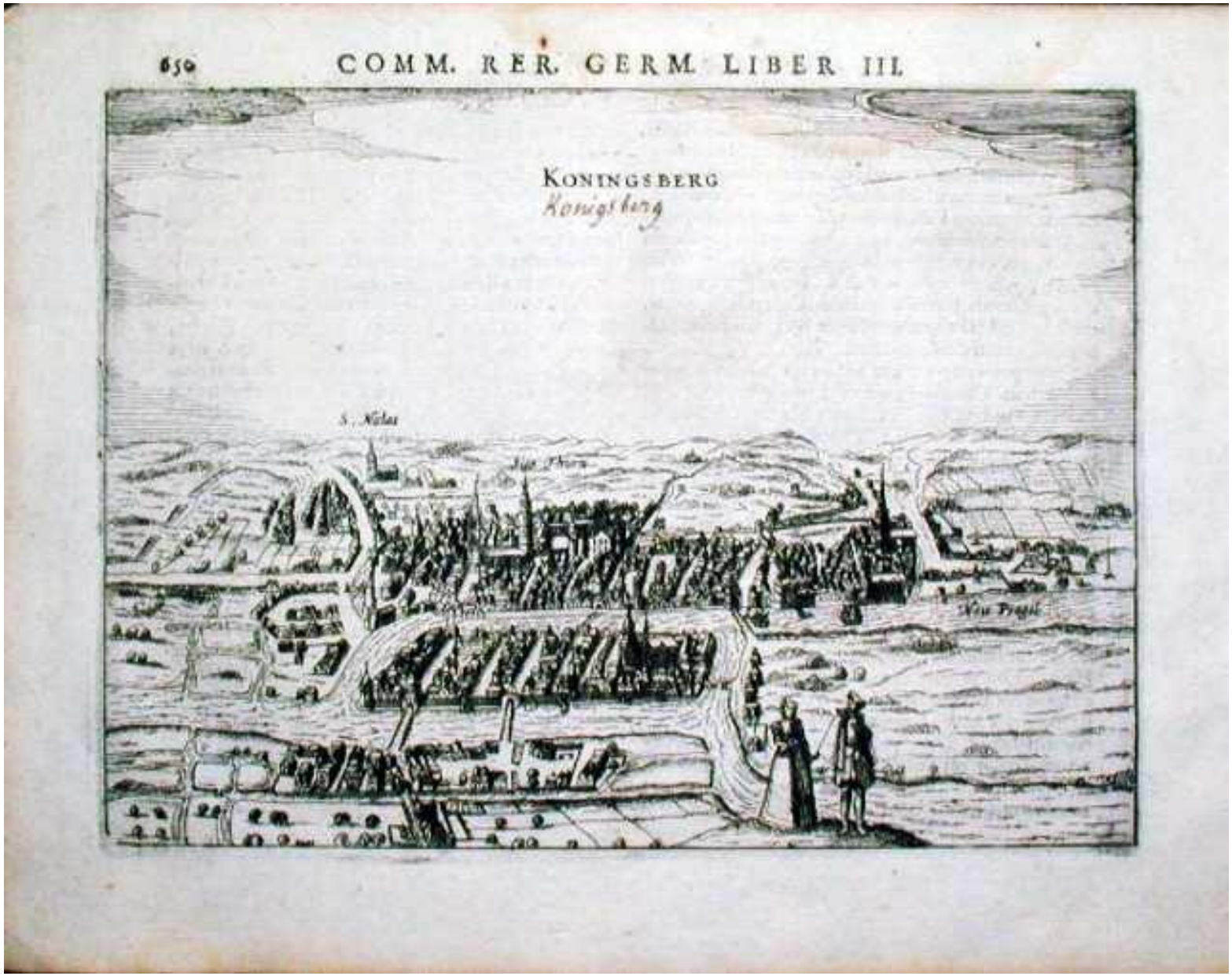


Leonhard Paul Euler

Basel, 15. travanj 1707. - St. Petersburg, 18. rujan 1783.

