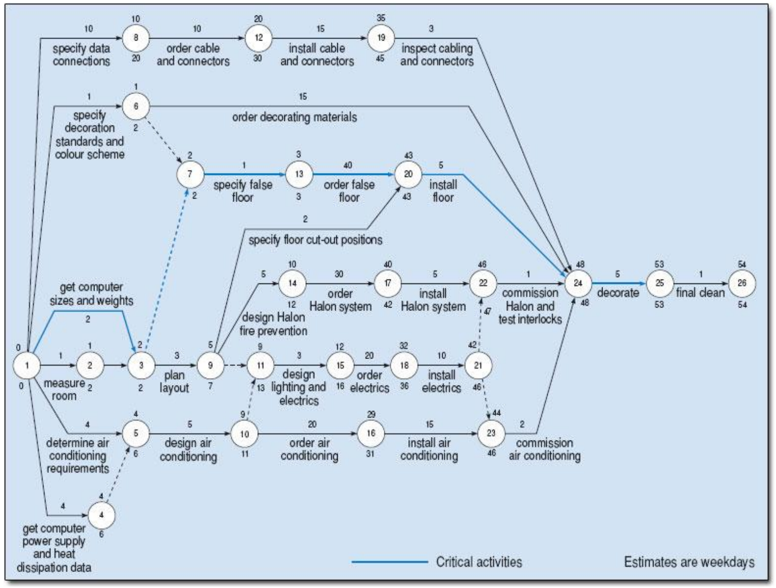
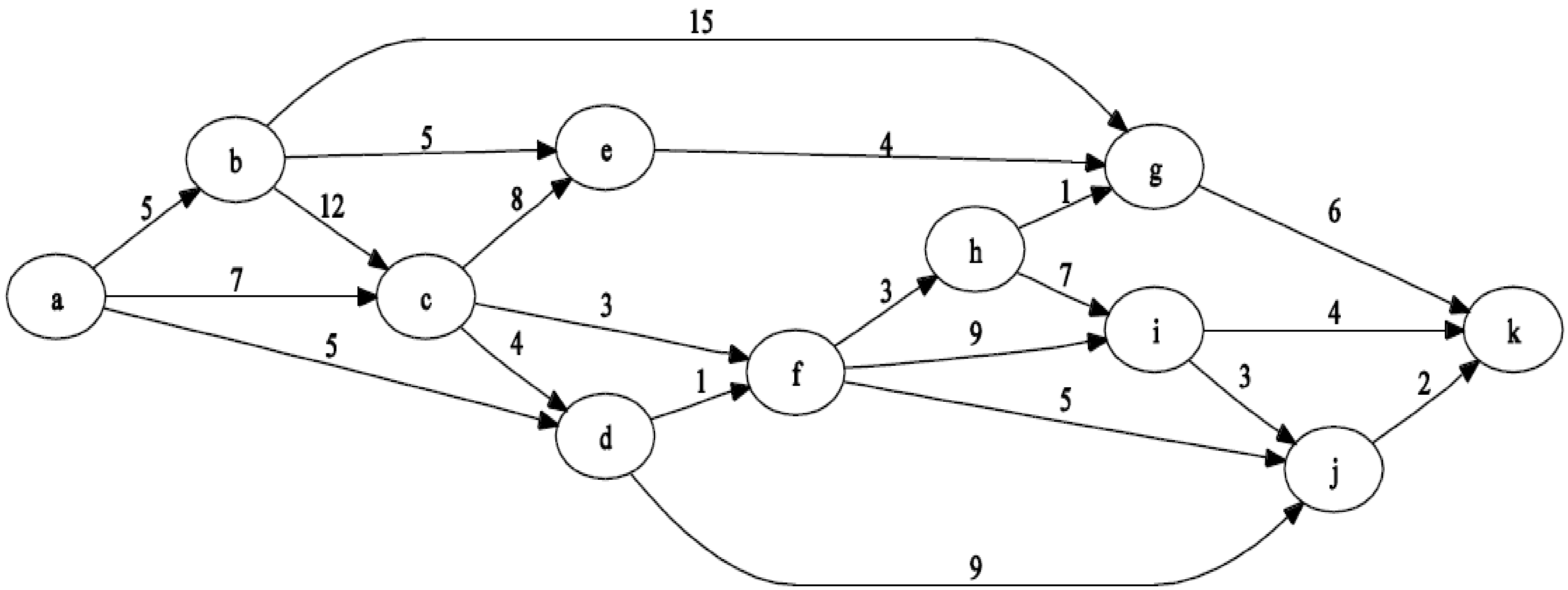


Analiza kritične poti

Mrežni diagram poteka



- projekt: delno urejen seznam akcij
    - vsaka akcija ima določen čas za izvršitev
- predstavimo z usmerjenim acikličnim grafom
    - vozlišča so časovni mejniki
    - povezave so akcije s časom izvajanja
- kritična pot je najdaljša pot od začetka do konca projekta

Čas med dvema mejnikoma

- naj bo eval(x,y) čas povezave <*x,y*>

- čas med poljubnima vozliščema *a* in *c*

$$t(a,a) = 0$$

$$t(a,c) = \max_{<a,b>\in E} (eval(a,b) + t(b,c))$$

- iz rekurzivne definicije dobimo preprost, a neučinkovit eksponenten algoritem

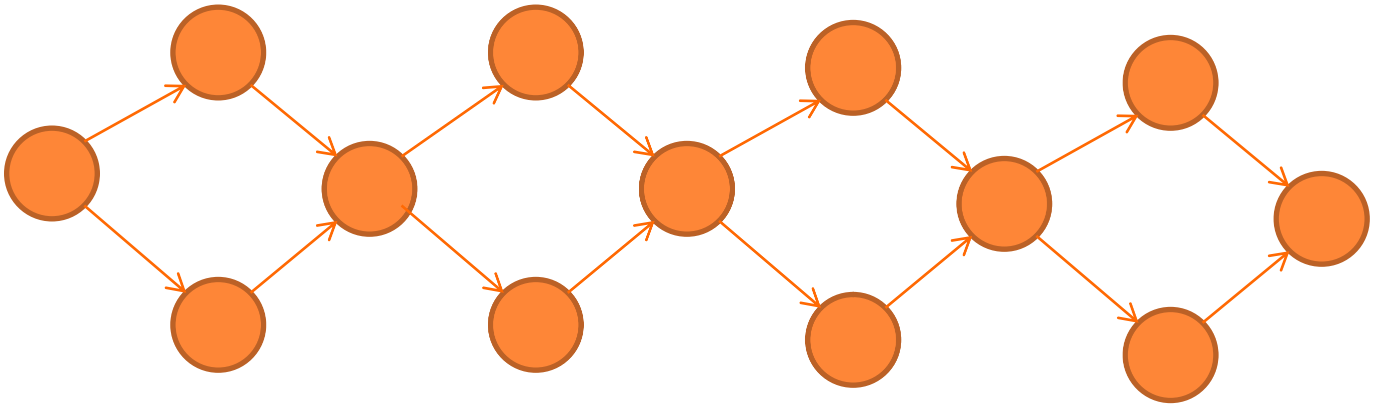
$$t(a,a) = 0$$

$$t(a,c) = \max_{<a,b>\in E} (eval(a,b) + t(b,c))$$

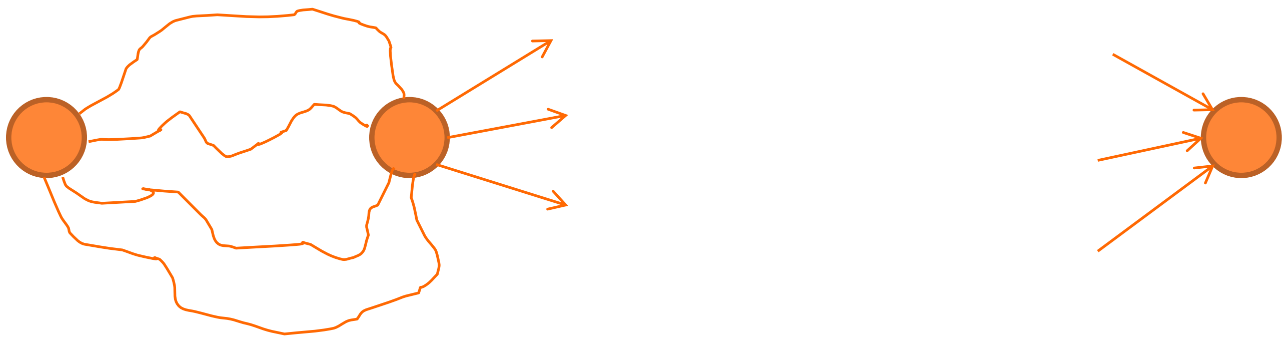
```
public double tExp(Vertex a, Vertex c, DiGraph g) {
   if (a == c) return 0 ;
   else {
         Vertex b ;
         double max = 0, temp;
         Edge e = g.firstEdge(a) ;
         while (e != null) {
              b = g.endPoint(e) ;
              temp = ((Double)e.evalue).doubleValue() + tExp(b,c, g);
                                                                    // rekurzija
              if (temp > max)
                     max = temp ;
              e = g.nextEdge(a, e) ;
         } // while
         return max ;
      } // else
   } // tExp
```

Kritična pot

- deli mnogih poti so enaki in se z rekurzivnim algoritmom ponovno preračunavajo



Kritična pot – dinamično programiranje



- graf pregledujemo od začetka projekta proti koncu
- hranimo seznam vozlišč, za katere smo pregledali že vse poti do njih, nismo pa še pregledali njihovih naslednikov
- za vsako vozlišče hranimo čas maksimalne poti, ki vodi do njega
- zato, da ugotovimo, če smo pregledali vse poti, ki vodijo do vozlišča, potrebujemo vstopno stopnjo vozlišča, ki se med iskanjem zmanjša ob pregledu vsake nove poti
- če želimo izpisati še kritično pot, shranimo še predhodnika na maksimalni poti

Kritična pot – dinamično programiranje

INICIALIZACIJA:
- izračunamo vstopne stopnje vseh vozlišč
- postavimo začetne čase za vsa vozlišča na 0

```
class ValueType {
    String name ;
    int inDegree;
// Vertex parent ; // kazalec na predhodnika
    double time ;
  } // class ValueType
```

Kritična pot – dinamično programiranje

```
public double tDynamic(Vertex a, Vertex c, DiGraph g) {
  // a – zacetno vozlisce, c – zakljucno vozlisce
  Vertex v, w ; // trenutno vozlisce in njegov naslednik
  Edge e ; // povezava <v, w>
  List ls = new ListLinked(); // seznam vozlisc, katerih
                              // naslednikov se nismo pregledali
  Object pos ;
```

```
ls.insert(a);
while (! ls.empty()) {
    pos = ls.first() ; // izberemo lahko poljubno vozlisce
    v = (Vertex)ls.retrieve(pos) ; // izberemo kar prvega
    ls.delete(pos);
    e = g.firstEdge(v);
    while (e != null) {
        w = g.endPoint(e);
        if (((ValueType)w.value).time < ((ValueType)v.value).time +
                                        ((Double)e.evalue).doubleValue())
           ( (ValueType) w.value).time = ((ValueType)v.value).time +
                                        ((Double)e.evalue).doubleValue();
      ((ValueType)w.value).inDegree −− ;//ena pot do w pregledana
       //ce so pregledane vse poti do w,je w kandidat za pregledovanje
        if ( ((ValueType)w.value).inDegree == 0)
            ls.insert(w);
        e = g.nextEdge(v, e) ;
      } // while e != null
   } // while ! ls.empty()
   // koncni cas je cas zakljucnega vozlisca
   return ((ValueType)c.value).time ;
} // tDynamic
```

Kritična pot – dinamično programiranje

- inicializacija (priprava grafa) ima zahtevnost O(n+m)
  - izračunamo vstopne stopnje vseh vozlišč
  - postavimo začetne čase za vsa vozlišča na 0
  - prehod preko vseh vozlišč (n) in vseh povezav (m)

- časovna zahtevnost algoritma za iskanje kritične poti z dinamičnim programiranjem je

$$O(n+m) = O(m)$$

- pregledamo vse povezave (m) in vsa vozlišča (n)
- ker je graf povezan, velja $n - 1 \leq m$

