Osnove algoritmov in podatkovnih struktur II

zapiski za leto 2005/06

- doc. dr. Marko Robnik Šikonja
- http://oaps2.fri.uni-lj.si
- marko.robnik@fri.uni-lj.si
- objavljena koda je psevdokoda, za dejansko izvajanje jo je potrebno nekoliko prilagoditi

Rekurzija - iteracija

- Evklidov algoritem za iskanje največjega skupnega deljitelja

$$\gcd(x, y) = \begin{cases} y & ; x \bmod y = 0 \\ \gcd(y, x \bmod y) & ; sicer \end{cases}$$

- Primera:
gcd(120,25) = gcd(25,20) = gcd(20,5) = 5
gcd(24,64) = gcd(64,24) = gcd(24,16) = gcd(16,8) = 8

Evklidov algoritem za gcd

Rekurzivno:

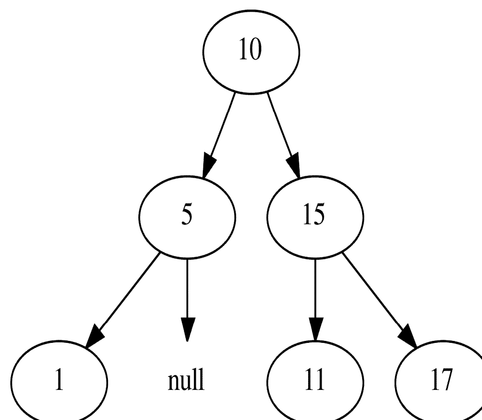
```
public static int gcdR(int x, int y)
{
   if  (x % y == 0)
     return y ;
   else
     return gcdR(y, x % y);
}
```

• Iterativno

```
public static int gcdI(int x, int y)
{
   int temp;
   while (x %  y != 0) {
     temp = x ;
     x = y ;
     y = temp mod y ;
   }
   return y ;
}
```


Rekurzivne podatkovne strukture BINARNO DREVO

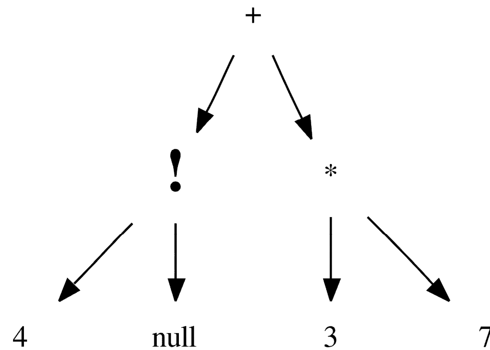
```
class Node {
   Object key ;
   Node left, right ;
}
```



Infiksni zapis

- 4!+3*7

- prefiksno: + ! 4 * 3 7
- infiksno: 4 ! + 3 * 7
- postfiksno: 4 ! 3 7 * +



Infiksni izpis rekurzivno

```
public static void infix(Node p)
{
   if (p != null)  {
      infix(p.left) ;
      System.out.print(p.key) ;
      infix(p.right) ;
   }
}
```

Spreminjanje rekurzije v iteracijo s skladom

- simuliramo obnašanje rekurzivne procedure
- na sklad moramo shraniti:
  - argumente procedure
  - lokalne spremenljivke
  - naslov izvajanja (povratno točko)
- v zanki simuliramo rekurzivne klice
- na sklad shranimo podatke za povratek in za klic

Infiksni izpis iterativno s skladom

```
public static void infix(Node p)
{
0 →   if (p != null )   {
           infix(p.left) ;
1 →        System.out.print(p.key) ;
           infix(p.right) ;
2 →   }
}
```

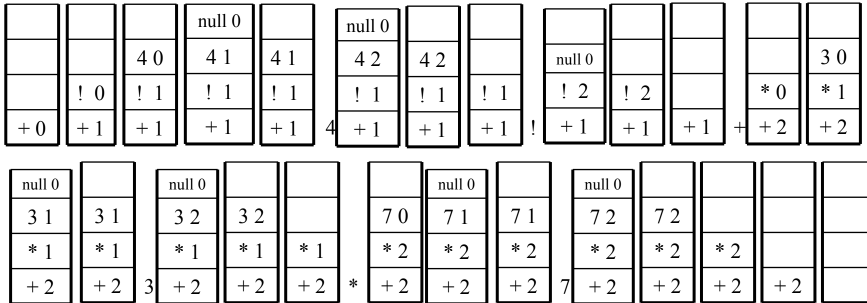
- podatkovna struktura za element sklada

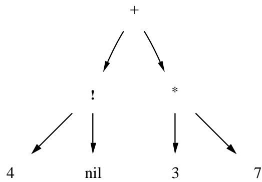
```
class StackElement {
   Node p ;
   int a ;
   StackElement() {}
   StackElement(StackElement e) {
     p = e.p ; a = e.a ;
   }
}
```



```
public static void infixIterativno(Node p) {
  Stack s = new StackArray();

  StackElement e = new StackElement();
   e.a = 0; e.p = p; s.push(e);
   do {
       e = (StackElement) s.top() ; s.pop() ;
       switch (e.a) {
           case 0: if (e.p != null)  {
                       e.a = 1; s.push(new StackElement(e));  // za povratek
                       e.a = 0;  e.p = e.p.left ; s.push(new StackElement(e)) ; // za klic
                     }
                     break ;
           case 1: System.out.print(e.p.key) ;
                     e.a = 2; s.push(new StackElement(e)); // za povratek
                     e.a = 0; e.p = e.p.right ; s.push(new StackElement(e) ) ; // za klic
           case 2:break ;
       } // switch
  } while (! s.empty()) ;
}  // infixIterativno
```

Dogajanje na skladu





Repna rekurzija

- rekurzivni klic v zadnjem ukazu v proceduri
- tak klic lahko eliminiramo - spremenimo v iteracijo
- pri zadnjem klicu se izvajanje prenese na začetek rekurzije (s spremenjenimi parametri)

Infiksni izpis rekurzivno brez repne rekurzije

```
public static void infix(Node p)
{
      if (p != null)  {
         infix(p.left) ;
         System.out.print(p.key) ;
         infix(p.right) ;
      }
}
```

```
public static void infix0t(Node p)
{
    while (p != null) {
        infix(p.left) ;
        System.out.print(p.key) ;
        p = p.right ;
    }
}
```