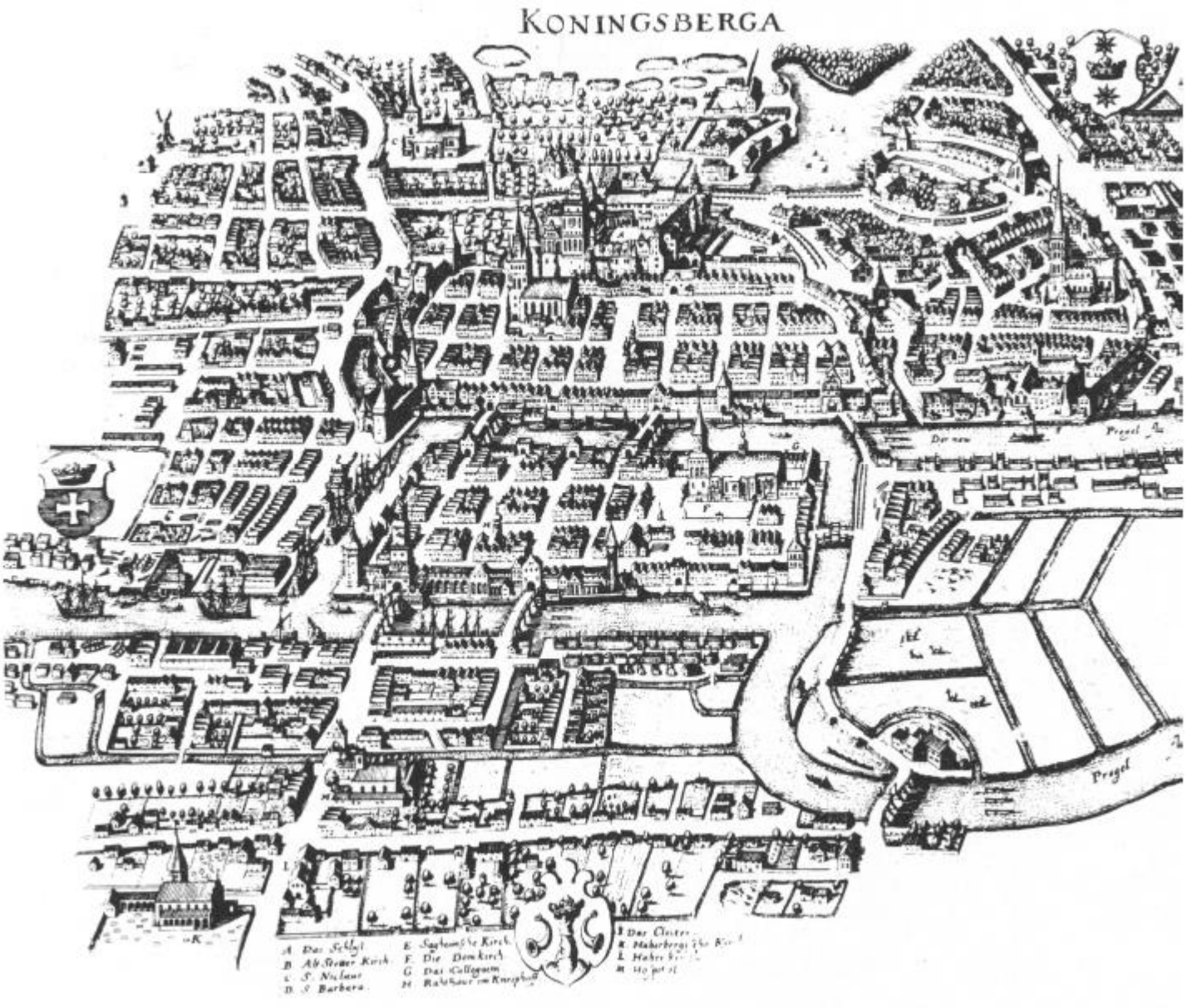

Microsoft MapPoint
Gulf of Finland
Stockholm
Tallinn
Kiviõli
Slantsy
Säffle
Örebro
Katrineholm
Vänern
SWEDEN
Nyköping
Haapsalu
Rapla
Jõgeva
Lake Peipus
ESTONIA
RUSSIA
Vänersborg
Motala
Linköping
Tartu
VÄSTRA GÖTALAND
Skövde
Pärnu
Põlva
Kuressaare
Võru
Pskov
Göteborg
Jönköping
Rūjiena
Valka
PSKOV
Västervik
Visby
Limbaži
Valmiera
Lavry
Eksjö
Vetlanda
Gislaved
Varberg
KALMAR
Ventspils
Balvi
Novorzhev
Falkenberg
Värnamo
Cēsis
Växjö
Tukums
Rīga
Ērgļi
Opochka
Halmstad
Kalmar
LATVIA
Saldus
Jelgava
Rēzekne
Laholm
Liepāja
Naujoji Akmenė
Jēkabpils
Preiļi
SKÅNE
Helsingborg
Skuodas
Mažeikiai
Biržai
Karlshamn
Karlskrona
Daugavpils
Krāslava
Copenhagen
Telšiai
Kretinga
Šiauliai
Panevėžys
Druya
Malmö
Simris
Baltic Sea
Klaipėda
Utena
Vidzy
Polatsk
Šilalė
LITHUANIA
Molėtai
Rønne
Tauragė
Pastavy
Šilutė
Ukmergė
Sovetsk
Kaunas
Jonava
Kaliningrad
Prienai
Vilnius
Stralsund
Wejherowo
Marijampolė
Barysaw
Gdańsk
KALININGRAD
Greifswald
Słupsk
Bartoszyce
Maladzyechna
Kołobrzeg
Elbląg
Ketrzyn
Suwałki
Varėna
Anklam
Koszalin
Białogard
POMORSKIE
Morąg
Minsk
Neubrandenburg
Chojnice
POLAND
Olsztyn
Ełk
Grodno
Lida
Prenzlau
Szczecin
Grudziądz
Ostróda
Pisz
Mir
Osipovichi
Złotów
Iława
Kolno
Sokółka
BELARUS
Schwedt
Stargard Szczeciński
Piła
Brodnica
Działdowo
Białystok
Vawkavysk
Baranavičy
Neuruppin
Bydgoszcz
Toruń
Ostrołęka
Inowrocław
PODLASKIE
Salihorsk
Berlin
Gorzów Wielkopolski
MAZOWIECKIE
Hajnówka
Byaroza
Potsdam
Frankfurt
Poznań
Włocławek
Płock
Warsaw
Luninyets
Konin
Kobryn
Pinsk
GERMANY
Zielona Góra
Warta
Kutno
Siedlce
Brest
Cottbus
WIELKOPOLSKIE
Głogów
Kalisz
Łódź
Skierniewice
Łuków
Torgau
Hoyerswerda
Kozienice
LUBELSKIE
RIVNE
Lubin
Puławy
SAXONY
Tomaszów Mazowiecki
Legnica
ŁÓDZKIE
Radom
Lublin
Chełm
UKRAINE
Görlitz
Końskie
Dresden
Wrocław
Ostrowiec Świętokrzyski
VOLYN
Děčín
Częstochowa
Zamość
Luts'k
Wałbrzych
Česká Lípa
Nowa Ruda
Kielce
©2008 Microsoft Corp.
16°
18°
20°
22°
24°
26°
12°
54°
56°
58°
30°
52°