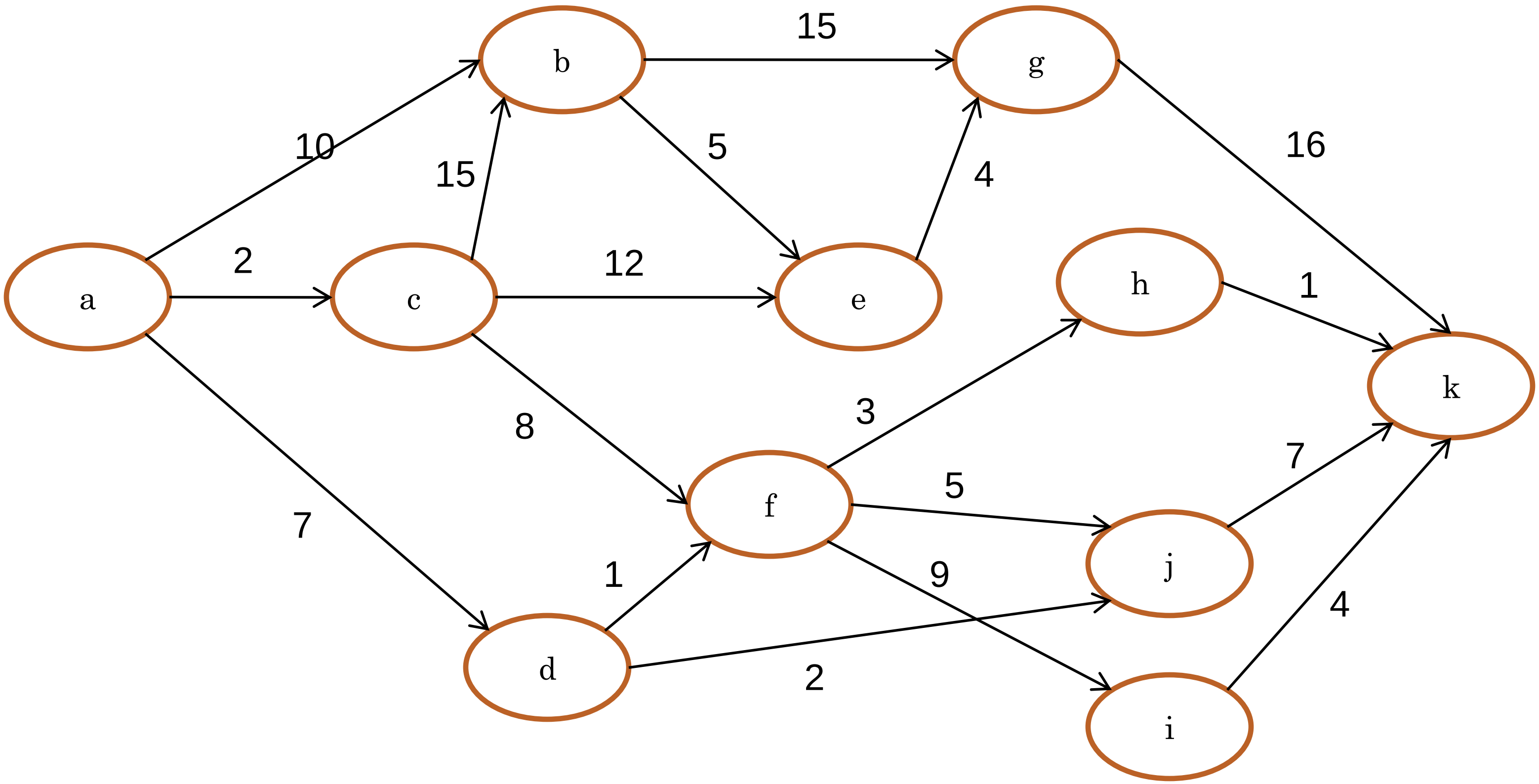
Shranitev in izpis kritične poti

```
if (((ValueType)w.value).time < ((ValueType)v.value).time +
                              ((Double)e.evalue).doubleValue()) {
  ( (ValueType) w.value).time = ((ValueType)v.value).time +
                              ((Double)e.evalue).doubleValue();
  ( (ValueType) w.value).parent = v; // *** dodano ***
}
```

```
// izpis kriticne poti
w = c ;
while (w != a) {
  v = ((ValueType)w.value).parent ;
  e = g.firstEdge(v) ;
  while (g.endPoint(e) != w)
    e = g.nextEdge(v, e) ;
  System.out.println("<" + v + ", " + w + ", " + e + ">") ;
  w = v ;
} // while
```

PRIMER - KRITIČNA POT (1/14)

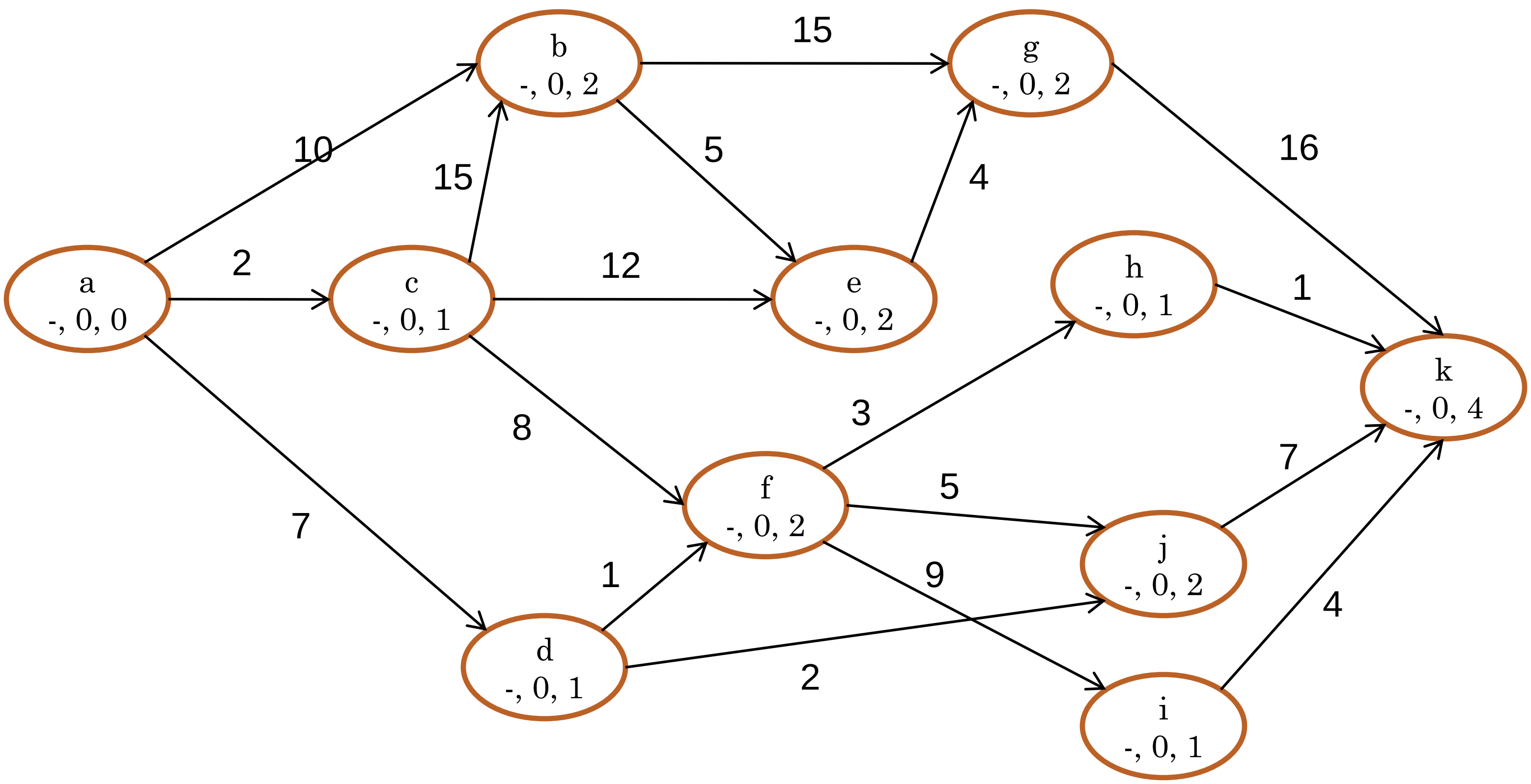
Podan je graf na sliki. Izračunajte kritično pot v grafu.



Primer - kritična pot (2/14)

Inicializacija

v vozliščih shranimo predhodno vozlišče, maksimalen čas do vozlišča, število še ne pregledanih vhodov (vstopna stopnja)

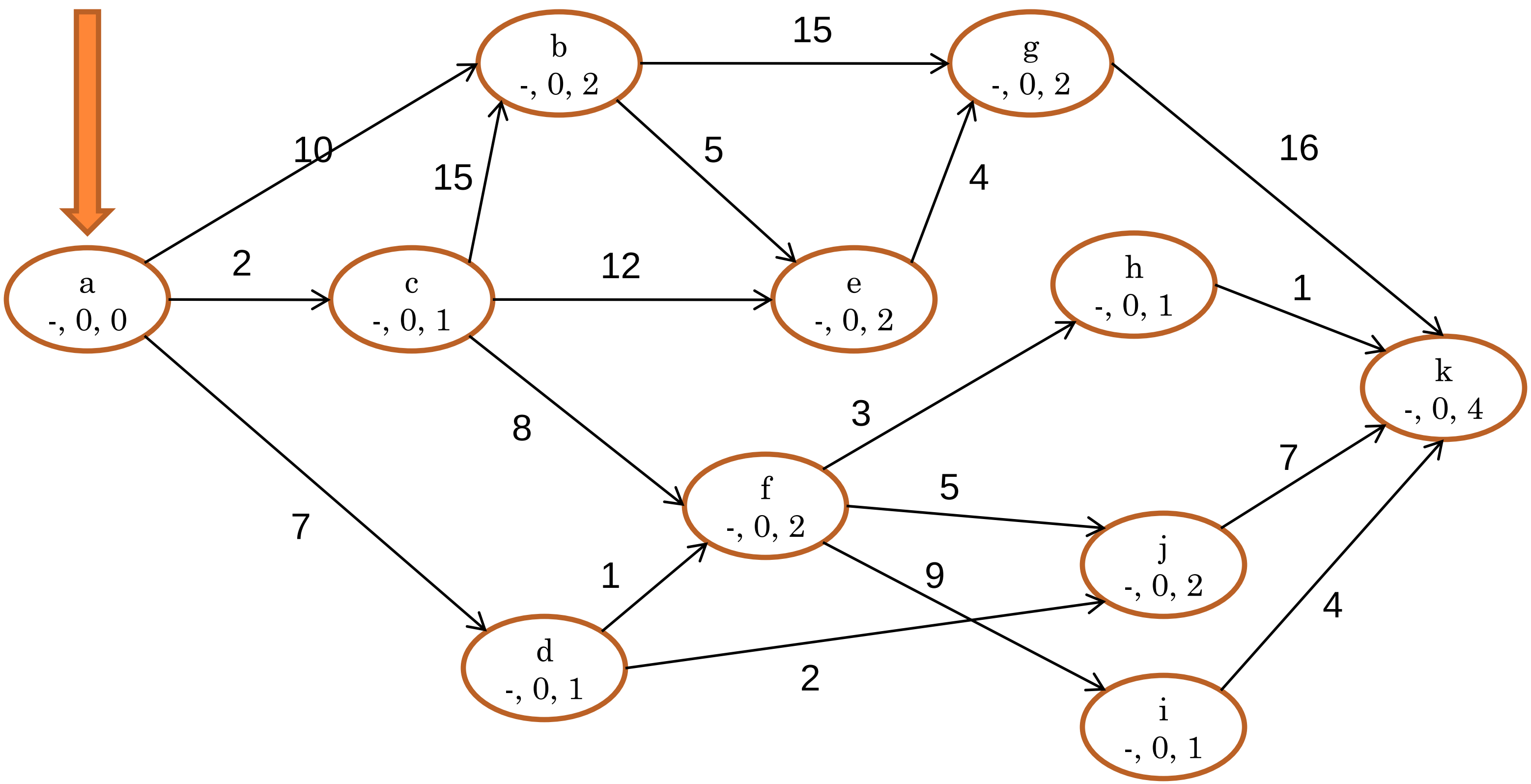


Primer - kritična pot (3/14)

Seznam vozlišč, katerih naslednikov še nismo pregledali:

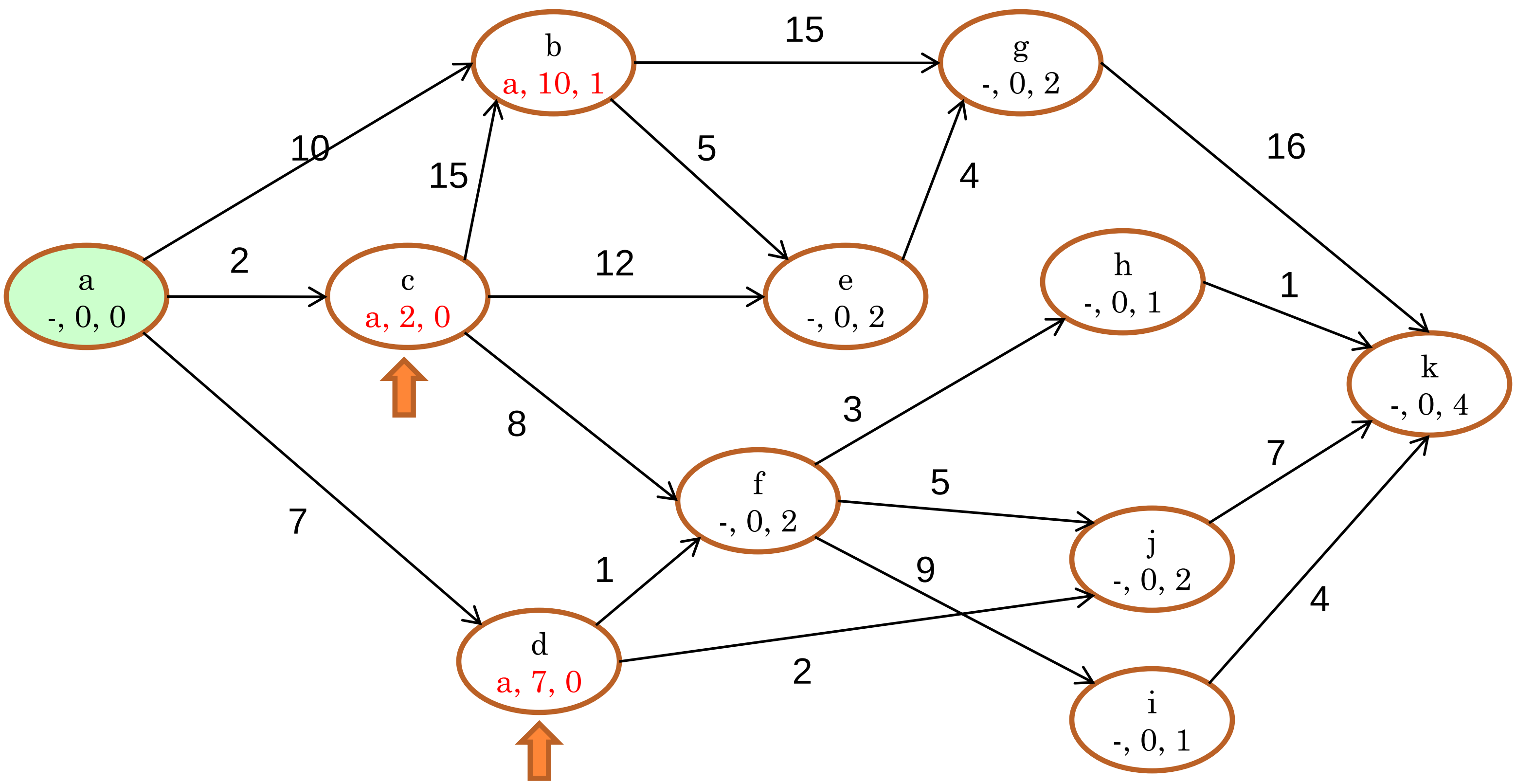
a

v seznam dodajamo samo vozlišča, katerim smo pregledali vse vhode



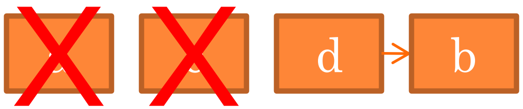
Primer - kritična pot (4/14)

Seznam vozlišč, katerih naslednikov še nismo pregledali:

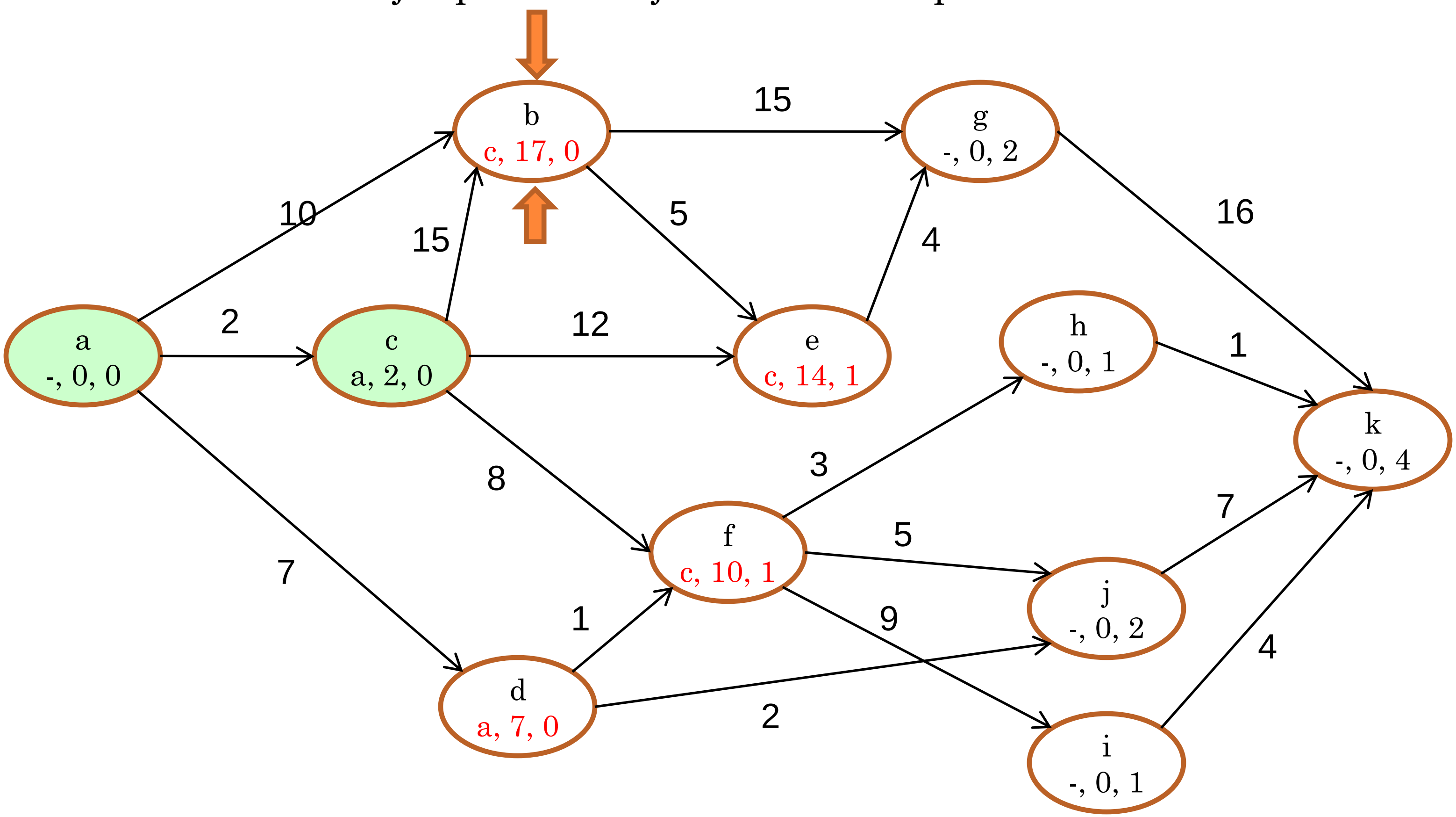


Primer - kritična pot (5/14)

Seznam vozlišč, katerih naslednikov še nismo pregledali:

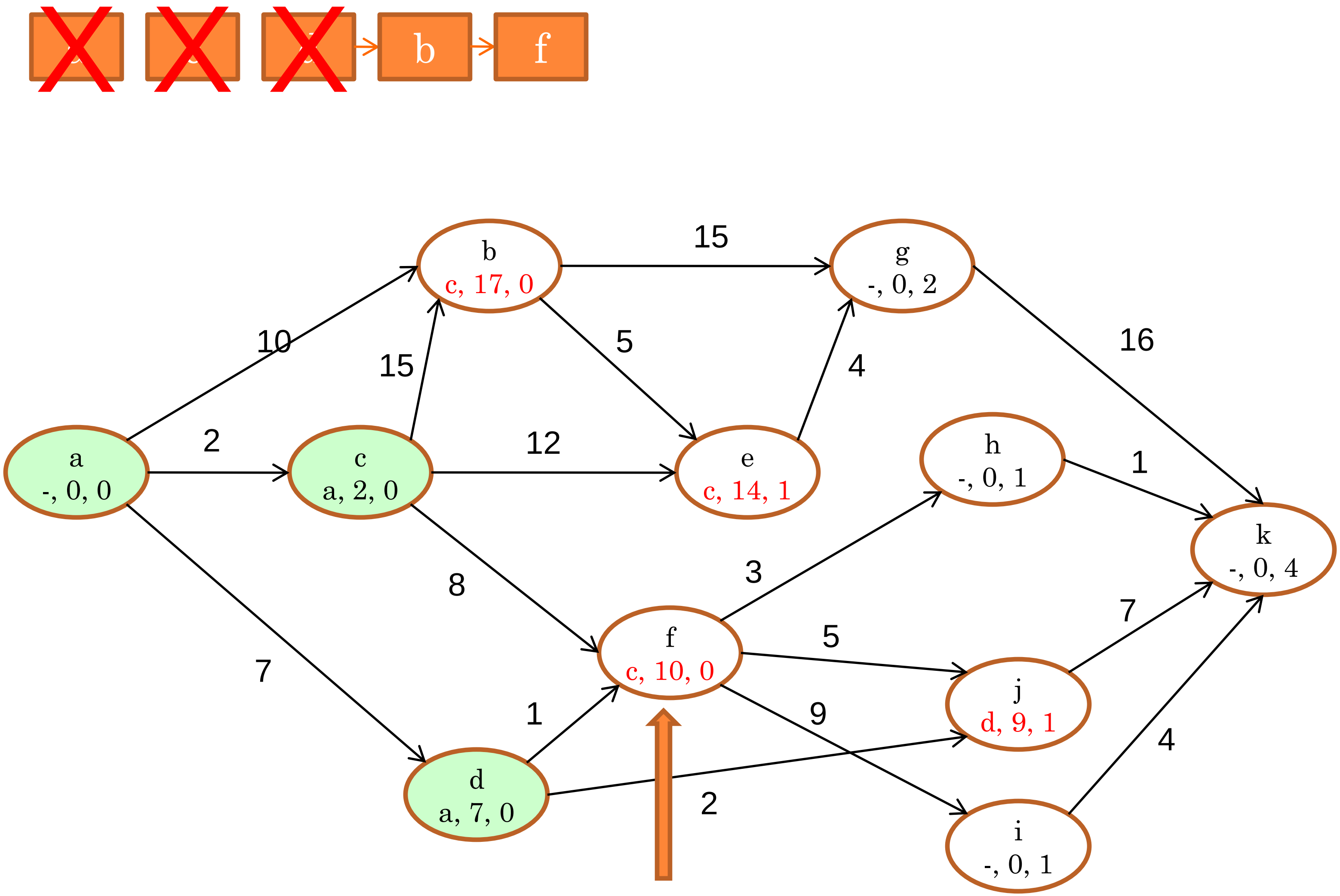


našli smo daljšo pot: razdalja vozlišča ‘c’ + povezava od ‘c’ do ‘b’



PRIMER - KRITIČNA POT (6/14)

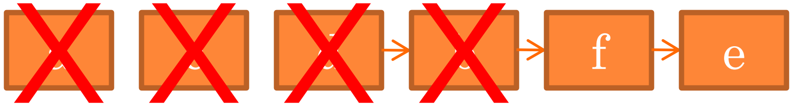
Seznam vozlišč, katerih naslednikov še nismo pregledali:

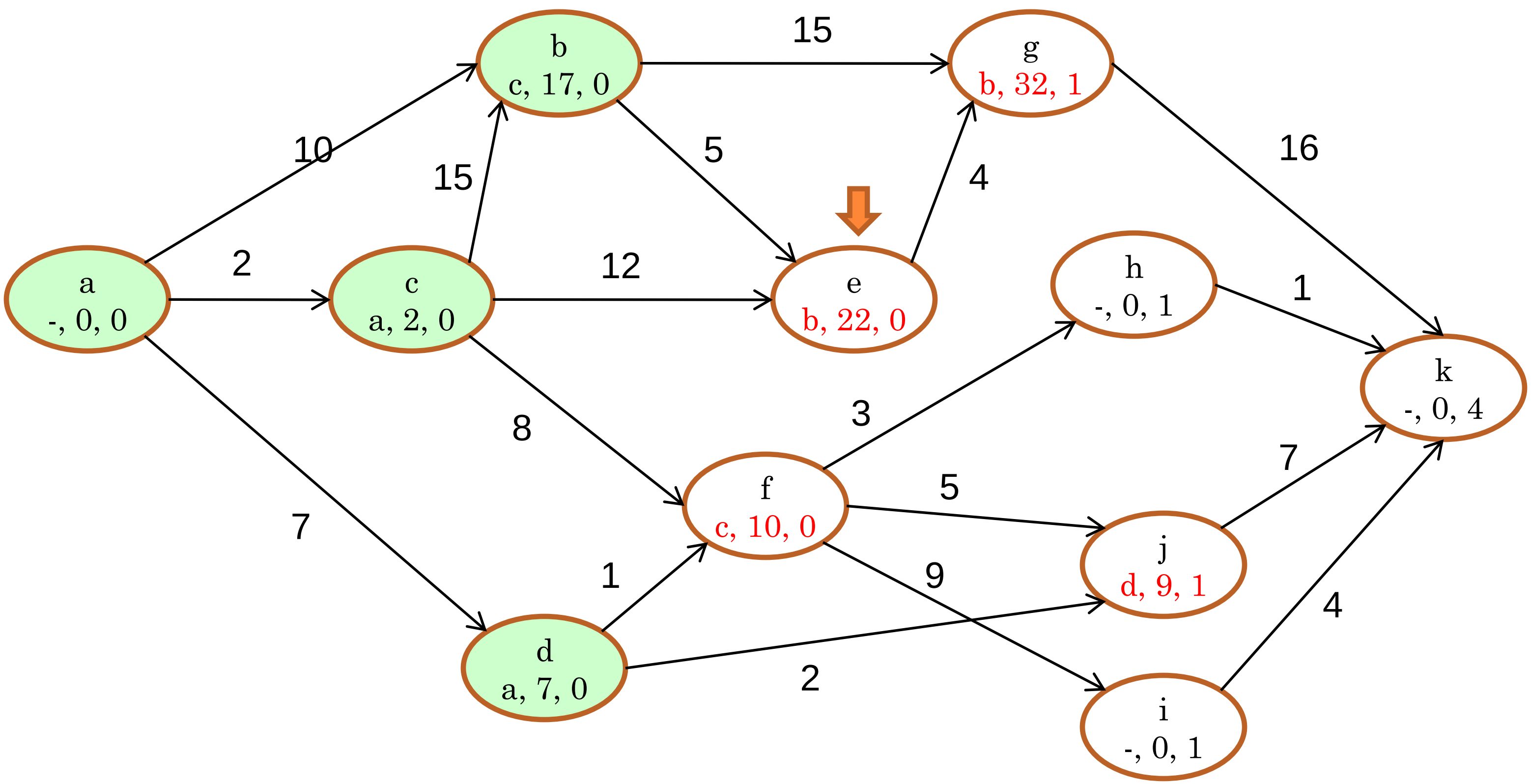


označimo, da je še ena pot pregledana, razdalje ne popravimo

PRIMER - KRITIČNA POT (7/14)

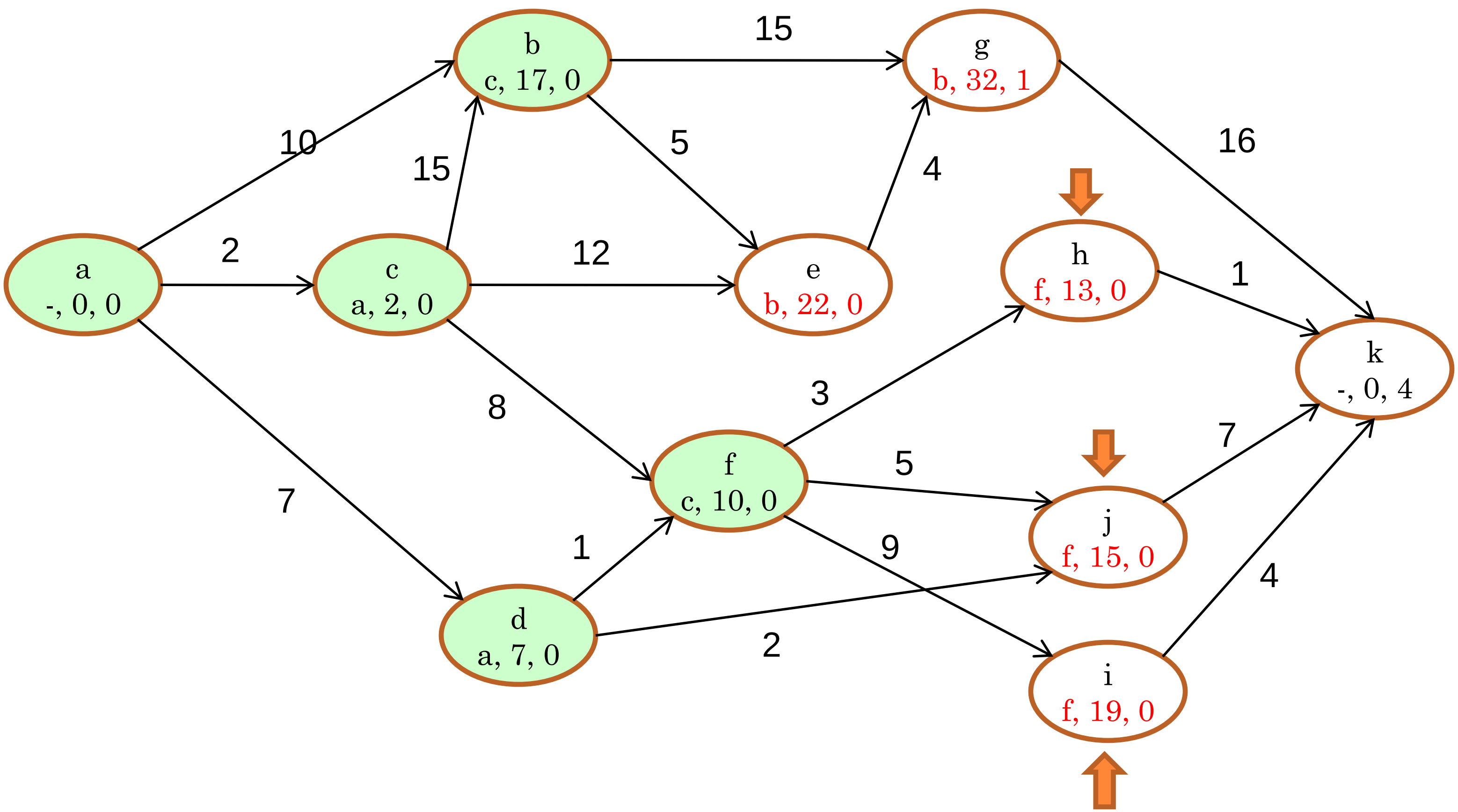
Seznam vozlišč, katerih naslednikov še nismo pregledali:





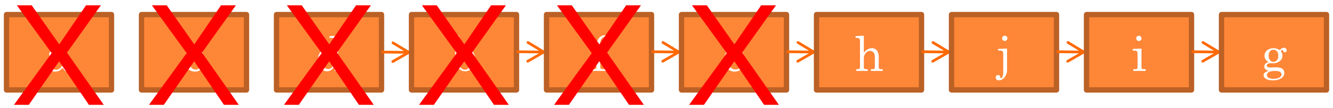
PRIMER - KRITIČNA POT (8/14)

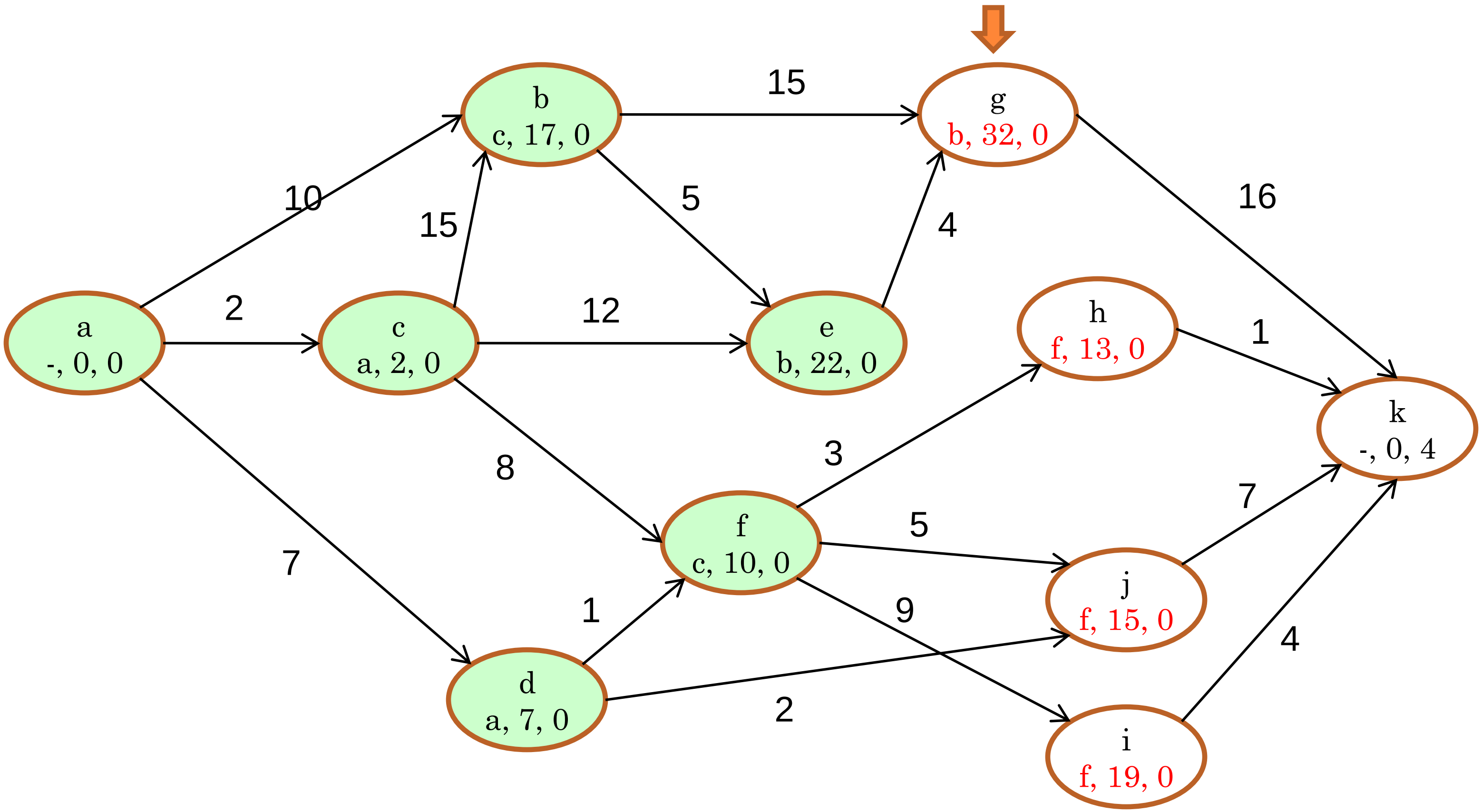
Seznam vozlišč, katerih naslednikov še nismo pregledali:



PRIMER - KRITIČNA POT (9/14)

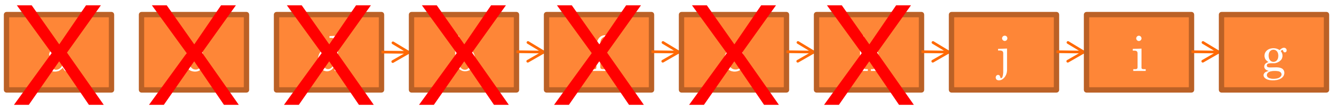
Seznam vozlišč, katerih naslednikov še nismo pregledali:

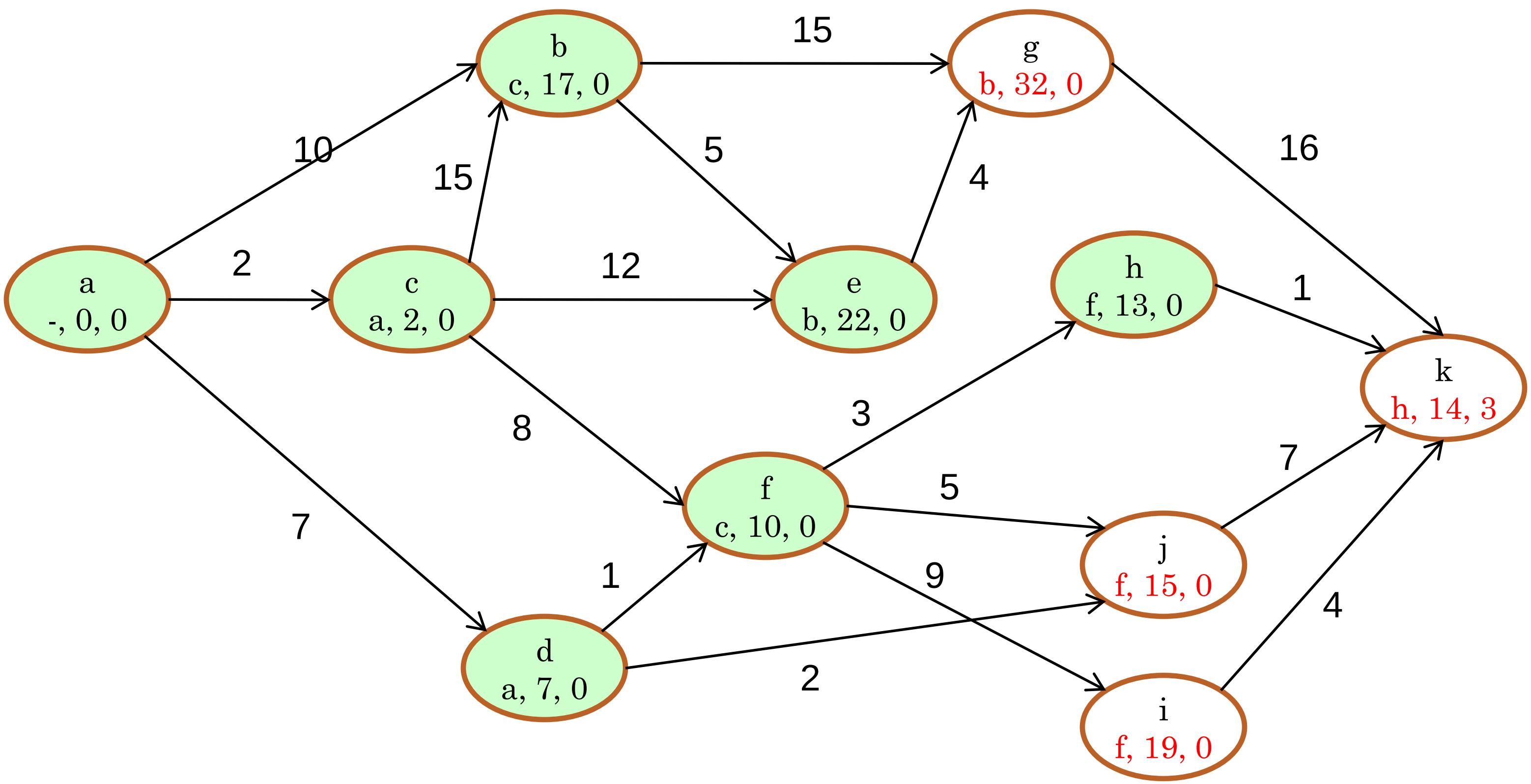




PRIMER - KRITIČNA POT (10/14)

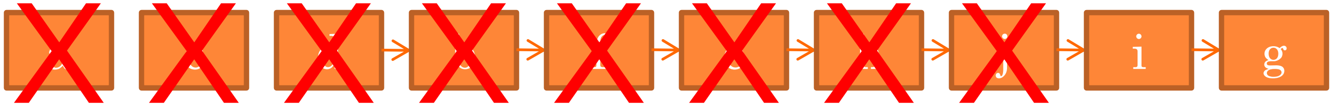
Seznam vozlišč, katerih naslednikov še nismo pregledali:

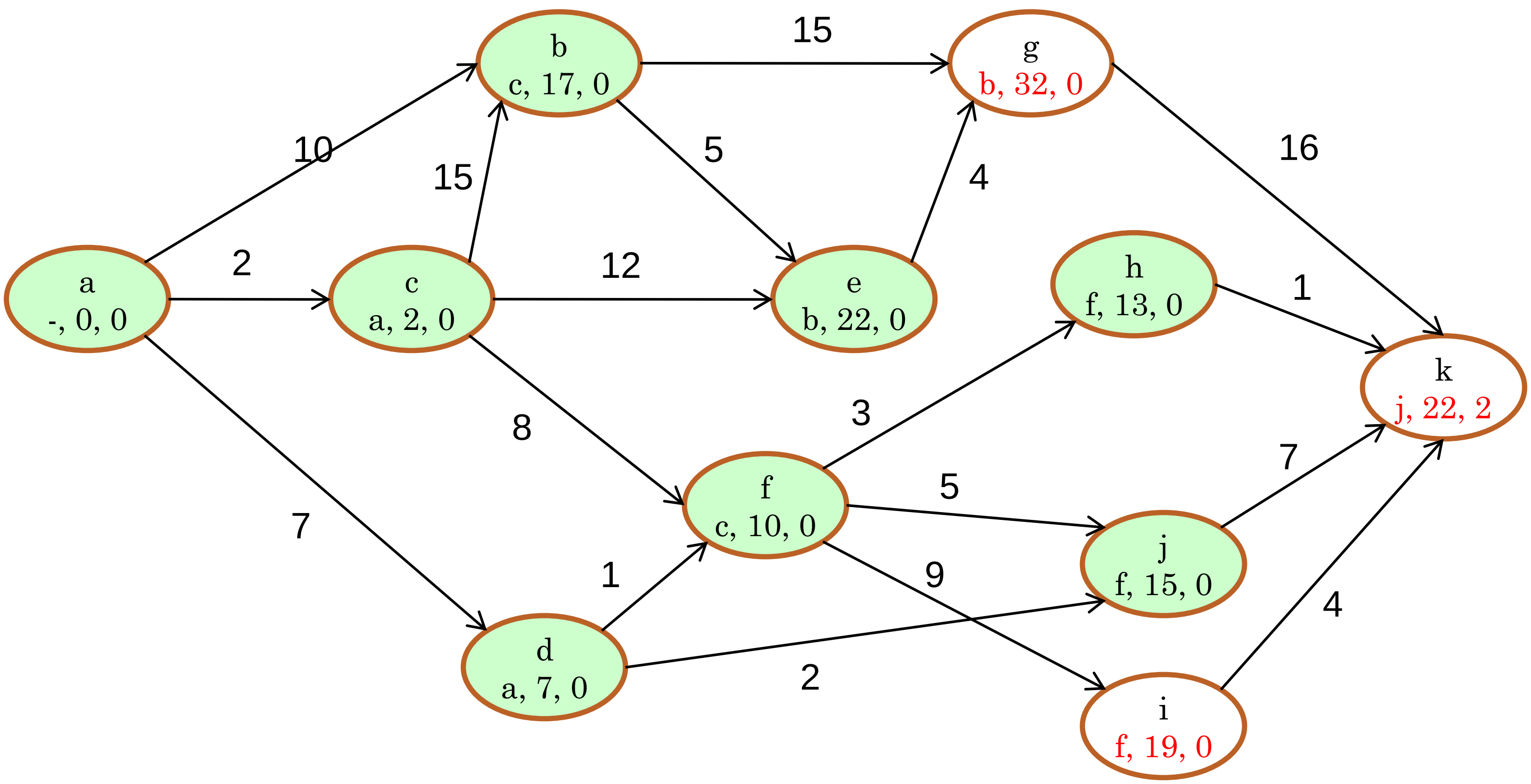




Primer - kritična pot (11/14)