Spreminjanja rekurzije v zanki v iteracijo s skladom

```
public static void loopRek(int m, int n)
{
   if (n ==1)
      System.out.println("+") ;
   else
        for (int i=0; i < m ; i++) // kakšen je vrstni red klicov?
          loopRek(i, n-1) ; // je to repna rekurzija?
}
```

```
public static void loopRekEq(int m, int n)
{
    if (n == 1)
      System.out.println("+") ;
    else {
      int i = 0 ;
      while  (i < m) {
        loopRekEq(i, n-1) ;
        i = i + 1 ;
      }
    }
}
```



```
class  StackElement {
   int m, n, i, a ;
   StackElement() {}
   StackElement(StackElement e) {
       m=e.m;  n = e.n;  i = e.i;  a = e.a;
   }
}
```

```
public static void loopRekIterativno(int m, int n)
{
    Stack s = new StackArray();
    StackElement e = new StackElement();
    e.a = 0; e.m = m; e.n = n; s.push(e);
    do {
        e = (StackElement) s.top() ; s.pop() ;
        switch (e.a) {
          case 0: if (e.n == 1)
                       System.out.println("+") ;
                  else {
                     e.i = 0 ;
                     if (e.i < e.m)  {
                        e.a = 1;  s.push(new StackElement(e)); // za povratek
                        e.a = 0; e.m = e.i ; e.n = e.n – 1;
                        s.push(new StackElement(e)) ; // za klic
                     }
                  }
                  break ;
          case 1: e.i ++ ;
                  if (e.i < e.m)  {
                        e.a = 1;  s.push(new StackElement(e)); // za povratek
                        e.a = 0; e.m = e.i ; e.n = e.n – 1;
                        s.push(new StackElement(e)) ; // za klic
                  }
                  break ;
        } // switch
    } while ( ! s.empty()) ;
}
```


Vračanje vrednosti pri spreminjanju v iteracijo s skladom

- vračanje vrednosti
  - s pomočjo sklada
  - s spremenljivko
- primer: izračun Fibonaccijevih števil

- prenos po referenci

```
public static  int fibR(int n)
{
   if  (n == 0)  return 0 ;
   else if (n == 1) return 1 ;
   else return fibR(n-1)+fibR(n-2);  // je to repna rekurzija?
}


public static int fibRekv(int n)
{
   if  (n == 0) return 0  ;
   else if (n == 1) return 1 ;
   else {
     int t = fibRekv(n-1) ;
     return t + fibRekv(n-2);
  }
}
```



```
class  StackElement {
   int n, t, a ;
   StackElement() {}
   StackElement(StackElement e) {
        n = e.n;  t = e.t;  a = e.a;
   }
}
```

```
public static int fibRsklad(int n) {
    Stack s = new StackArray();
    StackElement e = new StackElement();
    e.a = 0;  e.n = n; s.push(e);
    int vrni ;
    do {
        e = (StackElement) s.top() ; s.pop() ;
        switch (e.a) {
          case 0: if (e.n == 0)  vrni = 0 ;
                  else if (e.n == 1)  vrni = 1 ;
                  else {
                      e.a = 1;  s.push(new StackElement(e)) ;
                      e.a = 0;  e.n = e.n – 1; s.push(new StackElement(e));
                  }
                  break ;
          case 1:  e.t = vrni ;
                   e.a = 2;  s.push(new StackElement(e)) ;
                   e.a = 0;  e.n = e.n – 2;  s.push(new StackElement(e)) ;
                   break ;
          case 2:  vrni=vrni + e.t;
                   break ;
        } // switch
    } while ( ! s.empty()) ;
    return vrni ;
}
```


Kompleksnost algoritmov

- časovna kompleksnost
- prostorska kompleksnost
- velikostni red kompleksnosti
  - asimptotična zgornja meja O
  - spodnja meja Ω
  - stroga meja θ

Zgornja meja časovne kompleksnosti

- Definicija

$$T(n) = O(g(n)) \leftrightarrow \exists c, n_0 > 0 : n > n_0 \rightarrow c \cdot g(n) \geq T(n) \geq 0$$

- pri dovolj velikem n je kompleksnost našega programa navzgor omejena s funkcijo g(n)
- poljubna konstanta c

Računanje z O

- eliminacija konstante $c > 0 \rightarrow O(c \cdot f(n)) = O(f(n))$
- vsota $O(f(n)) + O(g(n)) = O(\max(f(n), g(n)))$
- prevladujoča funkcija $\forall n > n_0 : f(n) > g(n) \rightarrow O(f(n)) + O(g(n)) = O(f(n))$
- produkt $O(f(n)) \cdot O(g(n)) = O(f(n)g(n))$
- tranzitivnost $f(n) = O(g(n)), g(n) = O(h(n)) \rightarrow f(n) = O(h(n))$
- refleksivnost $f(n) = O(f(n))$

Določi asimptotično zgornjo mejo naslednjih funkcij

- $17n^3 + 93n^2 + n$
- $2n^2 + n \log n$
- $3n + n \log n$
- $2^n + 739 n^9$
- $(n+\sqrt{n})(\log n + n + 3)$

Pogoste kompleksnosti

$$\log n, n, n \log n, n^2, n^3, n^4, 2^n, n!, n^n$$

1. $a>b>0, c>0 \rightarrow n^a > c\, n^b$
2. $a>0, b>1, c>0 \rightarrow n^a > c \log_b^n$
3. $a>1, b>0, c>0 \rightarrow a^n > c\, n^b$
4. $a>1, c>0 \rightarrow n! > c\, a^n$
5. $c>0 \rightarrow n \log n > c \log n!$

Določanje kompleksnosti v programu - pravila

- določimo parametre kompleksnosti in časovno enoto
- pri zaporedju ukazov seštevamo zahtevnosti
- pri pogojih štejemo kompleksnost izračuna pogoja in maksimum vseh možnih izbir
- pri zankah seštejemo kompleksnost izračuna pogoja in enkratne izvedbe zanke ter pomnožimo s številom izvajanja zanke
- pri rekurziji izrazimo zahtevnost kot rekurzivno funkcijo

Določi asimptotično kompleksnost

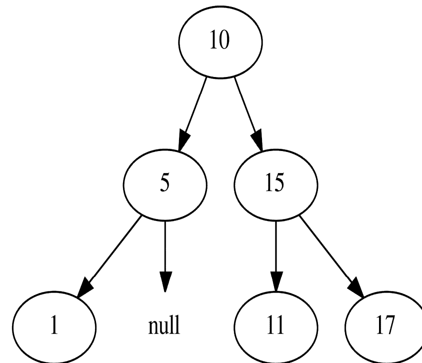
```
for (i=1; i <= n ; i++)
   for (j=1 ; j <= n ; j++)
      for (k=1 ; k <= n ; k++)
         if  (i +j+k<a)
            G[i][j] = A[i][j]+B[i][k]*C[k][j];
```

```
int i = n ;
int r =0 ;
while (i >1) {
    r = r+1 ;
    i = i / 2 ;
}
```



```
public static void loopRek(int m, int n)
{
    if (n == 1)
      System.out.println("+") ;
    else
      for (int i=0; i < m ; i++)
        loopRek(m, n-1) ;
}
```

```
public static void infix(Node p)
{
   if (p != null)  {
      infix(p.left) ;
      System.out.print(p.key) ;
      infix(p.right) ;
   }
}
```



Kaj je parameter kompleksnosti?


```
max = a[1] ;
for (i=2 ; i <= n ; i++)
   if (max < a[i])
      max = a[i] ;
System.out.print(max) ;
```

```
void p(int n, int m) {
   int i,j,k ;
   if (n > 0) {
     for (i=0 ; i < m ; i++)
       for (j=0 ; j < m ; j++)
         if (i < j – a)
           for (k=0 ; k < m ; k++)
             System.out.println(i+j*k) ;
     p(n/m, m) ;
   }
}
```


Za dani program so bili izmerjeni naslednji časi izvajanja za različne velikosti vhodnih podatkov:

velikost podatkov	5	10	15	30
čas	500	501	502	509

Katera funkcija najbolj ustreza časovni zahtevnosti tega programa v odvisnosti od vhodnih podatkov?

a. log(n)
b. n
c. n·log(n)
d. n^2
e. 2^n

Odgovor utemelji in predvidi koliko časa bo program potreboval za velikost podatkov 100.