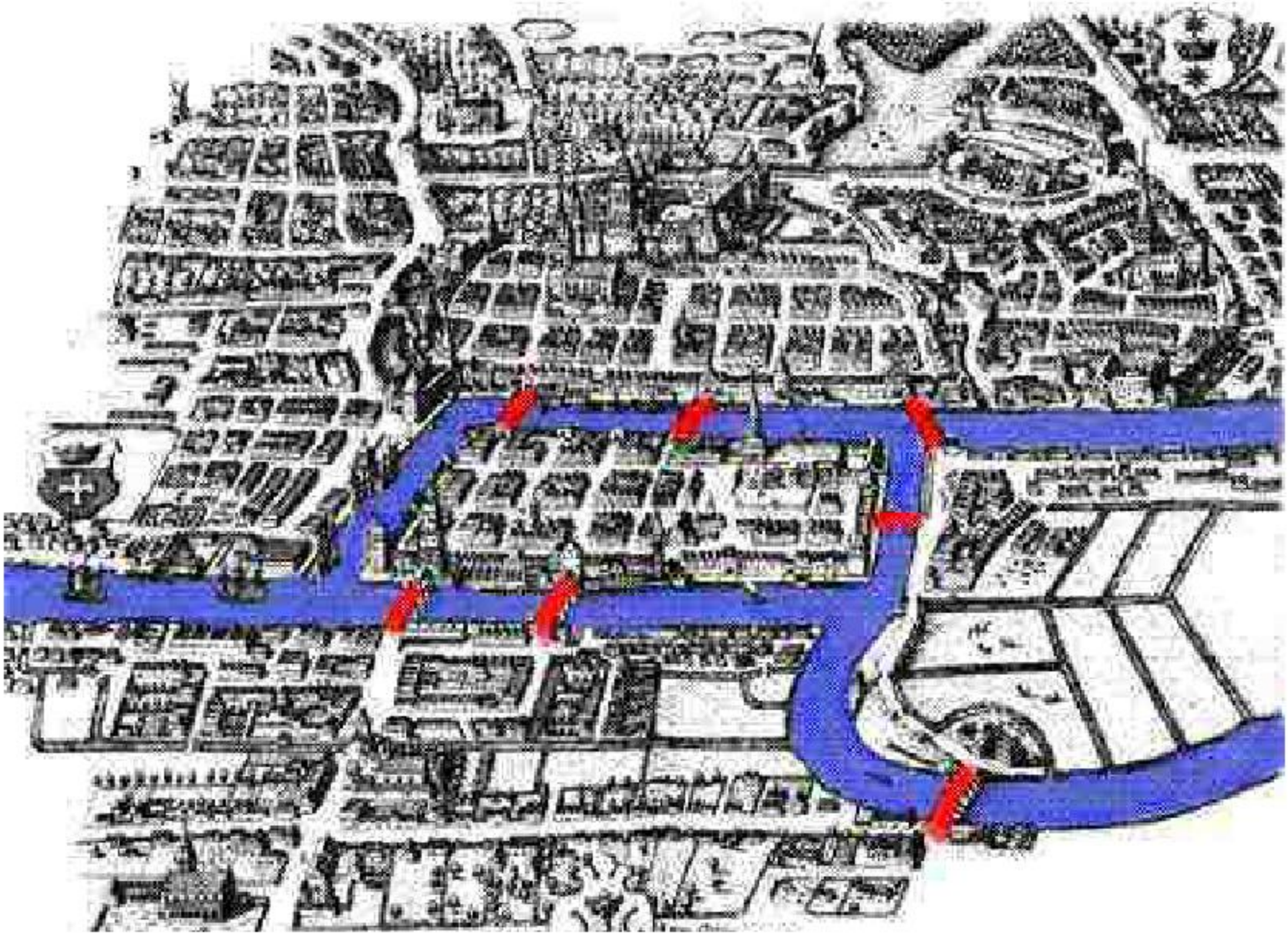
Koningsberg 1457. godine

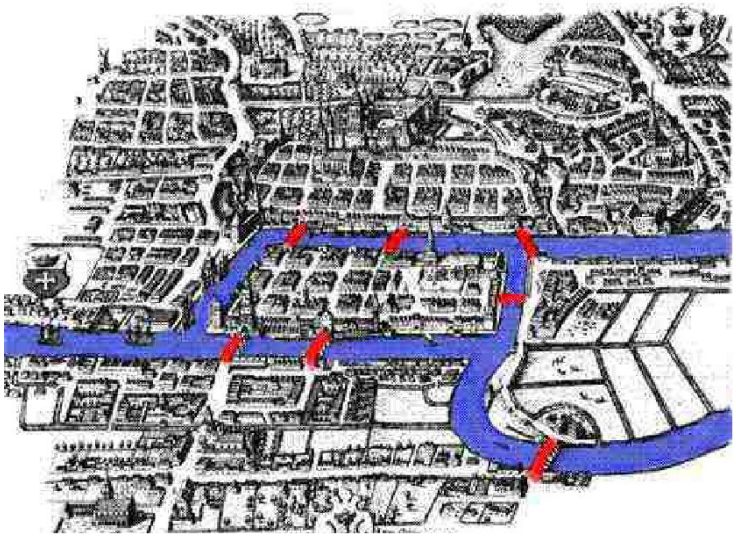


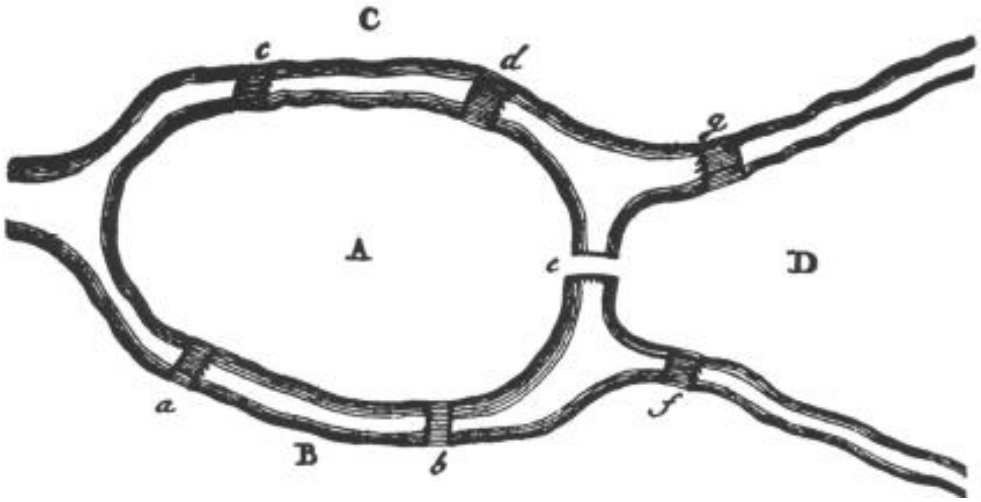
Koningsberg u Eulerovo vrijeme ...

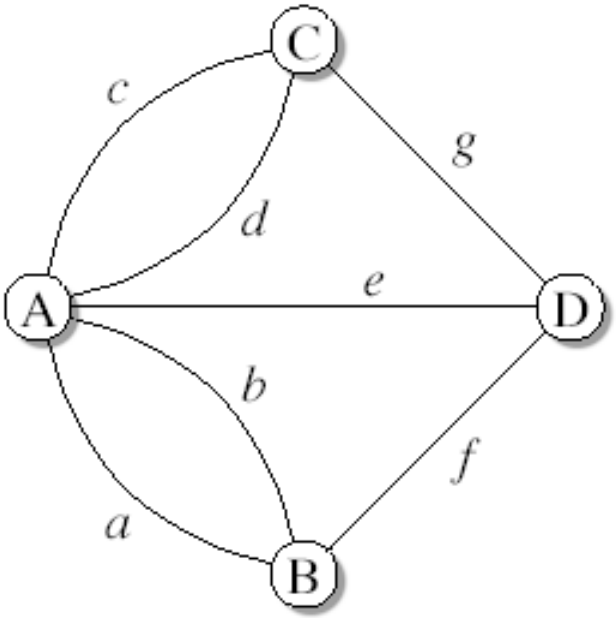
Koningsberg u Eulerovo vrijeme ...











Lijevo - skica sedam mostova u Eulerovom rukopisu

Desno - model

Pitanje: može li se proći svim mostovima i vratiti na polazno mjesto, a da se svakim mostom prođe samo jednom?

Ne, takav obilazak nije moguć!

Euler L.: Solutio problematis ad geometriam situs pertinentis,

Commentarii Academiae Scientiarum Imperialis

Petropolitanae 8, 128-140 (1736)

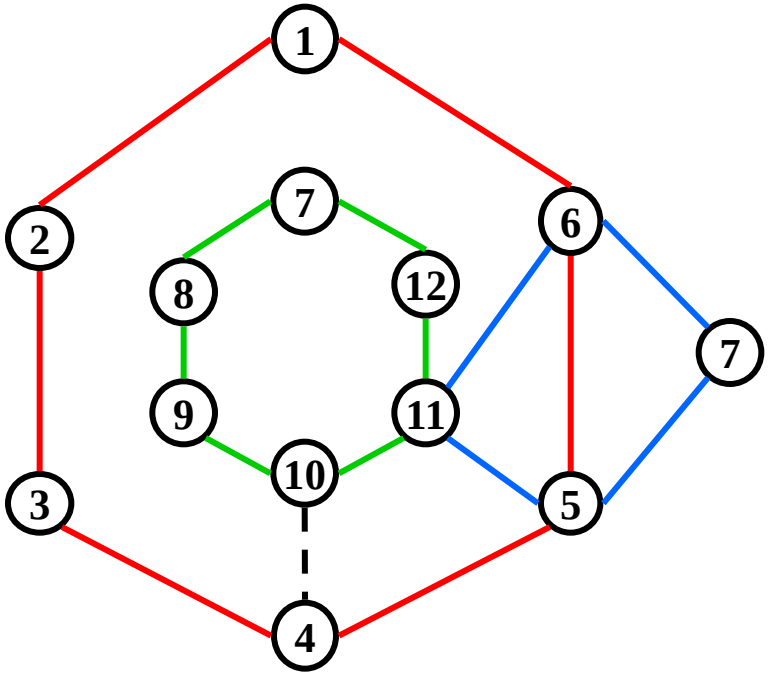
- članak objavljen u zborniku Ruske Carske Akademije znanosti u St. Petersburgu, a proizlazi iz predavanja koje je Euler u toj ustanovi održao 26. kolovoza 1735.
- taj se rad smatra začetkom teorije grafova

Eulerov graf je povezani graf u kojem postoji Eulerov ciklus.
Eulerov ciklus je krug (*circuit*) koji je ujedno i Eulerova putanja.
Eulerova putanja (*Euler Trail*) je putanja koji prolazi svim bridovima grafa.

Eulerov teorem (1736. god.):
Povezani neusmjereni graf ima Eulerov ciklus ako i samo ako su svi njegovi vrhovi parnog stupnja.
Povezani neusmjereni graf ima Eulerovu putanju ako i samo ako su točno dva njegova vrha neparnog, a svi ostali parnog stupnja. □

Dokaz: **a)** Nužnost uvjeta: Eulerov graf $\Rightarrow$ parni stupnjevi
Nužnost parnosti stupnjeva vrhova slijedi iz sljedećeg razmatranja. U svaki vrh na putanji dolazi se jednim bridom te jednim i izlazi (nastavlja) iz njega. Dakle, ako se vrh obilazi jednom, on mora imati dva prileže ća brida. Ako se neki vrh obilazi više puta, za svaki obilazak moramo imati jedan ulazni i jedan izlazni brid, odnosno ukupni broj bridova = broj obilazaka $\cdot$ 2 = paran broj.