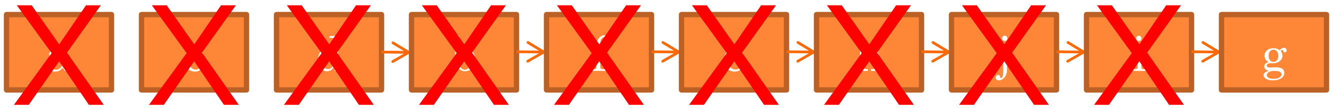
Seznam vozlišč, katerih naslednikov še nismo pregledali:

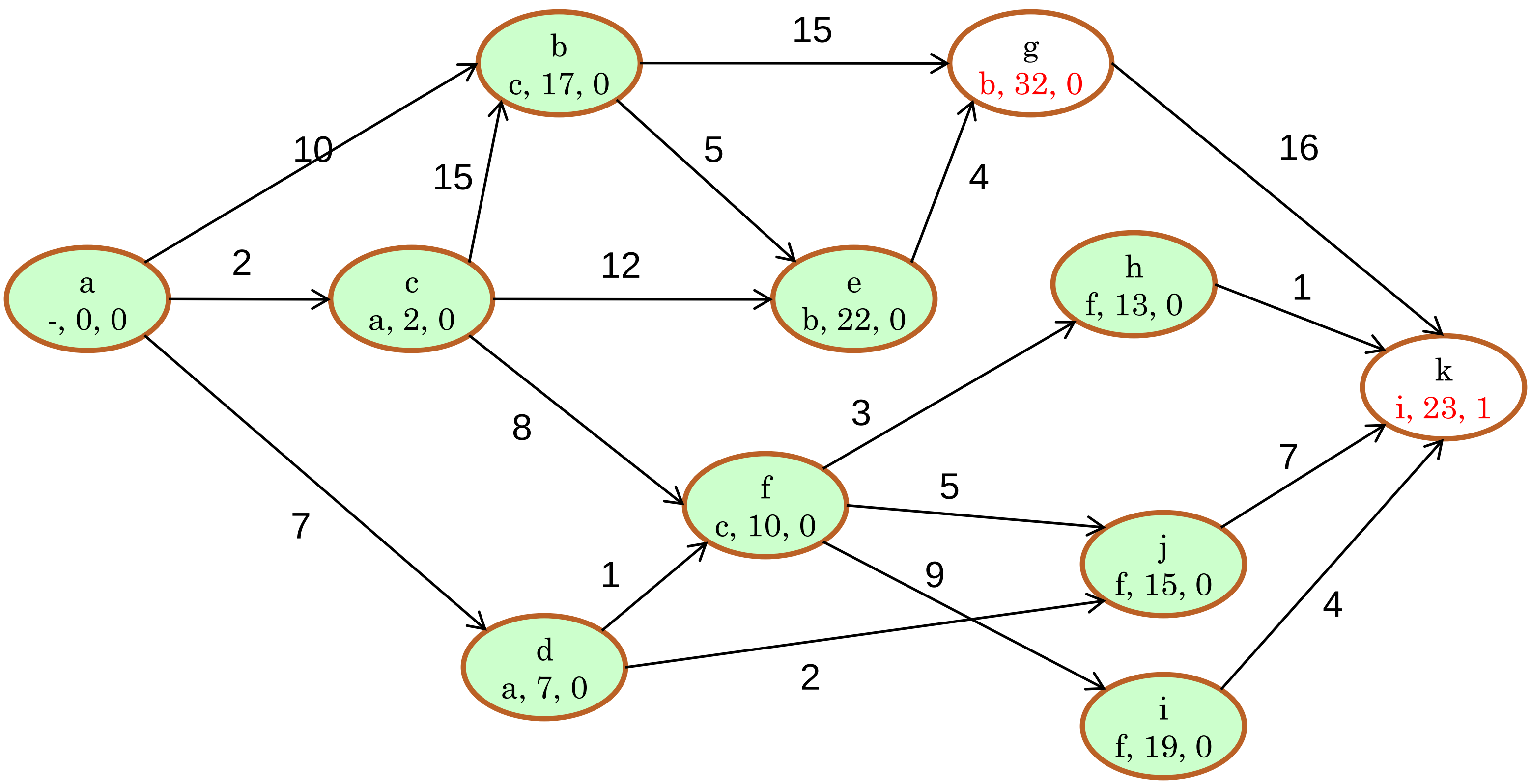




Primer - kritična pot (12/14)

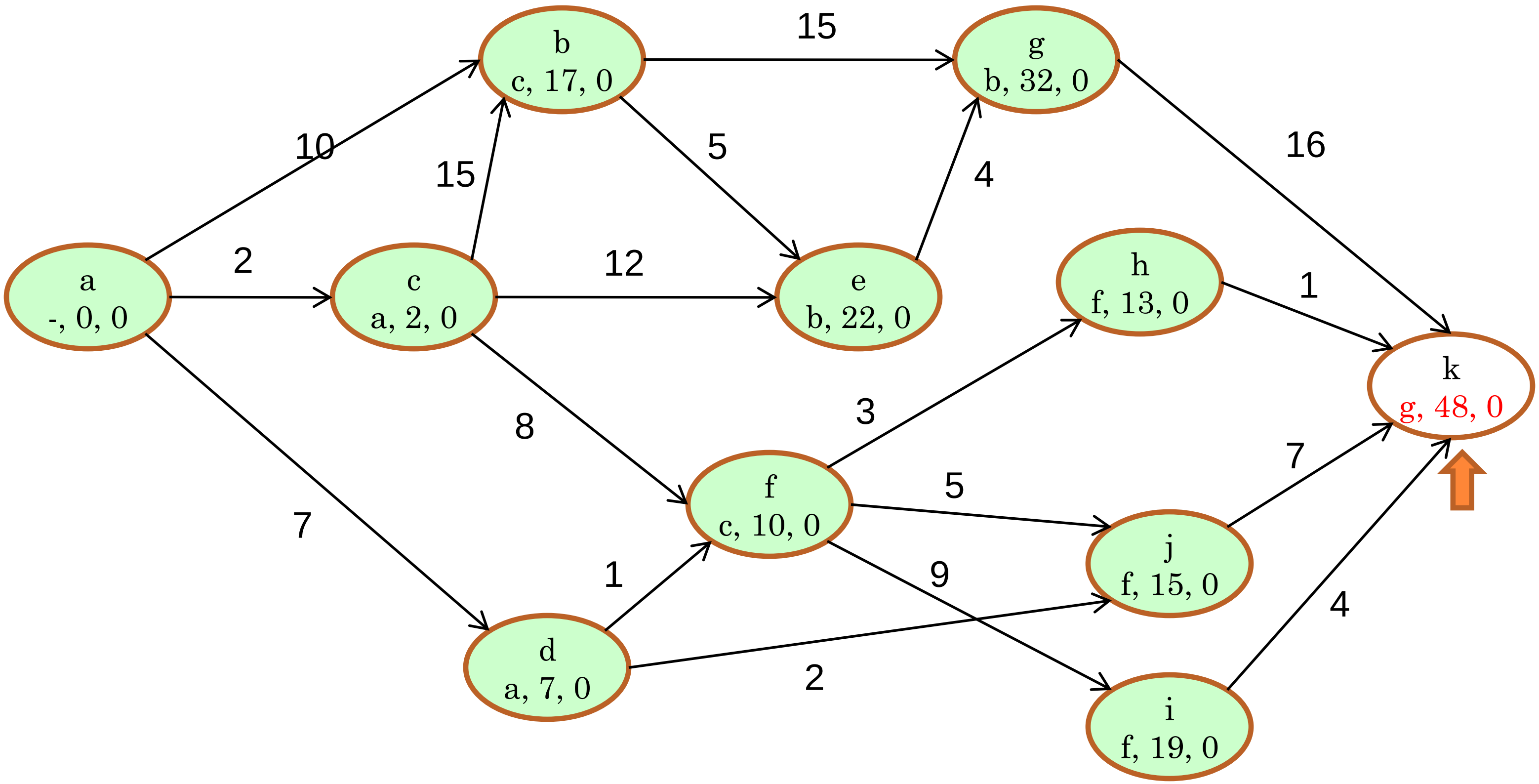
Seznam vozlišč, katerih naslednikov še nismo pregledali:





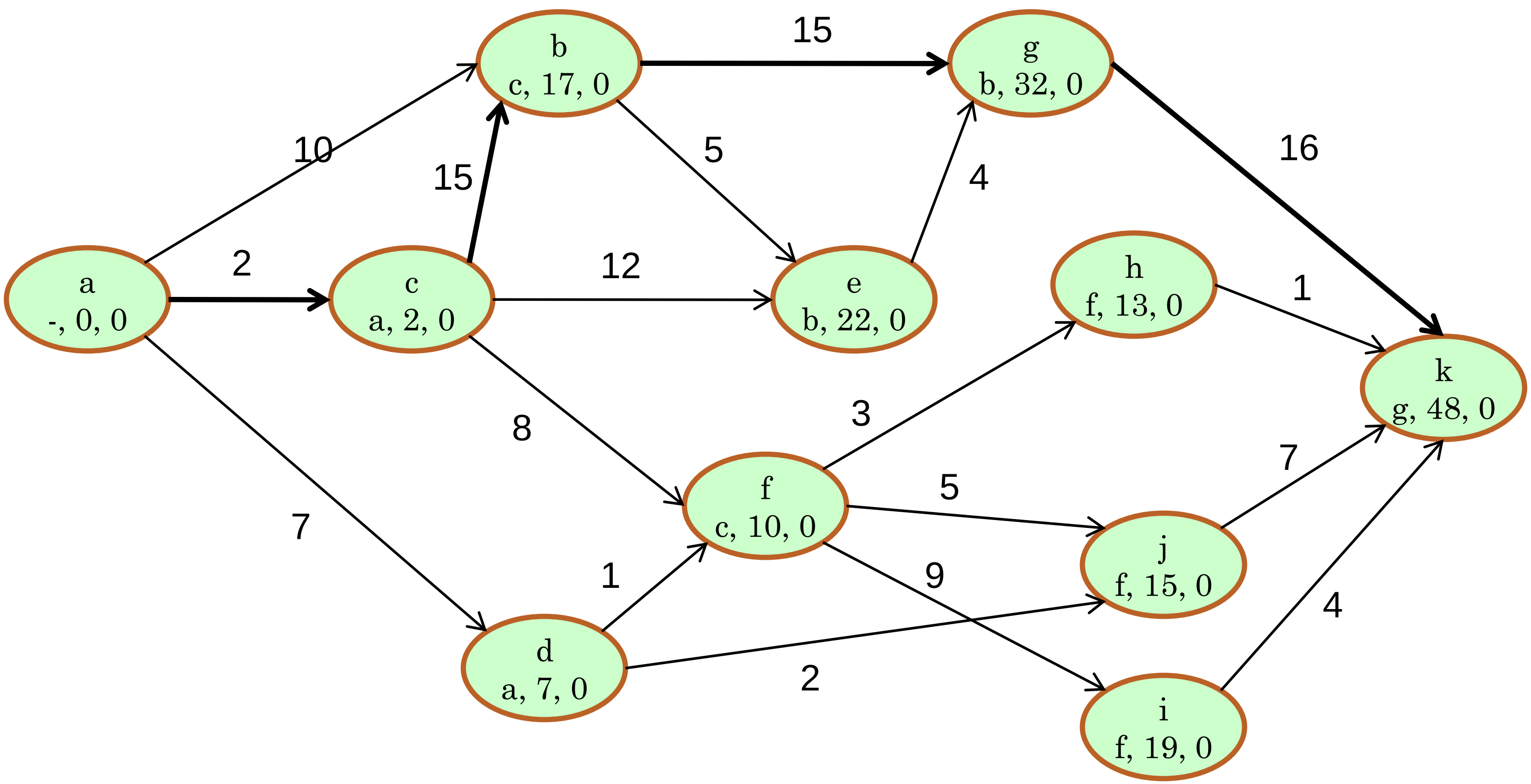
PRIMER - KRITIČNA POT (13/14)

Seznam vozlišč, katerih naslednikov še nismo pregledali:

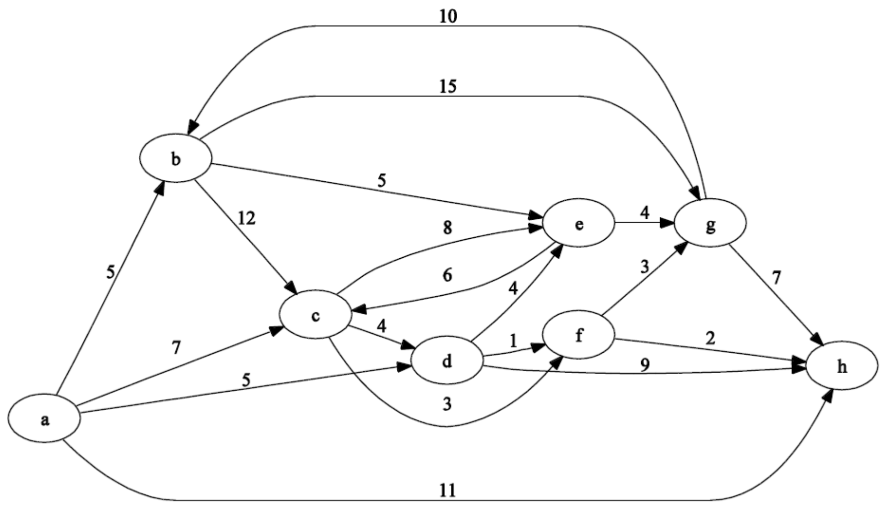


PRIMER - KRITIČNA POT (14/14)

Seznam je prazen, kar pomeni da je postopek končan



Rešitev: a-c-b-g-k v času 48



Iskanje najkrajših poti v grafu

Iskanje najkrajših poti v grafu

- algoritem za kritično pot lahko z majhnimi spremembami uporabimo tudi za iskanje najkrajše poti