Za dani program so bili izmerjeni naslednji časi izvajanja za različne velikosti vhodnih podatkov:

velikost podatkov	5	10	15	100
čas	1050	9800	33550	10.000.000

Katera funkcija najbolj ustreza časovni zahtevnosti tega programa v odvisnosti od vhodnih podatkov?

a. log(n)
b. n
c. n·log(n)
d. n^2
e. n^3
f. 2^n

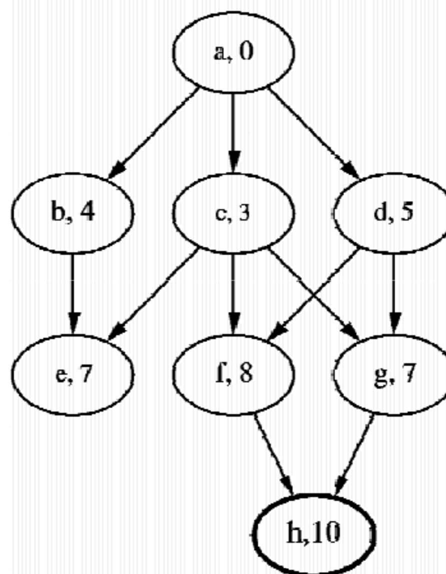
Odgovor utemelji in predvidi koliko časa bo program potreboval za velikost podatkov 1000.

Si predstavljate razlike med kompleksnostmi?

$\log_{10}(n)$	**√n**	**n**	**$n \cdot \log_{10}(n)$**	**n^2**	**2^n**
1	3	10	10	100	1024
2	10	100	200	10.000	$1.25 \cdot 10^{30}$
3	31	1.000	3.000	1.000.000	10^{301}
4	100	10.000	40.000	100.000.000	$2 \cdot 10^{3.010}$
5	316	100.000	50.000	10^{10}	$10^{30.103}$
6	1.000	1.000.000	6.000.000	10^{12}	$10^{301.030}$

Metode načrtovanja algoritmov

- enoten pogled na različne metode načrtovanja algoritmov
- Prostor stanj: $\mathbf{S} = \{S; S_Z \xrightarrow{*} S\}$
- kvaliteta stanja: $q(S)$
- končno stanje: $S_0 = \arg\max_{S \in \mathbf{S}} q(S)$
- različne strategije preiskovanja



Strategije preiskovanja

- deli in vladaj
- iskanje optimalnih rešitev
- iskanje približnih rešitev
- stohastični preiskovalni algoritmi

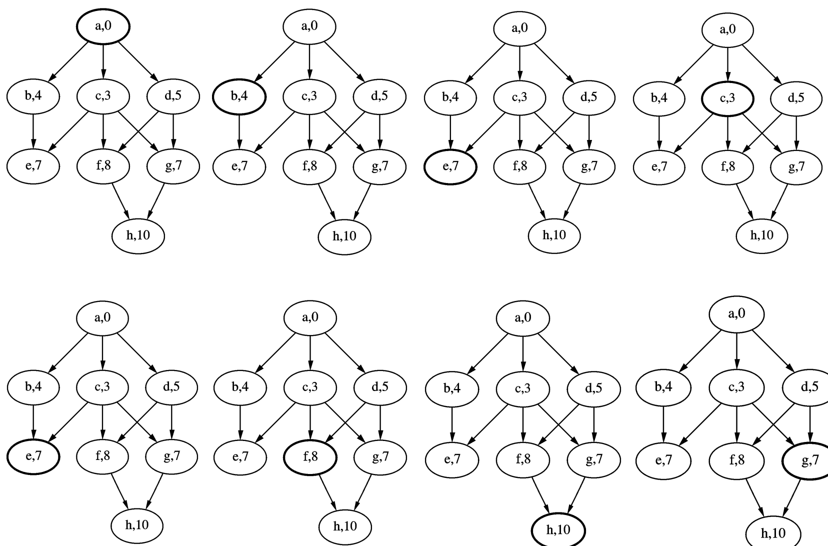
Iskanje optimalnih rešitev

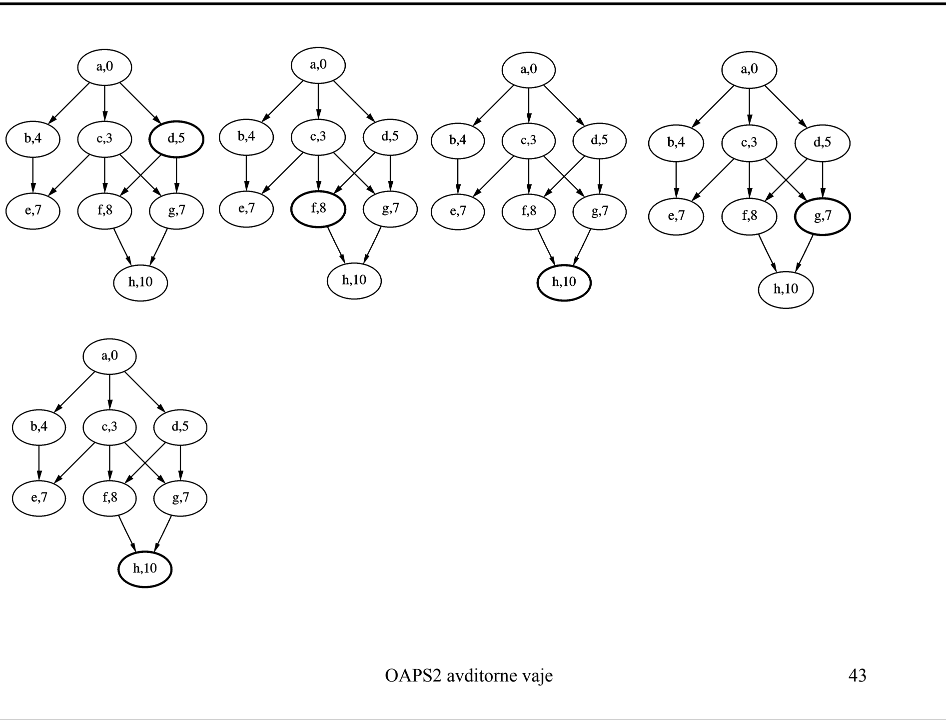
- izčrpno preiskovanje
- omejeno izčrpno preiskovanje
  – oceni q(S) in max(S)
  – v širino, v globino,
- metoda najprej najboljši
  – A*, IDA*, RBFS, ...
- dinamično programiranje