Dokaz: **b)** Dovoljnost uvjeta: parni stupnjevi $\Rightarrow$ Eulerov graf
Zamislimo graf koji zadovoljava pretpostavke (slika) i nađimo jedan ciklus u njemu. Neka to bude npr. 1-2-3-4-5-6-1 koji ćemo označiti s C_0. Ako je to Eulerov ciklus (obuhvaća sve bridove), dokazali smo tvrdnju. Ako nije, uklonimo iz grafa sve bridove iz C_0. U ostatku grafa mora biti još bridova jer C_0 nije obuhvatio sve. Štoviše, barem dva preostala brida moraju biti spojena s vrhovima kojima je prolazio ciklus C_0 (na slici su to vrhovi 5 i 6) jer kad ne bi bilo takvih, graf ne bi bio povezan. K tome, po pretpostavci su svi vrhovi parnog stupnja pa ako smo



zajedničkom vrhu uklonili dva brida, morao ih je ostati paran broj, odnosno najmanje dva, a iz polazne pretpostavke slijedi da i ostatak grafa ima vrhove parnog stupnja. Zbog toga se može pronaći drugi ciklus C_1 koji počinje i završava u zajedničkom vrhu, a prolazi samo bridovima kojima C_0 nije prošao. Taj se postupak može ponavljati *n* puta, sve dok ne uklonimo sve bridove iz grafa. No, tada unija $C_0 \cup C_1 \cup C_2 \cup \ldots \cup C_n$ čini Eulerov ciklus što dokazuje tvrdnju. ■

Dokaz uvjeta za postojanje Eulerove putanje slijedi iz prethodnog dokaza, a ilustrira ga iscrtkani brid 4-10. Kad bi on postojao, vrhovi 4 i 10 bili bi jedini vrhovi neparnog stupnja pa bi se moglo krenuti iz jednoga od njih, obići sve bridove i završiti u drugome (npr. 4-5-6-7-5-11-6-1-2-3-4-10-11-12-7-8-9-10).

Napomena

Ekvivalent Eulerovog teorema za usmjerene grafove je:
Usmjereni graf ima Eulerov ciklus ako i samo ako je čvrsto povezan i za sve njegove vrhove vrijedi indeg(v)=outdeg(v).

Eulerov ciklus mogao bi se naći rekurzivno (*backtracking*), ali takvo rješenje bi bilo složenosti O(|E|!).

Bolji način pronalaženja Eulerovih ciklusa slijedi izravno iz dokaza Eulerovog teorema:

1. uzimati brid po brid do zatvaranja ciklusa
2. pronaći vrh koji još ima bridova i krenuvši iz njega ponoviti korak 1; ponavljati korake 1 i 2 sve dok ima bridova

```
EulerCycleSlow(v)                    //'v' je polazni vrh
while (1)
      while u grafu postoji neki brid(v,u)          //O(|E|)
            ciklus = ciklus + brid(v,u);
            ukloniti brid(v,u) iz grafa;
            v = u;
      if postoji vrh z koji ima prilezeci brid      //O(|V|)
            v = z;
      else
            break;
```

Složenost: O(|E|) + O(|V|). Nije loše, ali može još brže!

Sve se temelji na sljedećem zanimljivom svojstvu E-grafova ...

Lema: Bilo koja putanja u Eulerovom grafu do vrha iz kojeg više nema odlaznih bridova završava u polaznom vrhu. □

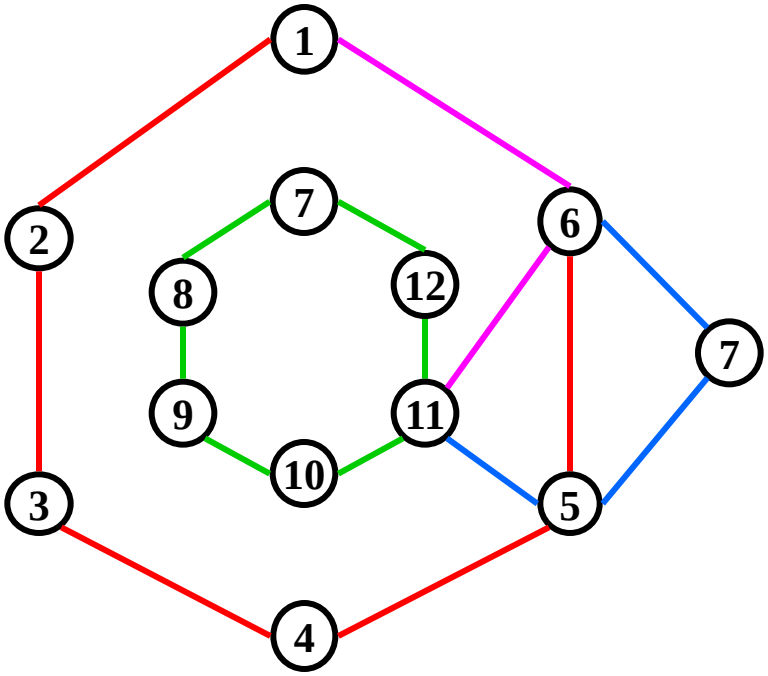
Dokaz:
Budući da se bridovima može proći samo jednom, obilazak mora biti konačan jer se iz koraka u korak broj preostalih bridova smanjuje. Nadalje, stati možemo samo u vrhu koji je do našeg dolaska imao samo jedan brid (odlazni), a znamo da su svi vrhovi koje smo obišli izgubili paran broj bridova pa im je paran broj i preostao ako ih još uopće i imaju (jer graf je po pretpostavci Eulerov). Jedina iznimka je vrh iz kojeg smo krenuli. Pri polasku je izgubio jedan brid, a prilikom možebitnih naknadnih obilazaka izgubio je paran broj bridova, dakle ukupno neparan broj. Prema tome, jedino njemu može preostati samo jedan brid pa jedino u njemu i možemo stati bez mogućnosti za nastavak obilaska. ■

Fleuryev algoritam - najstariji i najbrži; slijedi iz prethodne leme

Charles Rohault de Fleury: Deux problemes de geometrie de situation,
Journal de mathematiques elementaires 1883, 257-261.

1. Ako se traži Eulerov ciklus, krenuti iz bilo kojeg vrha. Ako se traži Eulerova putanja, krenuti iz jednog od dva vrha neparnog stupnja. Neka je polazni vrh v_1.
2. Recimo da smo do sada prošli vrhovima $v_1, v_2, ..., v_k$, odnosno bridovima $e_1, e_2, ..., e_{k-1}$, i neka je $G_k = G\backslash\{v_1v_2, v_2v_3, ..., v_{k-1}v_k\}$, odnosno $G_k = G \setminus \{e_1, e_2, ..., e_{k-1}\}$. Sljedeći korak ovisi o prileže ćim bridovima vrha v_k:
   a) Ako je v_k izolirani vrh u G_k, tj. ako nema prileže ćih bridova, postupak je završen, a skup $\{e_1, e_2, ..., e_{k-1}\}$ je Eulerov ciklus (to slijedi iz prethodne leme).
   b) Ako još ima prileže ćih bridova vrha v_k, kao sljedeći u ciklusu izabrati neki koji nije most.
   c) Ako su svi prileže ći bridovi vrha v_k mostovi, izabrati bilo koji.

Prema prethodnoj lemi, jasno je da je po završetku Fleuryevog algoritma skup $\{e_1, e_2, ..., e_{k-1}\}$ stvarno Eulerov ciklus, a formalni dokaz (koji je vrlo jednostavan) može se naći u svakoj ozbiljnijoj knjizi o teoriji grafova.



Primjer:
Krenemo iz 1 i slijedimo crvenu stazu do 6. Iz 6 možemo u 1, 7 i 11, ali brid 6-1 je most pa njega treba obići posljednjeg. Zato biramo npr. brid 6-7 i nastavljamo plavom stazom do 11. Iz 11 možemo u 6, 10 i 12, ali sada je brid 11-6 most pa biramo 11-12 i nastavljamo zelenom stazom opet do 11. Sada u 11 više nemamo izbora pa obilazimo most 11-6, a isto tako u 6 nemamo izbora i nastavljamo bridom 6-1, čime zatvaramo Eulerov ciklus.

Budući da se u svakom koraku Fleuryevog algoritma broj preostalih bridova smanjuje za jedan, složenost je $O(|E|)$.