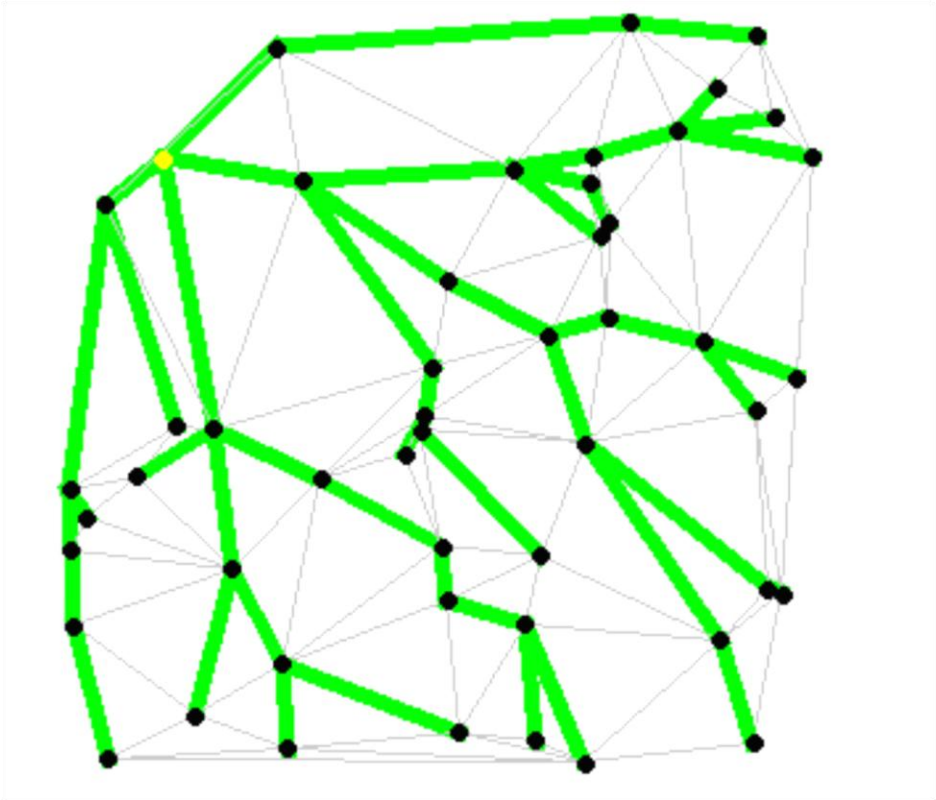
- vendar algoritem za kritično pot predpostavlja:
  - graf je brez ciklov
  - imamo natanko eno zaključno vozlišče

- algoritem Dijkstra
  - poišče najkrajše poti od začetnega vozlišča do vseh vozlišč v povezanem usmerjenem grafu (ki lahko vsebuje tudi cikle)

  - torej drevo najkrajših poti

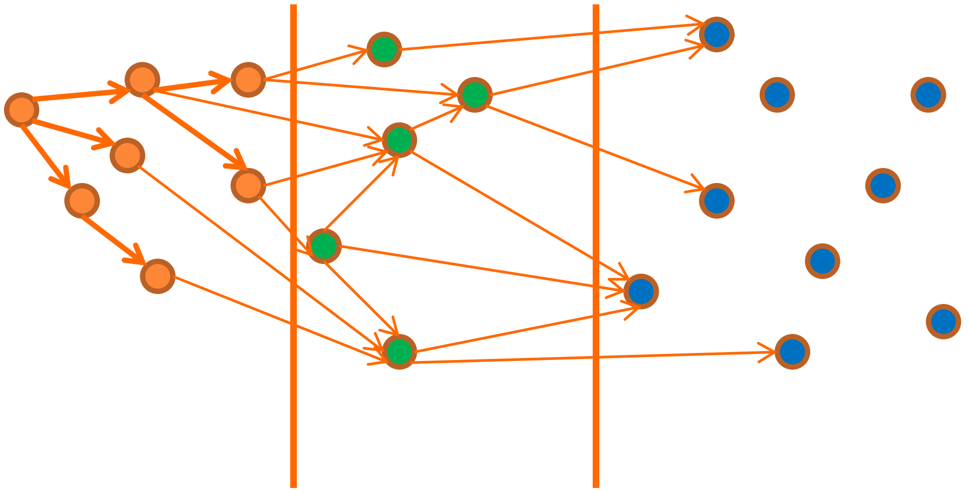
Drevo najkrajših poti

- vsaka najkrajša pot je brez ciklov
- tudi združitev vseh najkrajših poti v en graf ne more vsebovati ciklov, sicer vsaj ena od poti, ki smo jih združevali, ne bi bila najkrajša
- torej združitev vseh najkrajših poti v danem grafu zgradi vpeto drevo



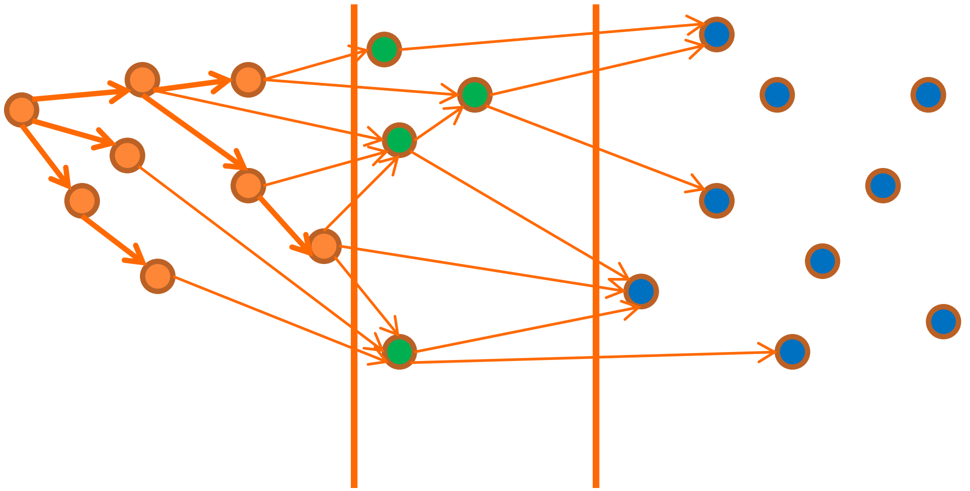
Dijkstra - ideja

- gradimo vpeto drevo od začetnega vozlišča, ki je koren vpetega drevesa, proti listom
- vsakič iz množice vozlišč, ki še niso v drevesu, izberemo tisto z najkrajšo potjo od začetnega vozlišča (**požrešno**)
- to zagotavlja, da ne obstaja krajša pot od začetnega vozlišča do *v* preko nekega drugega vozlišča *w*, ki še ni v drevesu



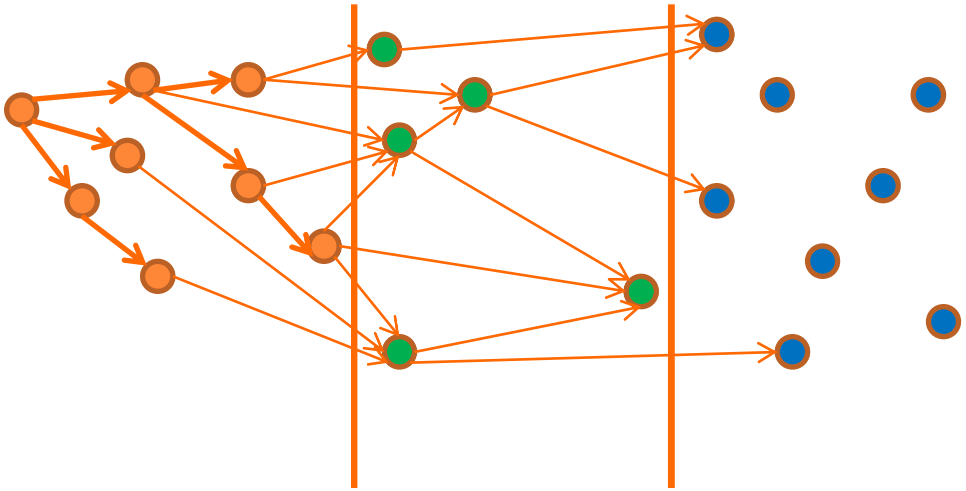
Dijkstra - ideja

- gradimo vpeto drevo od začetnega vozlišča, ki je koren vpetega drevesa, proti listom
- vsakič iz množice vozlišč, ki še niso v drevesu, izberemo tisto z najkrajšo potjo od začetnega vozlišča (**požrešno**)
- ko vozlišče dodamo, pregledamo njegove naslednike:
  1. če je naslednik že v drevesu, ga ignoriramo;
  2. če je že v prioritetni vrsti, eventuelno zmanjšamo prioriteto;
  3. sicer ga vstavimo v prioritetno vrsto;



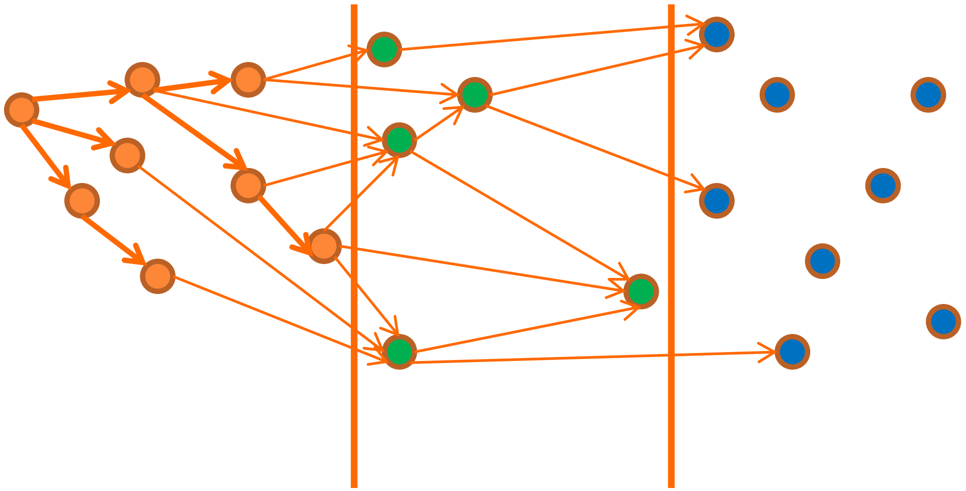
Dijkstra - ideja

- gradimo vpeto drevo od začetnega vozlišča, ki je koren vpetega drevesa, proti listom
- vsakič iz množice vozlišč, ki še niso v drevesu, izberemo tisto z najkrajšo potjo od začetnega vozlišča (**požrešno**)
- ko vozlišče dodamo, pregledamo njegove naslednike:
    1. če je naslednik že v drevesu, ga ignoriramo;
    2. če je že v prioritetni vrsti, eventuelno zmanjšamo prioriteto;
    3. sicer ga vstavimo v prioritetno vrsto;



Dijkstra - implementacija

- za izbiro vozlišča *v* z najkrajšo potjo uporablja algoritem prioritetno vrsto vozlišč, za katera je že znana dolžina vsaj ene poti od začetnega vozlišča *a*
- v prioritetni vrsti se hranijo dolžine **najkrajših znanih poti** za vsako vozlišče
- **z napredovanjem algoritma se te poti lahko skrajšajo,** zato je potrebno uvesti še operacijo zmanjšanja prioritete



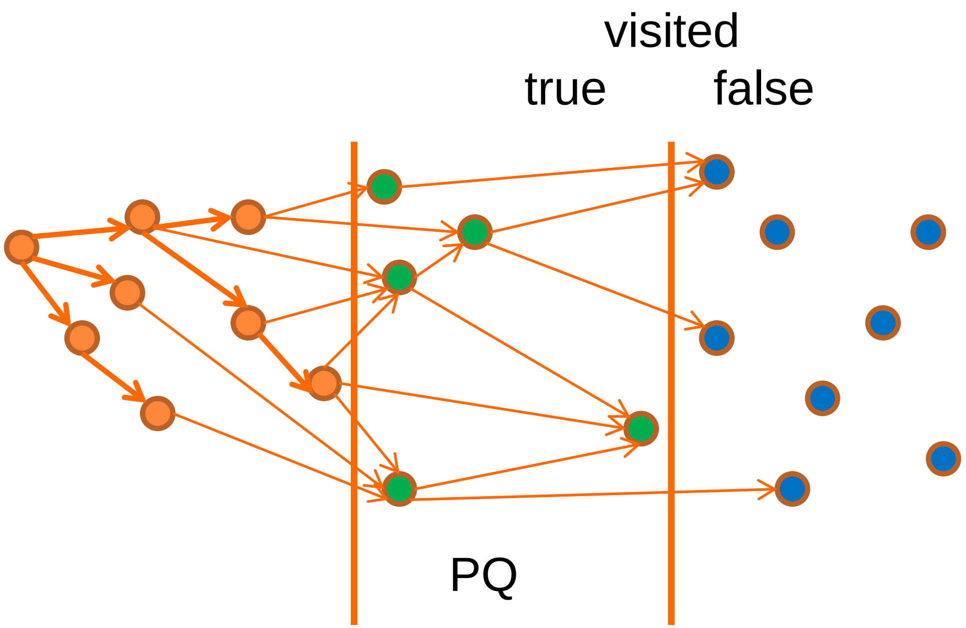
Zmanjšanje prioritete elementa v vrsti

- DECREASE_KEY(x, New, Q)
- zmanjša prioriteto elementa *x* na New
- v kopici operacijo implementiramo tako, da element z zmanjšano prioriteto zamenjujemo z očetom
- postopek se ustavi, bodisi če je oče manjši od elementa ali če element pride v koren kopice
- časovna zahtevnost je reda *O*(log n) pod pogojem, da imamo direkten dostop do elementa v kopici
- **vsako vozlišče hrani svoj položaj (indeks) v kopici**

Dijkstra - implementacija

```
class DijkstraVertex extends VertexAdj implements HeapPosNode {
   boolean visited;
   DijkstraVertex parent;
   double distance ;
   int heapIndex ;
 } // class DijkstraVertex
```

rezultat algoritma



Dijkstra - implementacija

```
public void dijkstra(DijkstraVertex a, DiGraph g) {
  // rezultat sta za vsako vozlisce 'parent' in 'distance'
  PQDecrease q = new HeapPos(); // prioritetna vrsta vozlisc
                                // urejena po distance
  Edge e; // trenutna povezava
  DijkstraVertex v, w ; // trenutno vozlisce in njegov naslednik
 // nobeno vozlisce se ni v prioritetni vrsti
 for (DijkstraVertex t=(DijkstraVertex)g.firstVertex(); t!=null;
          t = (DijkstraVertex)g.nextVertex(t))
       t.visited = false;
 // pripravi zacetno vozlisce in prioritetno vrsto
 a.visited = true;
 a.parent = null;
 a.distance = 0.0 ;
 q.insert(a);
```