Izčrpno preiskovanje

- v globino
- v širino
- iterativno poglabljanje

Izčrpno preiskovanje v globino

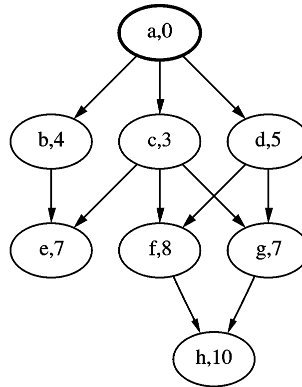




Izčrpno preiskovanje v širino

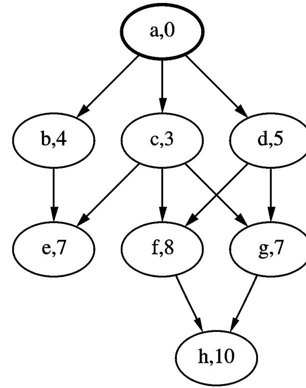
- vrsta stanj:

a
b,c,d
c,d,e
d,e,e,f,g
e,e,f,g,f,g
e,f,g,f,g
f,g,f,g
g,f,g,h
f,g,h,h
g,h,h,h,
h,h,h,h
h,h,h
h,h
h



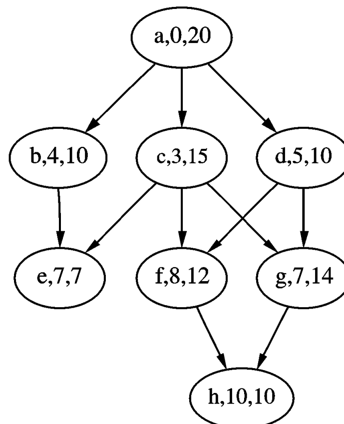
Iterativno poglabljanje

- vrstni red iskanja
- d=1: a
- d=2:a,b,c,d
- d=3:a,b,e,c,e,f,g,d,f,g
- d=4:a,b,e,c,e,f,h,g,h,d,f,h,g,h



Omejeno izčrpno preiskovanje (razveji in omeji)

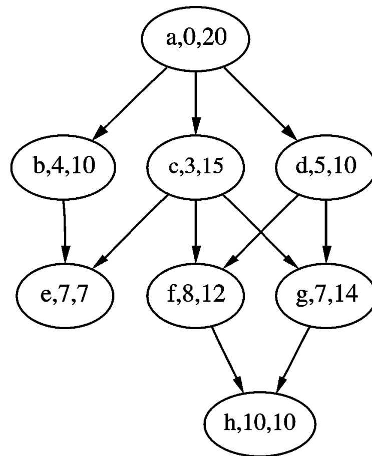
- oceni q(S) in max(S)
- v širino,
- v globino,



Omejeno izčrpno preiskovanje v globino

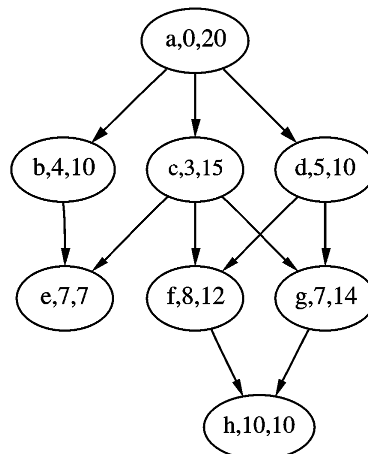
vrstni red preiskovanja:

a,b,e,c,e,f,h,g,h,d



Omejeno izčrpno preiskovanje v širino

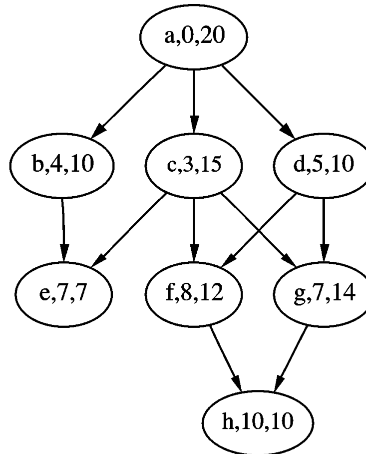
- vrsta stanj:
- a
- b,c,d
- c,d,e
- d,e,e,f,g
- e,e,f,g,f,g
- e,f,g,f,g
- f,g,f,g
- g,f,g,h
- f,g,h,h
- g,h,h,h,
- h,h,h,h
- h,h,h
- h,h
- h



Iskanje najprej najboljši

- A*, IDA*, RBFS, ...
- stanja prioritetne vrste:

a
c,b,d
g,f,b,d,e
f,b,d,h,e
b,d,h,h,e
d,h,h,e
g,f,h,h,e
f,h,h,h,e
h,h,h,h,e
h,h,h,e



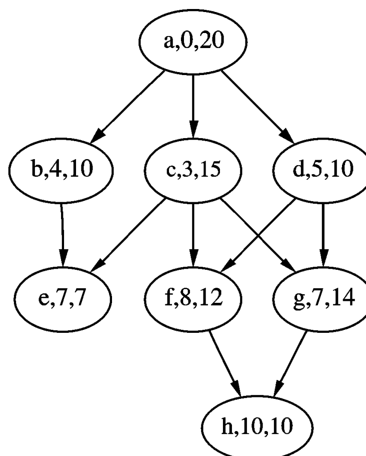
Iskanje približnih rešitev

- požrešno: vedno po najbolj obetavni poti
- v snopu: shranimo več najbolj obetavnih
- lokalna optimizacija
  - v prostoru rešitev
  - izboljšujemo rešitev z lokalnimi operatorji
- gradientno iskanje: v zveznem prostoru v smeri gradienta
- stohastični algoritmi