Fleuryev algoritam - pseudo kod

```
Fleury (graph)
v = arbitrary starting vertex;
path = v;
ToBeCh = all edges in graph;  //Bridovi koje još treba obići.
while there is an edge(v,u) in ToBeCh;  //brid koji počinje u 'v'
      if edge(v,u) is the only edge originating in v
            choose edge(v,u) as the next one;        //most
      else
            choose any edge(v,u), which is not a bridge
                                          as the next one;
      path = path + edge(v,u);
      v = u;
      remove choosen edge(v,u) form ToBeCh;
if there are no more edges in ToBeCh
      success;
else
      failure;
```

Je li neki brid most ili ne može se ustanoviti na razne načine.
Najjednostavnije je primijeniti neku od metoda za detekciju ciklusa.
Na primjer, obaviti BSF ili DFS krečući od polaznog vrha tog brida.
Ako tijekom BFS ili DFS opet stignemo u polazni vrh, brid nije most i obratno.
Može se i rastaviti graf na SCC sastavnice ili ...

Usmjereni grafovi

Osnovni uvjet za postojanje Eulerovog ciklusa jest čvrsta povezanost grafa (*strong connectedness*).
(vidi teoriju grafova...)

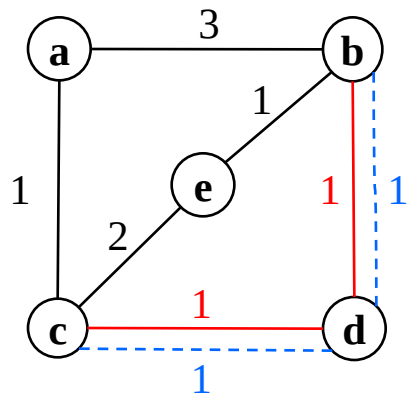
Traženje Eulerovog ciklusa obavlja se istim algoritmima kao i za neusmjerene grafove, uzimajući u obzir usmjerenost bridova.

Problem: naći putanju (ciklus) koja najkraćim putem obilazi sve bridove.

- ako je graf Eulerov, bilo koji od prethodnih algoritama
- svaki Eulerov ciklus ili putanja u istom grafu su jednako dugi
- ako graf nije Eulerov, treba ga “eulerizirati”

Eulerizacija grafa: postupak dodavanja bridova vrhovima neparnog stupnja u neEulerovom grafu da bi on postao Eulerov.

- moguće samo u povezanim grafovima jer oni imaju paran broj vrhova neparnog stupnja (rezultat iz teorije grafova, ali je jasno samo po sebi ...)



Primjer: dva vrha neparnog stupnja

Vrhovi b i c su neparnog stupnja. Najkraći put uzmeđu njih je b-d-c duljine 2. Dodajemo bridove b-d i c-d duljine 1+1=2.

Sada su svi vrhovi parnog stupnja pa je graf euleriziran.

Eulerov ciklus: a-b-d-c-e-b-d-c-a.

Ako graf ima više od dva vrha neparnog stupnja, radi se o kineskom problemu poštara (*The Chinese Postman Problem* (CPP)).

The Chinese Postman Problem: obići sve ulice u nekom naselju i vratiti se na polazno mjesto najkraćim mogućim putem.

- problem se može modelirati grafom;
  ako je graf Eulerov, Eulerov ciklus je traženo rješenje
- ako graf nije Eulerov, treba ga eulerizirati, ali tako da dodani bridovi (ponovni obilasci istih ulica) budu najkraći mogući
  - vrhove neparnog stupnja razvrstati u parove tako da ukupni zbroj duljina novih bridova među njima bude najmanji mogući → *Matching (Assignment) Problem*
  - riješeno 1960. god.

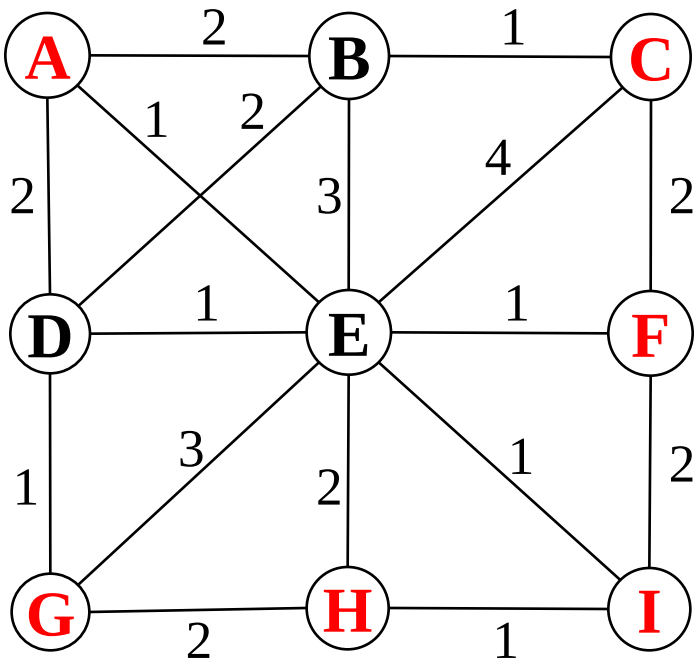
  (Mei-ko Kwan: Graphic Programming Using Odd or Even Points, Chinese Mathematics 1 (1962), pp. 273-277;
  izvorno objavljeno u
  Acta Mathematica Sinica 10 (1960), pp. 263-266)
- nakon eulerizacije, Fleuryev algoritam

Algoritam rješavanja CPP:

`ChinesePostman (G = (V,E))`

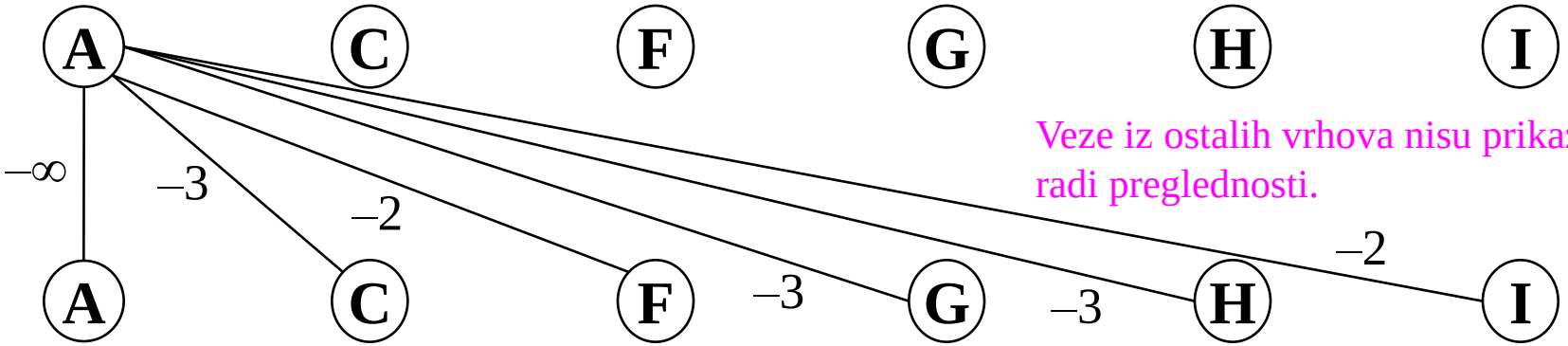
1) ODD = *svi vrhovi neparnog stupnja iz* G;
2) `OddPaths` = *najkraći putevi svi-svi u* ODD; //npr. WFI(G)
3) *od vrhova u* ODD *konstruirati potpuni bipartitni graf* H=(X∪Y,E')
   (*vrhovi iz ODD su na obje strane, tj.* X = V(ODD) i Y=V(ODD))
   tako da: a) *težine bridova weight*(v_i,v_i) = $w_{ii} = -\infty$;
   b) *težine bridova weight*(v_i,v_j) = $-w_{ij}$; $i \neq j$
4) *u dobivenom grafu* H *naći optimalnu pridruženost* M
   (*riješiti* "*optimal assignment" problem*);
5) `for` *svaki brid* (v_i,v_j) *iz* M *za koji je* `deg`(v_i) *još uvijek neparan*
   proširiti graf svim bridovima koji čine put `OddPaths`(v_i,v_j),
   tj. $E = E \cup (v_r,v_s) \in$ `OddPaths`(v_i,v_j);
6) *Fleuryevim algoritmom odrediti Eulerov ciklus*;