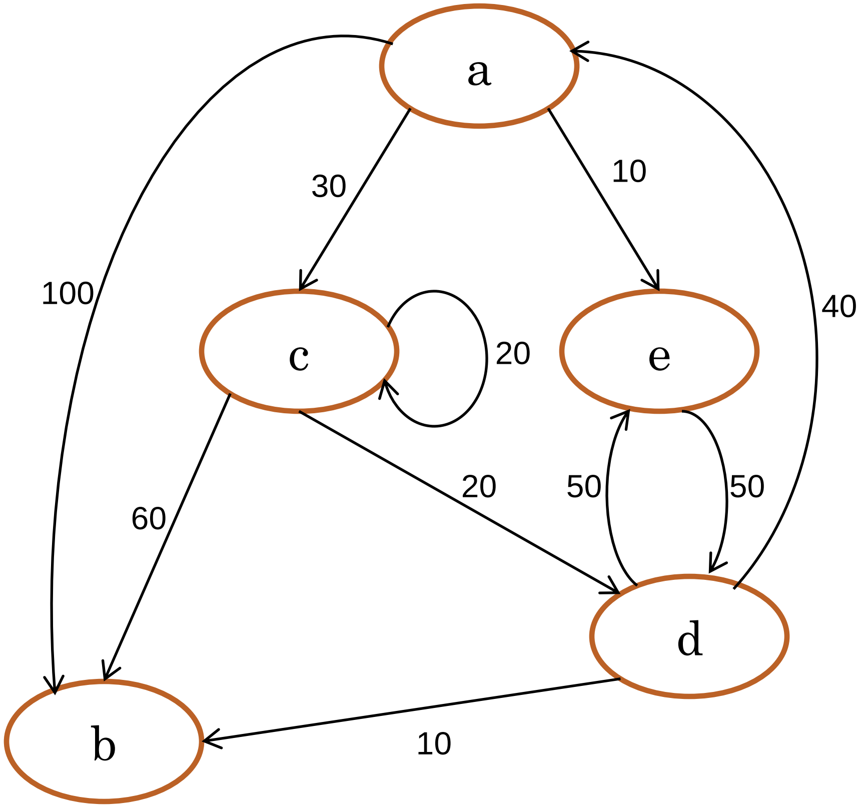
```
  // glavna zanka: dokler ne dodamo v drevo vseh vozlisc
  while (!q.empty()) {
    v = (DijkstraVertex)q.deleteMin();
    e = g.firstEdge(v);
    while (e != null) {
      w = (DijkstraVertex)g.endPoint(e); // naslednik vozlisca v
      if (!w.visited) {
        // uredi w in dodaj v prioritetno vrsto
        w.visited = true;
        w.parent = v;
        w.distance = v.distance+((Double)e.evalue).doubleValue();
        q.insert(w);
      }
      else if (v.distance + ((Double)e.evalue).doubleValue() <
                w.distance) { // nova, krajsa pot do w

        w.parent = v;
        q.decreaseKey(w, new Double(v.distance +
                             ((Double)e.evalue).doubleValue()));
      }
      e = g.nextEdge(v, e);
    } // while e
  } // while !empty
} // dijkstra
```

Dijkstra – časovna zahtevnost

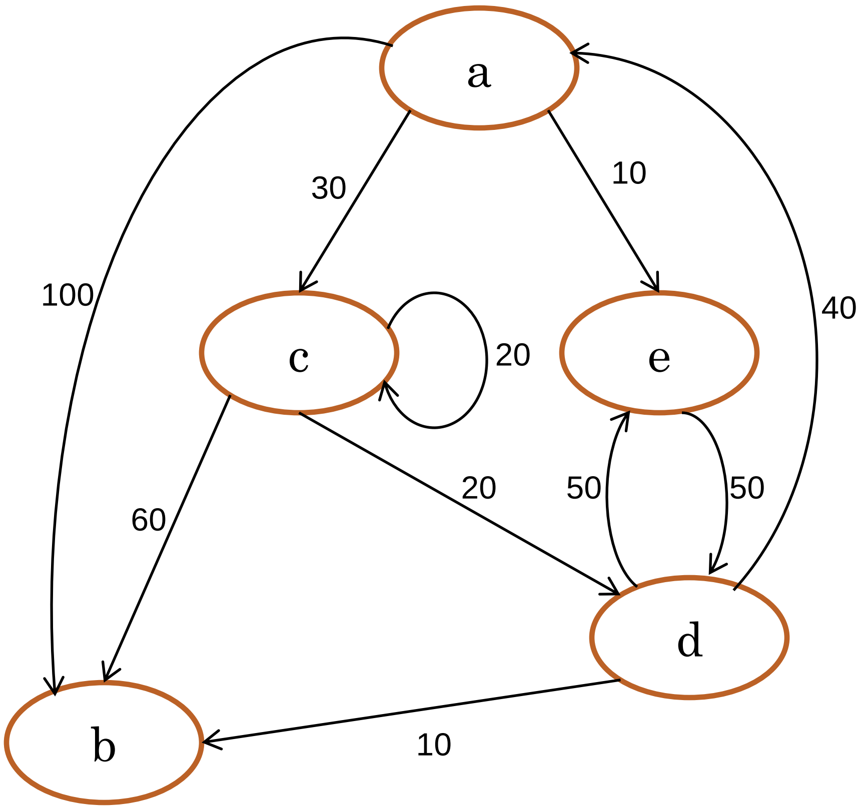
- vsako vozlišče dodamo in zbrišemo iz prioritetne vrste, torej n operacij INSERT in n operacij DELETEMIN
- notranja zanka gre preko vseh povezav, torej se izvrši m-krat (ena izvršitev zahteva bodisi INSERT bodisi DECREASE KEY ali pa nobene od teh operacij)
- če implementiramo prioritetno vrsto s kopico, potem je časovna zahtevnost v najslabšem primeru reda $O(2n \log n + m \log n) = O((n + m) \log n)$
- ker za povezan graf velja $m \geq n-1$, je časovna zahtevnost algoritma reda $O(m \log n)$
- Algoritem Dijkstra je **POŽREŠEN,** pa vseeno zagotavlja optimalno rešitev.

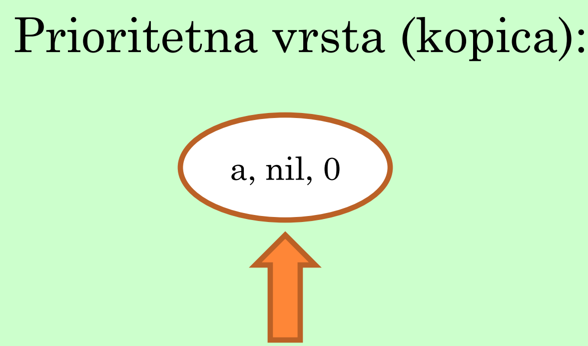
Primer izvajanja algoritma Dijkstra (1/7)

Podan je graf na sliki. Izračunajte najkrajše poti od vozliča ‘a’ do vseh ostalih vozlišč v grafu.



Primer izvajanja algoritma Dijkstra (2/7)

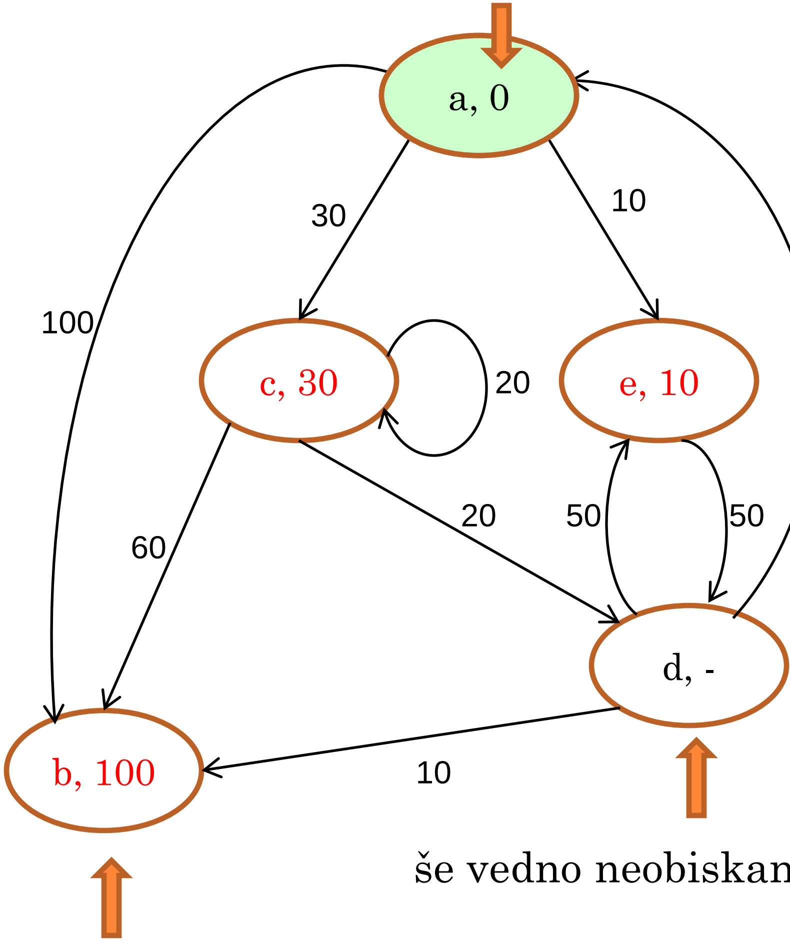




zapis vsebuje:
- ime vozlišča
- ime očeta v vpetem drevesu
- dolžino najkrajše znane poti od začetnega vozlišča

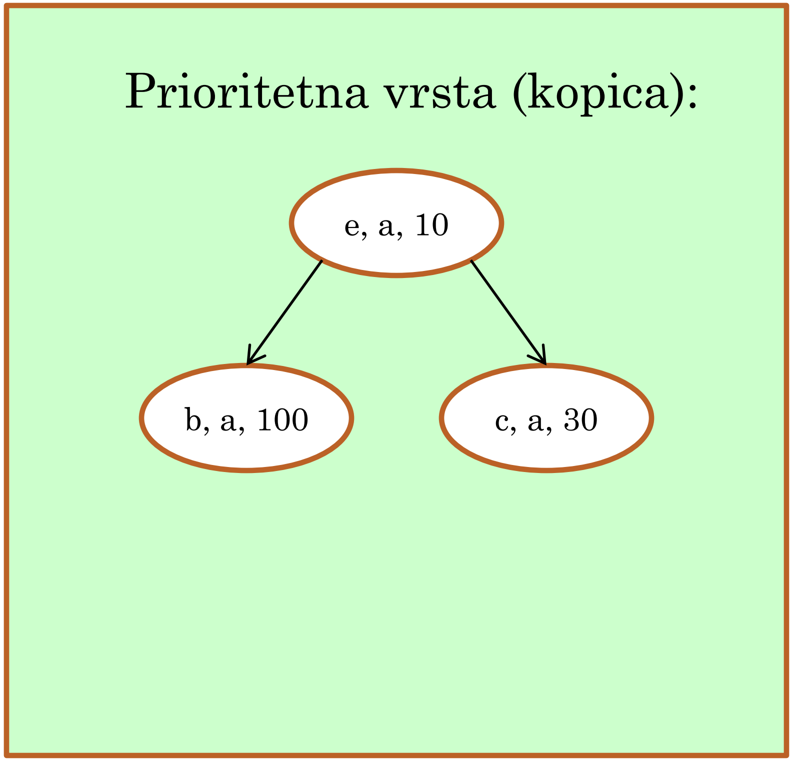
Primer izvajanja algoritma Dijkstra (3/7)

odstranjeni element iz prioritetne vrste
vsebuje rezultat

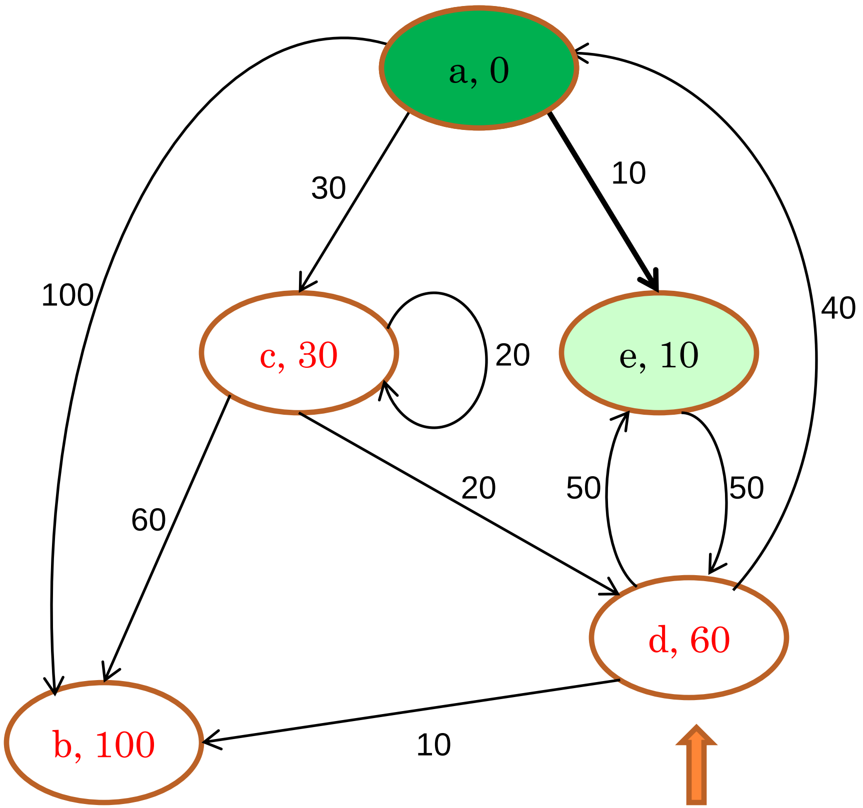


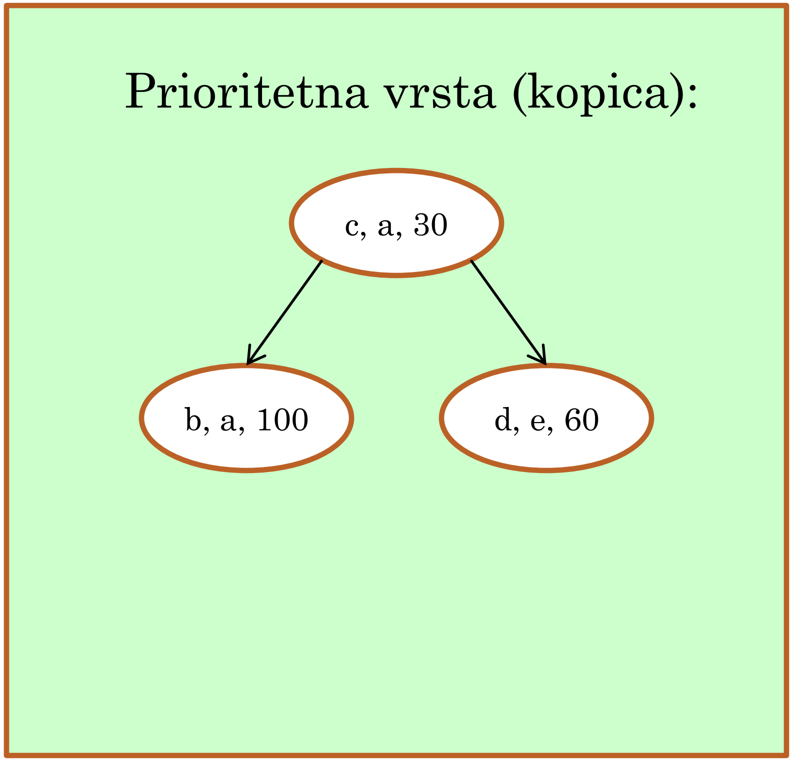
še vedno neobiskano vozlišče

razdalja vozlišča ‘a’ + povezava od ‘a’ do ‘b’