Požrešno iskanje

- vedno po najbolj obetavni poti:

a,c,g,h



Iskanje v snopu

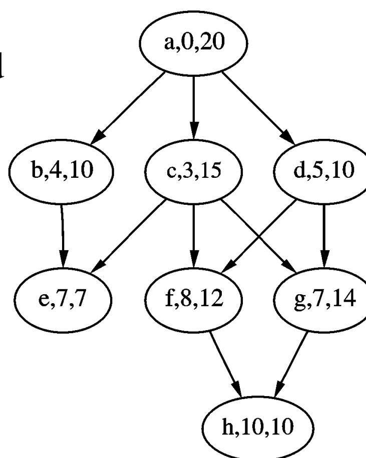
- na vsakem koraku izmed naslednikov izberemo m najbolj obetavnih
- za m=2:

a

c,d

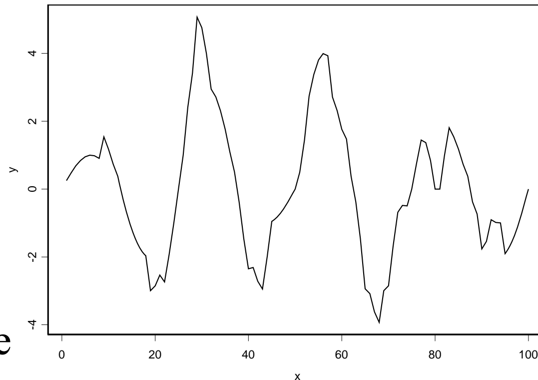
g,g

h



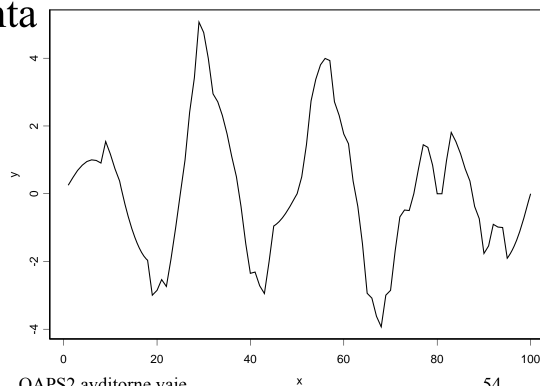
Lokalna optimizacija

- v prostoru rešitev
- izboljšujemo rešitev z lokalnimi operatorji- premikamo se le v soseščini



Gradientno iskanje

- požrešno iskanje v zveznem prostoru
- v smeri gradienta



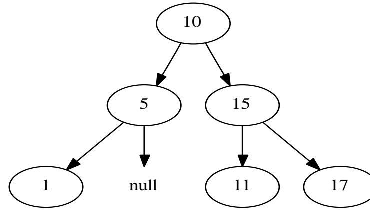
Stohastični preiskovalni algoritmi

- simulirano ohlajanje
  - ideja iz fizike
  - najprej dovolimo velike skoke, nato vse bolj le lokalno optimizacijo
- genetski algoritmi
  - ideja iz evolucije
  - populacija rešitev
  - uspešne rešitve kombiniramo (križanje)
  - dovolimo tudi mutacije
  - iz najboljših tvorimo novo populacijo

Dinamično programiranje

- postopno od spodaj navzgor gradimo optimalno rešitev

Optimalna binarna iskalna drevesa



- binarno drevo
- iskalno drevo
- urejenost elementov
- optimalno drevo: če iščemo vedno iste elemente in želimo čim krajši dostopni čas
- primer: ključne besede programskega jezika do, if, while, {, }
- večkrat iskane (bolj verjetne) želimo čimbližje korena drevesa

Opis problema

- ključi $k_1 < k_2 < ... < k_n$
- verjetnost iskanja ključa k_i je p_i
- verjetnost iskanja elementa med k_j in k_{j+1} je q_j
- verjetnost iskanja elementa manjšega od k_1 je q_0
- verjetnost iskanja elementa večjega od k_n je q_n
- T_{ij} vsebuje ključe od k_{i+1} do k_j
- celo drevo je T_{0n}
- verjetnost (teža) poddrevesa T_{ij} je w_{ij}

Povprečni dostopni čas

- optimalno drevo ima najmanjši povprečni dostopni čas

$$t(T)=\sum_{i=1}^{n} p_i h_i + \sum_{j=0}^{n} q_j h_j^{'}$$

Izračun

- teža poddrevesa $w_{i,j}=\sum_{k=i+1}^{j} p_k + \sum_{k=i}^{j} q_k$
- vsako optimalno drevo je sestavljeno iz optimalnih poddreves

$$\min t(T_{i,j}) = \min_{k\in[i+1,j]} (\min t(T_{i,k-1}) + w_{i,j} + \min t(T_{k,j}))$$

$$t(T_{i,i})=0$$

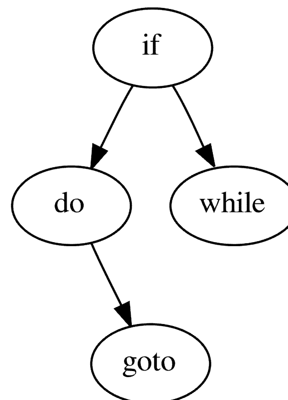
- gradimo od spodaj navzgor po principu dinamičnega programiranja

Primer

- Za dane verjetnosti elementov in pogreškov določite urejenost, izračunajte povprečni dostopni čas optimalnega iskalnega drevesa in ga narišite.

i	0	1	2	3	4
		do	goto	if	while
p_i		0.20	0.05	0.35	0.25
q_i	0.01	0.02	0.03	0.04	0.05

j-i \ *i*	0	1	2	3	4
0	0.01 0 null	0.02 0 null	0.03 0 null	0.04 0 null	0.05 0 null
1	0.23 0.23 1	0.10 0.10 2	0.42 0.42 3	0.34 0.34 4	
2	0.31 0.41 1	0.49 0.59 3	0.72 1.06 3		
3	0.70 1.11 3	0.79 1.23 3			
4	1.00 1.75 3				



Algoritem CYK

- CYK: Cocke – Younger – Kasami
- konteksno neodvisna gramatika KNG
- KNG={V,T,S,P}
- ali velja $x \in L$ (oziroma $S \Rightarrow x$)?
- primer: ali je program pravilen glede na dano BNF gramatiko?
- normalna oblika po Chomskem

CYK - ideja

- za vsak i, j, A ugotoviti ali $A \rightarrow x_{ij}$, kjer je x_{ij} podniz dolžine j, ki se začne na poziciji i
  če $S \rightarrow x_{1n,}$ potem $x \in L$
- induktivni dokaz:
  za dolžino 1: obstajati mora produkcija
  za dolžino n: obstajati mora k tako, da lahko sestavimo x_{ik} in $x_{i+k,j-k}$; k in j-k sta manjša od n, torej smo ju že izračunali
- dinamično programiranje

CYK - primer

- $S \rightarrow A\ B$
  $A \rightarrow BC \mid a$
  $B \rightarrow CC \mid b$
  $C \rightarrow a$
- ali $S \rightarrow aaab$?