▼ Primjer CPP:



Najkraći putevi u ODD:

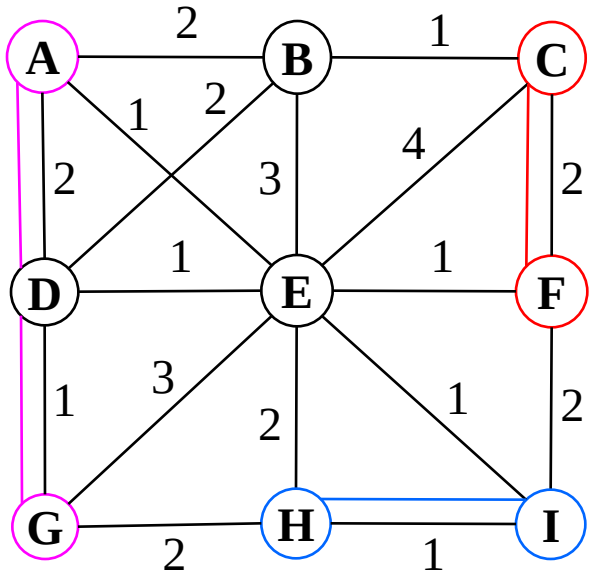
	A	C	F	G	H	I
A	0	3	2	3	3	2
C		0	2	4	5	4
F			0	3	3	2
G				0	2	3
H					0	1
I						0



Veze iz ostalih vrhova nisu prikazane radi preglednosti.

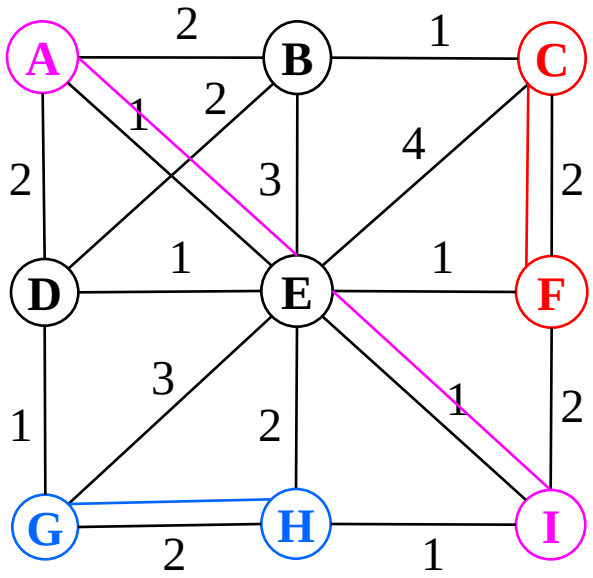
Eulerizirani graf
(1. mogućnost):

$w_{AG} + w_{CF} + w_{HI}$
$= -3 + (-2) + (-1)$
$= -6$



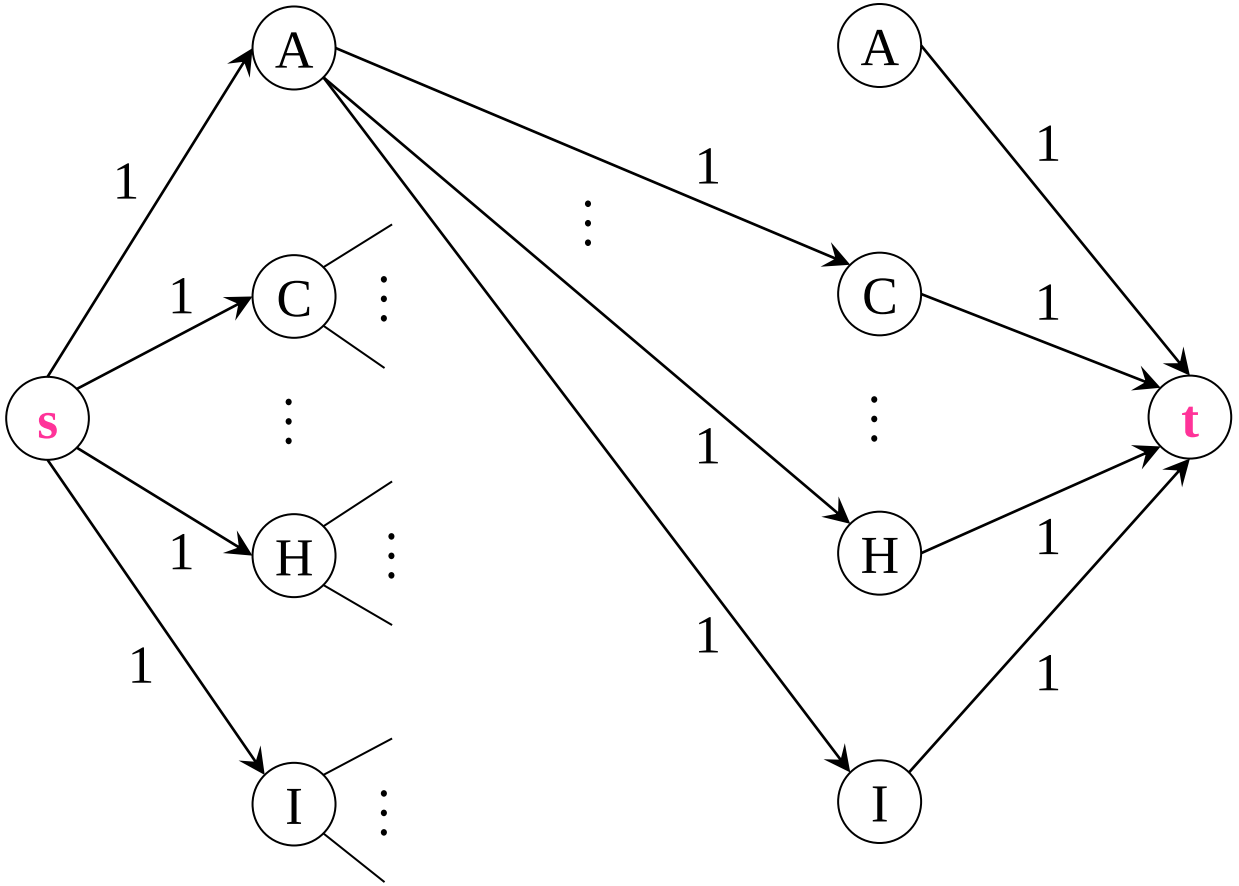
Eulerizirani graf
(2. mogućnost):

$w_{AI} + w_{CF} + w_{GH}$
$= -2 + (-2) + (-2)$
$= -6$



II način: konstruirati ekvivalentnu mrežu i odrediti najveći mogući protok u njoj

– svi bridovi usmjereni i kapaciteta jedan



Hamiltonovi grafovi (*Hamiltonian Graphs*)

- nazvani u čast irskom astronomu Williamu Rowanu Hamiltonu koji je 1857. god., radi zabave svoje djece, izmislio igru "Icosian Game" (još se naziva i *Hamiltonian's puzzle*) u kojoj je zadatak pronaći ono što danas nazivamo Hamiltonov ciklus

Hamiltonov graf je graf koji ima (sadrži) Hamiltonov ciklus.

Hamiltonov ciklus je staza (*path*) koja prolazi svim vrhovima grafa, ali samo po jednom, i završava u polaznom vrhu.