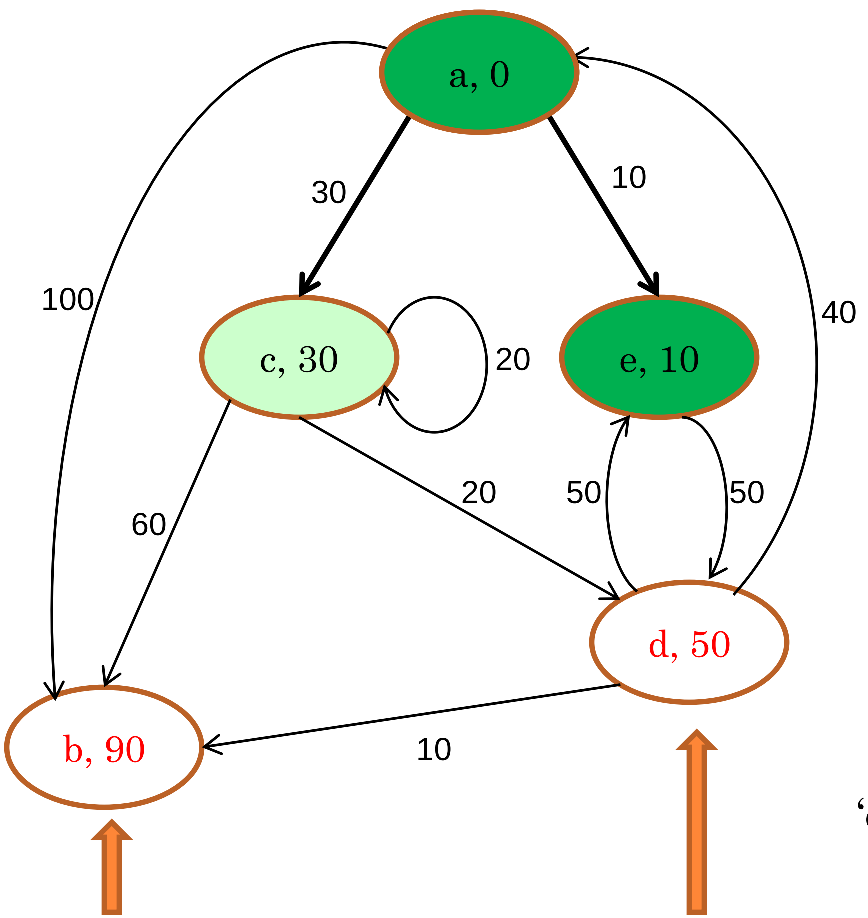
Primer izvajanja algoritma Dijkstra (4/7)



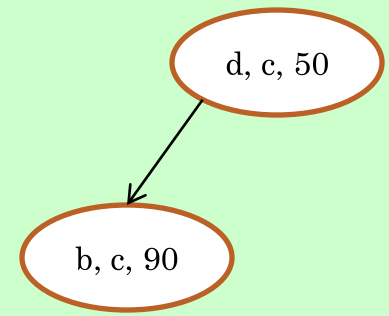


razdalja vozlišča ‘e’ + povezava od ‘e’ do ‘d’

PRIMER IZVAJANJA ALGORITMA DIJKSTRA (5/7)



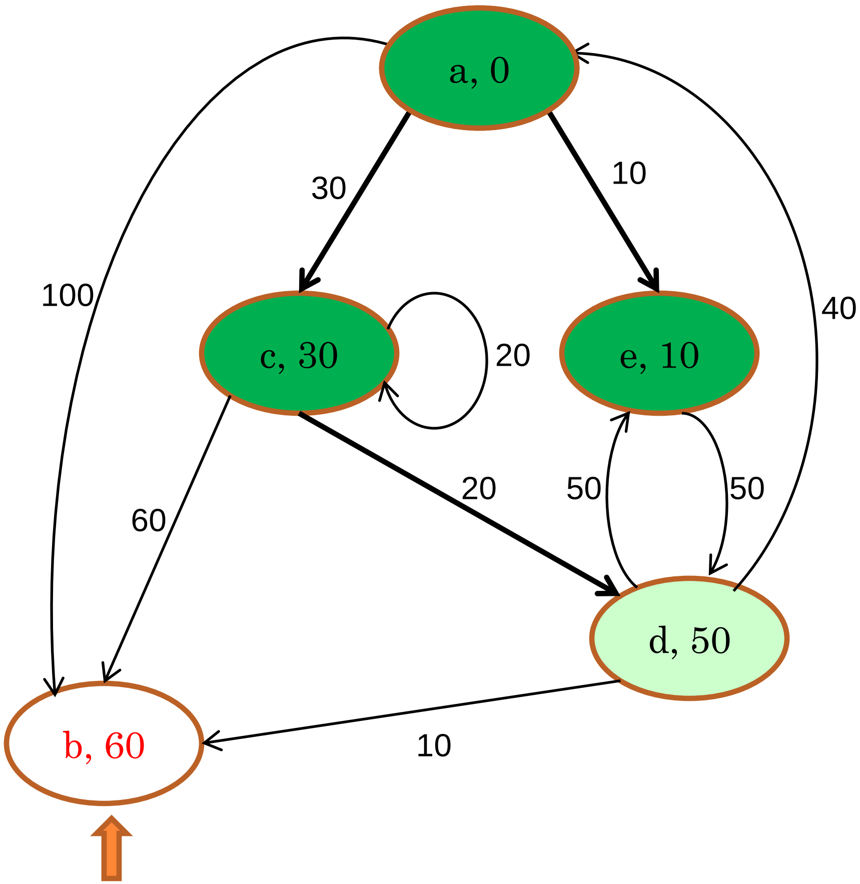
Prioritetna vrsta (kopica):

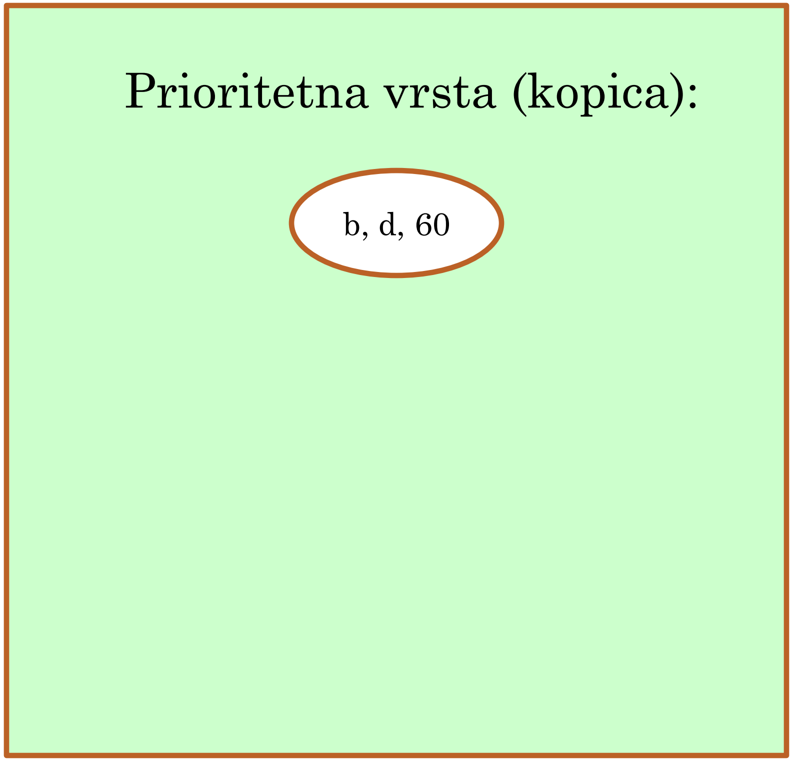


'c' ne dodamo v prioritetno vrsto, ker je razdalja 30 + 20 daljša od najkrajše znane razdalje 30

našli smo krajšo povezavo (čez vozlišče 'c')

Primer izvajanja algoritma Dijkstra (6/7)



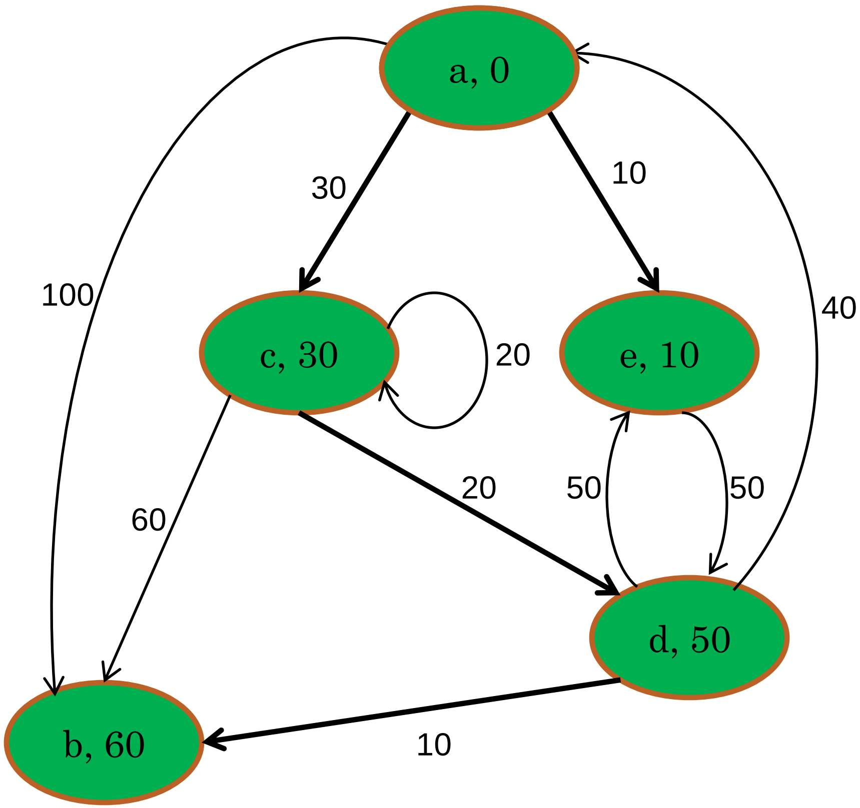


našli smo krajšo povezavo:
razdalja vozlišča ‘d’ + povezava od ‘d’ do ‘b’

‘a’ ne dodamo v pr. vrsto, ker je razdalja
50 + 40 daljša od najkrajše znane razdalje 0

‘e’ ne dodamo v pr. vrsto, ker je razdalja
50 + 50 daljša od najkrajše znane razdalje 10

Primer izvajanja algoritma Dijkstra (7/7)



Prioritetna vrsta (kopica):

prioritetna vrsta je prazna kar pomeni, da je postopek končan

Poseben primer: Vse povezave enako dolge

- Namesto prioritetne vrste zadošča navadna vrsta : *O(log n)* → *O(1)*

- w.distance = v.distance + 1 ;

- Vrstni red iskanja nam zagotavlja, da ko do vozlišča najdemo prvo pot, je ta najkrajša.

- Drevo gradimo po nivojih: “iskanje v širino”

POSEBEN PRIMER: VSE POVEZAVE ENAKO DOLGE

```
public void breadthFirstSearch(DijkstraVertex a, DiGraph g) {
  // rezultat sta za vsako vozlisce 'parent' in 'distance'
  Queue q = new QueueArray(); // vrsta vozlisc implicitno
                              // urejena po distance
  Edge e; // trenutna povezava
  DijkstraVertex v, w ; // trenutno vozlisce in njegov naslednik

  // nobeno vozlisce se ni v prioritetni vrsti
  for (DijkstraVertex t=(DijkstraVertex)g.firstVertex(); t!=null;
           t = (DijkstraVertex)g.nextVertex(t))
        t.visited = false;
  // pripravi zacetno vozlisce in vrsto
  a.visited = true;
  a.parent = null;
  a.distance = 0.0 ;
  q.enqueue(a);
```

Poseben primer: Vse povezave enako dolge

```
// glavna zanka: dokler ne dodamo v drevo vseh vozlisc
while (!q.empty()) {
        v = (DijkstraVertex)q.front();
        q.dequeue();
        e = g.firstEdge(v);
        while (e != null) {
           w = (DijkstraVertex)g.endPoint(e); // naslednik vozlisca v
           if (!w.visited) {
              // uredi w in dodaj v prioritetno vrsto
              w.visited = true;
              w.parent = v;
              w.distance = v.distance + 1 ;
              q.enqueue(w);
           }
           e = g.nextEdge(v, e);
        } // while e
    } // while !empty
} // breadthFirstSearch
```