	a	a	a	b
1	A,C	A,C	A,C	B
2	B	B	S	
3	S,A	/		
4	S			

CYK nalogi

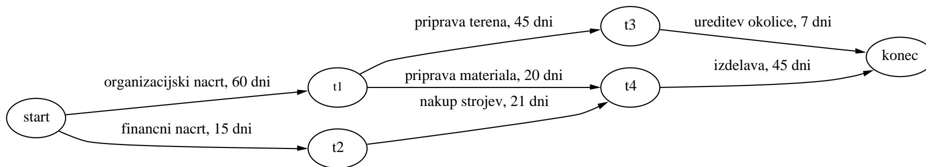
- $S \rightarrow P\ N \mid drugo$
  $P \rightarrow I\ E$
  $I \rightarrow if$
  $E \rightarrow izraz$
  $N \rightarrow T\ S$
  $T \rightarrow then$

- $S \rightarrow A\ C \mid B\ D \mid A\ E$
  $C \rightarrow B\ B$
  $D \rightarrow A\ A$
  $E \rightarrow B\ A \mid A\ B$
  $A \rightarrow a \mid A\ E \mid E\ A \mid B\ D$
  $B \rightarrow b \mid B\ E \mid E\ B \mid A\ C$
- ali $S \rightarrow baabba$

- ali $S \rightarrow$ if izraz then if izraz then drugo

Kritična pot

- primer: mrežni diagram projekta
  - delno urejen seznam akcij predstavimo z acikličnim usmerjenim grafom
  - vozlišča – časovni mejniki
  - povezave – akcije s časi izvajanj



Izračun kritične poti

- kritična pot je najdaljša pot v grafu, saj določa trajanje projekta
- <a,b> je povezava s časom eval(a,b)

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

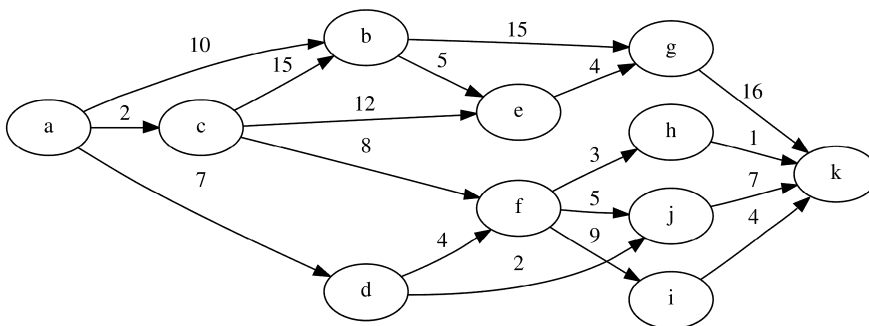
$$t(a,c) = \max_{<a,b>\in E} [eval(a,b) + t(b,c)]$$

- naivna rekurzivna rešitev preišče vse mogoče poti v grafu – kar je lahko pomeni eksponentno zahtevnost
- dinamično programiranje

Algoritem za izračun kritične poti

- dinamično programiranje
- za vsako vozlišče hranimo maksimalen čas do njega, predhodnika na maksimalni poti in število še ne pregledanih vhodnih poti
- od začetka proti koncu
- vozlišča, ki smo do njih pregledali že vse poti lahko razvijemo (pregledamo naslednjike)

Kritična pot: primer



Kritična pot: rešitev primera

- za vsako vozlišče shranjujemo (predhodno vozlišče, maksimalen čas do vozlišča, število še ne pregledanih vhodov)
- vrsta preiskovanih vozlišč: a,c,d,b,f,e,h,i,j,g,k

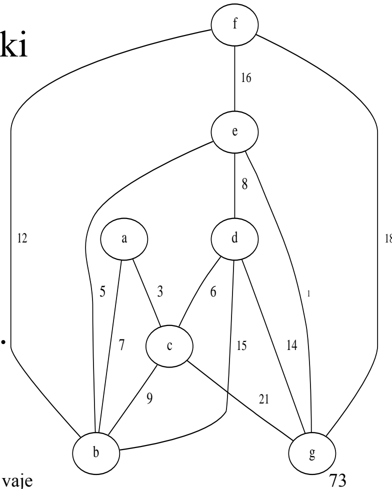
a: (-,0,0)
b: (-,0,2), (a,10,1), (c,17,0)
c: (-,0,1), (a,2,0)
d: (-,0,1), (a,7,0)
e: (-,0,2), (c,14,1), (b,22,0)
f: (-,0,2), (c,10,1), (d,11,0)
g: (-,0,2), (b,32,1), (b,32,0)
h: (-,0,1), (f,14,0)
i: (-,0,1), (f,20,0)
j: (-,0,2), (d,9,1), (f,16,0)
k: (-,0,4), (h,15,3), (i,24,2), (i,24,1), (g,48,0)
Rešitev: a-c-b-g-k v času 48

Požrešni algoritmi

- vedno v najbolj obetavni smeri glede na nek optimizacijski kriterij

Minimalno vpeto drevo

- vpeto drevo grafa je drevo, ki povezuje vsa vozlišča
- minimalno vpeto drevo je vpeto drevo, katerega vsota cen povezav je najmanjša
- primer: komunikacijska mreža, elektro napeljava, ...



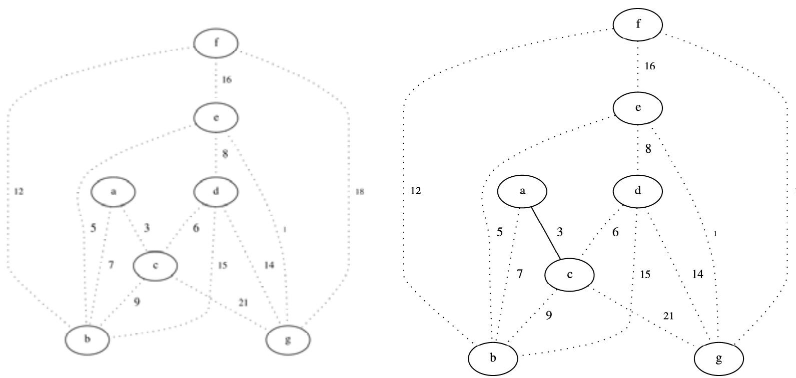
Iskanje minimalnega vpetega drevesa

- požrešni metodi:Primov in Kruskalov algoritem
  - m = število povezav v grafu
  - n =število vozlišč v grafu
- Primov algoritem
  - O(m log n)
- Kruskalov algoritem
  - O(m log m)