- klasa NP-complete problema
- opći uvjet koji graf treba zadovoljiti da bi bio Hamiltonov, tj. kriterij prepoznavanja sličan Eulerovom teoremu kad je riječ o Eulerovim ciklusima, nije poznat, a nije poznato ni postoji li uopće takav uvjet
- problem se jednostavno rješava *backtrackingom*, ali složenost takvog algoritma je O(|V|!)
- postoje brojna rješenja za posebne slučajeve; jedno relativno jednostavno se temelji na Bondy-Chvátalovom teoremu

Teorem (Bondy-Chvátal):

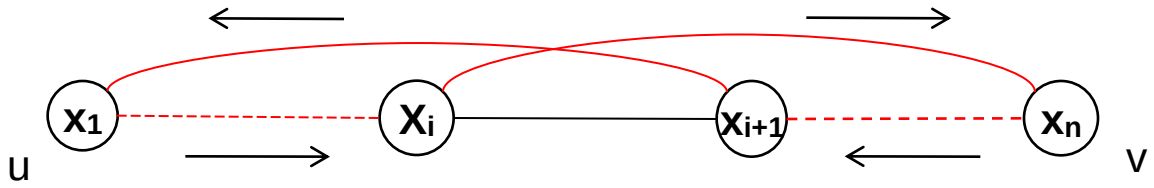
Neka je G = (V,E) jednostavni (*simple*) neusmjereni graf u kojem je $|V| = n \geq 3$ i neka su u i v nesusjedni vrhovi u njemu za koje vrijedi $\deg(u) + \deg(v) \geq n$. Tada je G' = G + (u,v) Hamiltonov graf ako i samo ako je G Hamiltonov graf. □

Dokaz: Nužnost uvjeta (ako je G Hamiltonov, onda je i G') je očita pa se dokazuje samo dovoljnost (ako je G' Hamiltonov, onda je i G).

Pretpostavimo da je G' Hamiltonov, a G nije. Znači, G + (u,v) ima Hamiltonov ciklus koji sigurno zahvaća brid (u,v). Iz toga slijedi da u samome G postoji Hamiltonova staza $u = x_1, x_2, ..., x_n = v$ koja prolazi svim vrhovima grafa G. Sada treba dokazati da u G ne postoji samo Hamiltonova staza, nego i Hamiltonov ciklus.

Neka je S skup svih indeksa, umanjenih za jedan, vrhova koji su susjedi polaznog vrha u; simbolički $S = \{j: (u,x_{j+1}) \in E\}$ i ima $\deg(u)$ elemenata. Nadalje, neka je T skup svih indeksa vrhova koji su susjedi završnog vrha v; simbolički $T = \{j: (x_j,v) \in E\}$ i ima $\deg(v)$ elemenata. Za uniju S i T sigurno vrijedi da je podskup skupa $\{1, 2, ..., n-1\}$, a kako je po pretpostavci $\deg(u) + \deg(v) \geq n$, zaključujemo da skupovi S i T moraju imati barem jedan zajednički element.

Isti indeks u S i T znači da među susjedima u i v postoje susjedni vrhovi, a tada se Hamiltonova staza u G sigurno može zatvoriti u Hamiltonov ciklus putem "ukriženih" bridova (*crossover edges*). ■



"Iz fore" ... ☺ Dokaz kontradikcijom.

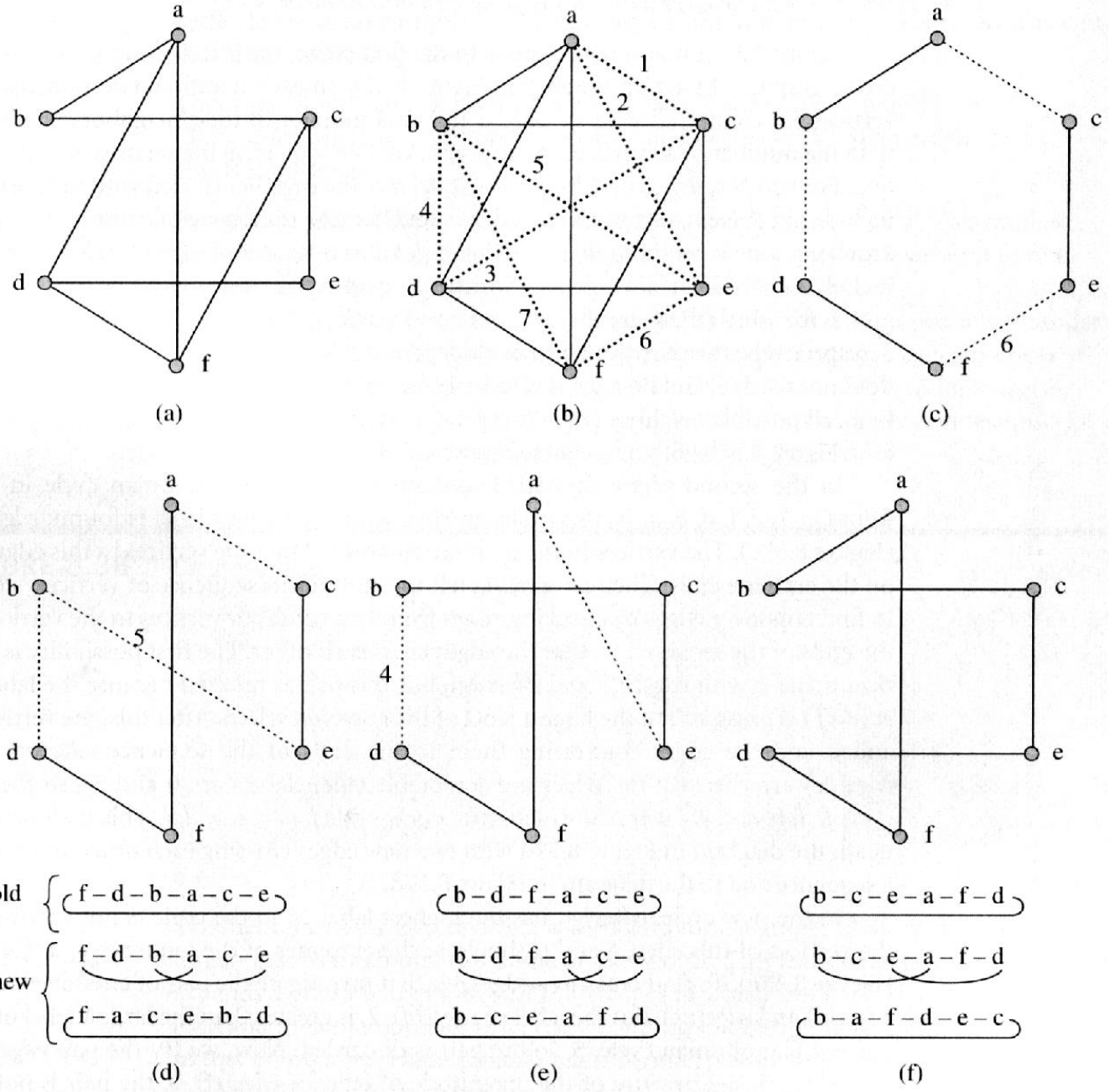
Pretpostavimo da je G' Hamiltonov, a G nije. Znači, G+(u,v) ima Hamiltonov ciklus koji sigurno zahvaća brid (u,v). Iz toga slijedi da u G postoji Hamiltonova staza $x_1 = u, x_2, ..., x_n = v$ koja prolazi svim vrhovima grafa G. No tada, ako je x_i susjed vrha x_1, pri čemu je $2 \leq i \leq n$, vrh x_{i-1} ne smije biti susjed vrha x_n jer bi inače postojao Hamiltonov ciklus $(x_1, x_i, ... , x_n, x_{i-1}, ... , x_1)$, a po pretpostavci ga nema. To vrijedi za sve susjede vrha x_1, a njih ima $\deg(u = x_1)$ pa za vrh x_n vrijedi $\deg(v = x_n) \leq (n-1) - \deg(u = x_1)$, odnosno $\deg(v) + \deg(u) \leq (n-1) < n$, a to je protivno pretpostavci. ■