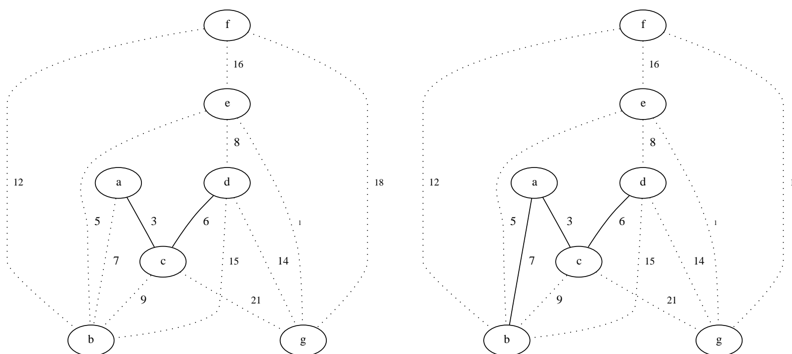
Primov algoritem

```
// naj bo V = {1,2, … n} univerzalna množica vseh vozlišč
public SetOfEdges prim(Graph  g) {
    SetOfVertex U = new SetOfVertex(); //  že pregledana vozlišča
    Vertex u, v  ;
     SetOfEdges T = new SetOfEdges() ;
     T = {}; U = {1} ;
     while (U != V)  {
          naj bo (u, v) najcenejša povezava tako, da u ∈ U in v ∈V-U
          T = T ∪ { (u,v) } ;
          U = U ∪ { v }
     }
}
```

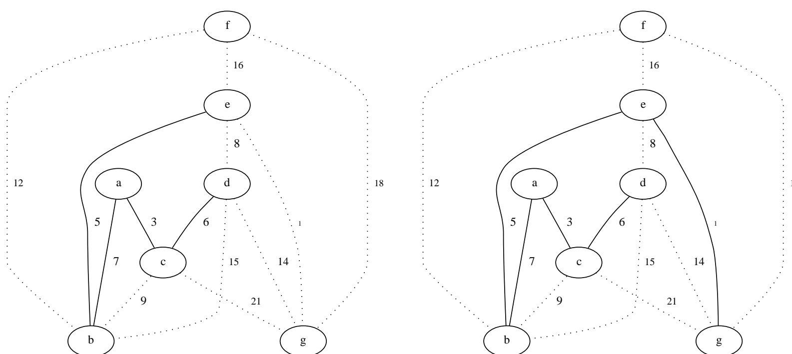

Simulacija Primovega algoritma 1/4



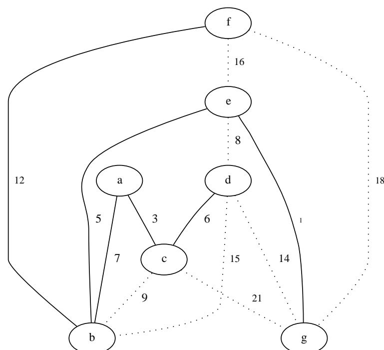
Simulacija Primovega algoritma 2/4



Simulacija Primovega algoritma 3/4



Simulacija Primovega algoritma 4/4

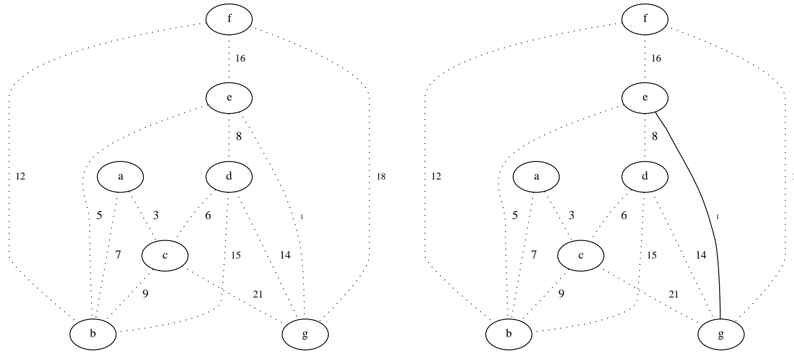


- začnemo lahko v poljubni začetni točki
- požrešno dodajamo najcenejše povezave iz preledanega v še ne pregledani del

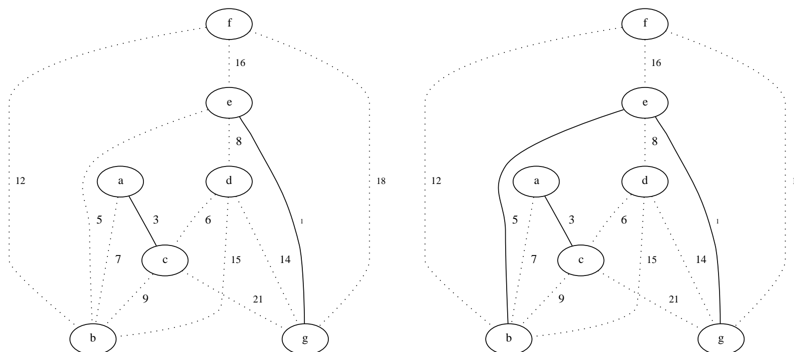
Kruskalov algoritem

```
public SetOfEdges kruskal(Graph g) {
    Vertex u, v  ;
     SetOfEdges T = new SetOfEdges() ;
     for (vsako vozlišče v∈G)
          makeSet(v) ;
     sortiraj povezave e ∈G naraščujoče glede na ceno
     for (vse povezave (u,v) ∈G)
          if (findSet(u) != findSet(v))  {
               T = T ∪ { (u,v) } ;
               union(SetOf(u), SetOf(v)) ;
          }
}
```

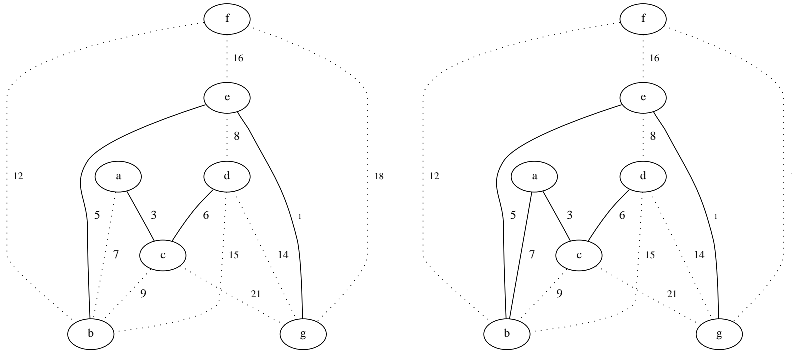
Simulacija Kruskalovega aloritma 1/4



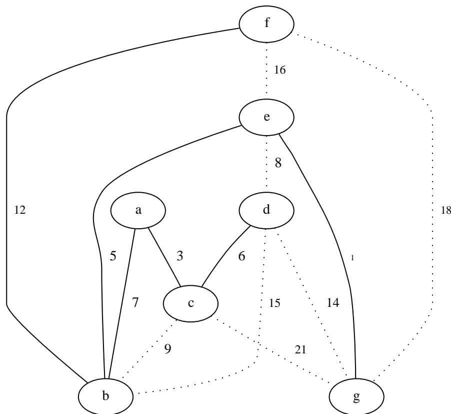
Simulacija Kruskalovega aloritma 2/4



Simulacija Kruskalovega aloritma 3/4



Simulacija Kruskalovega aloritma 4/4



- požrešno povezujemo disjunktne množice z najcenejšimi povezavami

Iskanje najkrajših poti v usmerjenem grafu

- najkrajše poti od izbrane točke do vseh drugih
- dolžina povezave (a,b) naj bo c(a,b)
- rekurzivna definicija $t(a,a) = 0$