Iz Bondy-Chvátalovog teorema slijedi da se iz nekih Hamiltonovih grafova mogu dobiti drugi (rjeđi) Hamiltonovi grafovi uklanjanjem određenih bridova.
Taj je zaključak osnova algoritma za nalaženje H-ciklusa (algoritam objavljen 1985. god.):

1. polazni graf, za koji uopće ni ne znamo ima li ili ne Hamiltonov ciklus, proširiti novim bridovima; pri tome je važno bilježiti redni broj novododanih bridova
   - dok ima nesusjednih vrhova za koje je $\deg(u) + \deg(v) \geq n$, dodavati brid (u,v); (tj. povezivati ih tako da postanu susjedi)
   - time se graf samo "obogaćuje", ali teorem kazuje da ako graf nije bio Hamiltonov, neće ni postati, a ako je bio, ostat će i dalje
2. u tako proširenom grafu pronaći bilo kakav Hamiltonov ciklus ako on uopće postoji; to trebalo biti relativno jednostavno
3. iz polaznog H-ciklusa uklanjati novododane bridove nadomještajući ih dvama "ukriženim" bridovima (izvorni ili manjeg rednog broja); krenuti od kasnije dodanih bridova prema ranije dodanima (veći ka manjim rednim brojevima); sve dok ih ima

Primjer:

pronalaženje Hamiltonovog ciklusa na temelju Bondy-Chvátalovog teorema



Hamiltonov ciklus na temelju Bondy-Chvátalovog teorema: pseudokod

```
HamiltonianCycle(graph)
initialisation: label all edges in graph with num(edge(vi,vj))=0;
newedges = 0;
newG = graph;
while newG contains nonadjacent vertices u and v
                                such that deg(u)+deg(v)≥|V|
      num(edge(u,v)) = ++newedges;
      newG = newG ∪ {edge(u,v)};
if there is a Hamiltonian cycle C in newG
      while (k = maximal label num()
                                among all the edges in C) > 0
            C = new cycle, due to a crossover with edges labeled
                  by label < k;
```

Pitanje je kako detektirati postojanje, a potom i pronaći polazni Hamiltonov ciklus.
To je znanost – dobro došli, pridružite se! ☺
Srećom, to treba napraviti samo jednom i to kada je graf još uvijek gust ...

Problem trgovačkog putnika (*Travelling Salesman Problem*) – rješenje dvostruko duže od najkraćeg mogućeg

- zapravo, problem pronalaženja najkraćeg H-ciklusa
- većina stvarnih TSP spada u klasu *metričkih problema*
- **metrički problem** je problem za koji vrijedi nejednakost trokuta

$$dist(u,v) \leq dist(u,k) + dist(k,v)$$

- za klasu metričkih problema postoji vrlo dobro približenje (aproksimativno rješenje) koje u najgorem slučaju daje ciklus 1,5 puta dulji od najkraćeg (N. Christofides, Worst-case analysis of a new heuristic for the traveling salesman problem, Report 388, Graduate School of Industrial Administration, Carnegie Mellon University, 1976)

Postoji slično, a jednostavno, rješenje koje uvijek možemo primijeniti za TSP problem i koje u najgorem slučaju daje ciklus dvostruko dulji od najkraćeg razapinjajućeg stabla, a u najboljem slučaju ciklus približno jednako dug kao MST. To je posve dobro rješenje jer je duljina MST donja granica duljine H-ciklusa, tj. vrijedi (vidi Cormen...)

$$\text{duljina(najkraći H-ciklus)} \geq \text{duljina(MST)} .$$

Najviše 2·MST rješenje za metrički TSP

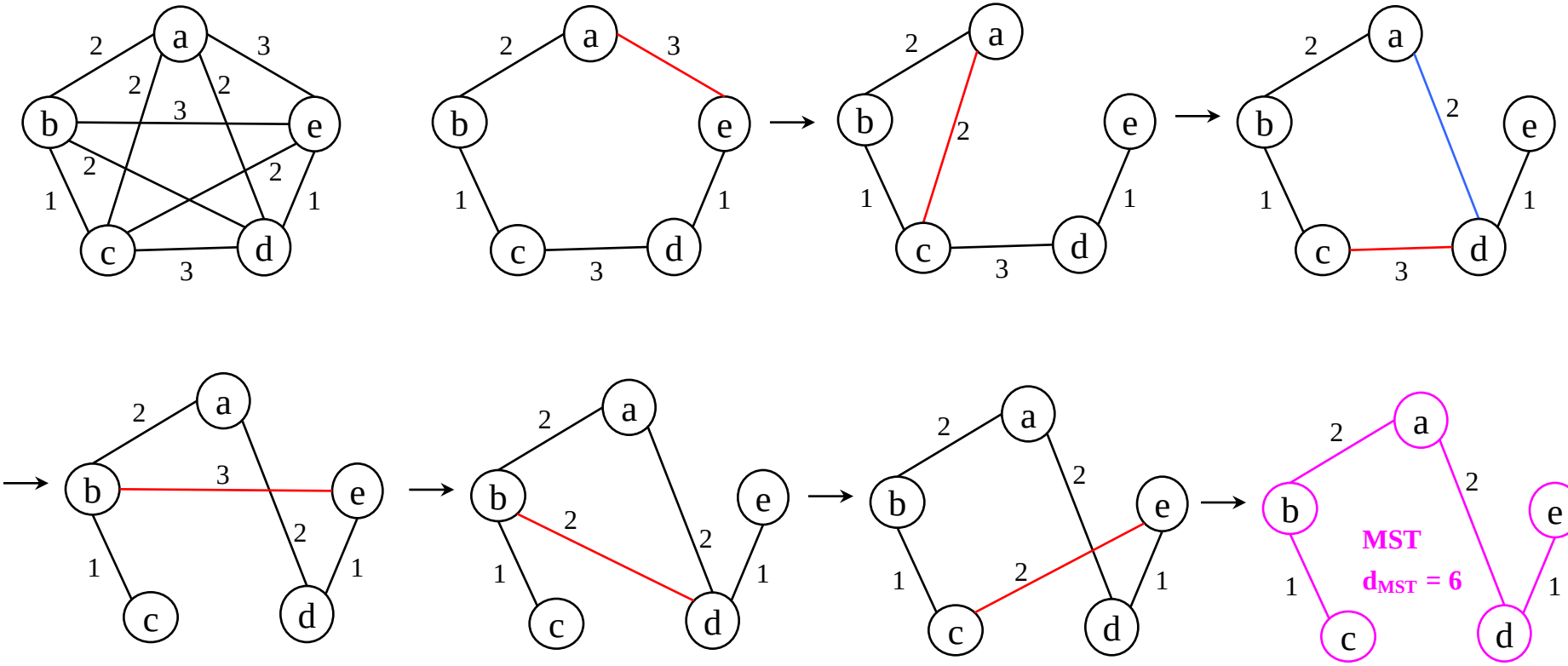
- pretpostavka: potpuni (veze svi-svi) nesumjereni metrički graf

1. konstruirati najkraće razapinjajuće stablo (MST)
2. iz tog MST napraviti Eulerov graf udvostručavanjem svih bridova MST
3. pronaći Eulerov ciklus u proširenom grafu
   - očito, taj je ciklus dvostruko duži od MST
4. pretvoriti E-ciklus u Hamiltonov tako da se obilazi Eulerov, a pri svakom nailasku na već obiđeni vrh on se jednostavno preskoči i traži se prvi neobiđeni koji tada postaje sljedeći u Hamiltonovom ciklusu
   - DFS razapinjajućeg stabla, a preskakanje tijekom povratka iz rekurzije
   - *preorder* obilazak MST i, na kraju, spajanje posljednjeg vrha s polaznim

Iz nejednakosti trokuta slijedi da svako preskakanje vrha u Eulerovom ciklusu zapravo skraćuje obilazak pa slijedi da konačni Hamiltonov ciklus ne može biti dulji od 2·(duljina MST).

Primjer: polazni vrh je A

Traženje MST; Dijkstrin algoritam



Pretvorba u Eulerov pa u Hamiltonov ciklus