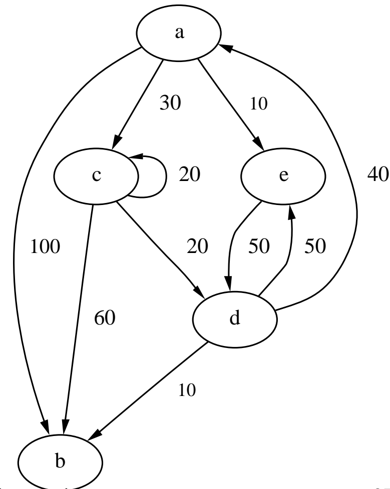
$$t(a,c) = \min_{(a,b)\in E}[c(a,b) + t(b,c)]$$

- za nenegativne cene Dijkstrin algoritem
- požrešno delovanje, uporablja prioritetno vrsto
- če algoritem spremenimo, da uporablja največje razdalje, ga lahko uporabimo za izračun kritične poti

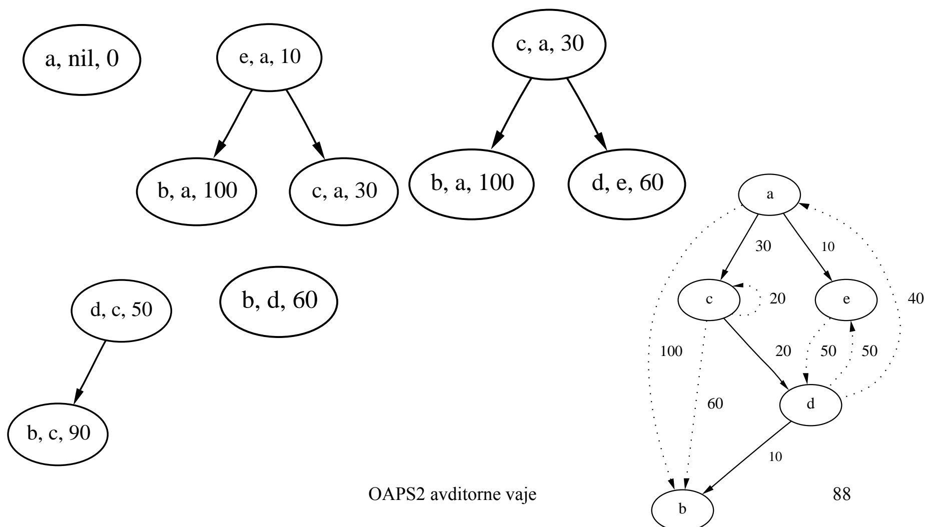

```
public void  Dijkstra(Graph g, Vertex a) {
  Vertex v, w ;
  PriorityQueue q = new HeapPos() ;

        for (each vertex v ∈ G)
            v.visited = false ;
        a.parent = null ; a.distance = 0 ; a.visited = true ;
        q.makenull() ;
        q.insert(a) ;
        while  (! q.empty()) {
                v = q.deleteMin() ;
                for (each succesor w of vertex v)
                        if (!  w.visited) {
                                w.parent = v ; w.visited = true ;
                                w.distance = v.distance + c(v,w) ;
                                q.insert(w) ;
                        }
                        else if  (v.distance + c(v, w) < w.distance) {
                                w.parent := v :
                                q.decreaseKey(w, v.distance + c(v, w)) :
                        }
                }
        }
```

Simulacija algoritma Dijkstra 1/2

- izpišimo zaporedje stanj prioritetne vrste, implementirane s kopico
- izpisujemo (vozlišče, parent,distance)
- vozlišča, ki jih vzamemo iz prioritetne vrste, vsebujejo rezultat



Simulacija algoritma Dijkstra 2/2



Dokazovanje pravilnosti programov

- preverjanje pravilnosti
  - branje in razumevanje
  - testiranje na izbrani množici testnih podatkov
  - formalno dokazovanje
- program f: $X \rightarrow Z$ preslika vhodne podatke $<x_1, x_2, \ldots x_n> \in X$ v izhodne $<z_1, z_2, \ldots z_m> \in Z$
- začetni pogoj $\Phi(x_1, x_2, \ldots x_n)$ – kakšni so lahko podatki
- izhodni pogoj $\Psi(z_1, z_2, \ldots z_m, x_1, x_2, \ldots x_n)$ – kakšen mora biti rezultat

Pravilnost programa

- **parcialna pravilnost**: če se za vse vhodne podatke, ki izpolnjujejo Φ ustavi, izhodni podatki pa izpolnjujejo Ψ
- **totalna pravilnost**: če je parcialno pravilen in se za vse vhodne podatke, ki izpolnjujejo Φ ustavi po **končnem** številu korakov

Postopek dokazovanja za sintaktične elemente 1/3

- prireditev

```
// P(izraz)
y = izraz ;
// P(y)
```

- izbira

```
// P(y)
if (Pogoj(y))
    // P(y) ∧ Pogoj(y)}
    St ;
  else
     // P(y) ∧ ¬Pogoj(y)
     Se ;
```


Postopek dokazovanja za sintaktične elemente 2/3

- zanka

```
// P1(y)
while (Pogoj(y))  {
  // za i-to izvajanje Pi(y) ∧ Pogoj(y)}
  S ;
 // Pi+1(y)
}
// Pk(y) ∧ ¬Pogoj(y)
```

pogoj $P_{i+1}(y)$ izpeljemo iz $P_i(y) \wedge Pogoj(y)$

$P_k(y)$ je odvisen od števila izvajanj zanke

Postopek dokazovanja za sintaktične elemente 3/3

- zaporedje

// $P_0(y)$

S_1; S_2; ... S_k;

// $P_k(y)$

pri čemer $P_{i+1}(y)$ izpeljemo iz $P_i(y)$

- združitev več poti izvajanja

if ... St // $P_t(y)$}

else Se // $P_e(y)$}

//$P_t(y) \vee P_e(y)$

Parcialna pravilnost

- največ težav pri zankah, saj je $P_k(y)$ odvisen od števila izvajanj, ki ga ne poznamo vnaprej
- definiramo zančno invarianto I(y), ki je vsebovana v $P_i(y)$ (pred, med in po izvajanju)
- I(y) mora biti zadosti močna, da lahko izpeljemo $\Psi(y)$, vendar ne premočna, da jo lahko dokažemo

// I(y)

while Pogoj(y) **{**

// $I(y) \wedge Pogoj(y)$

S ;

// I(y) }

}

// $I(y) \wedge \neg Pogoj(y)$}

Totalna pravilnost

- dodatno zahtevamo ustavitev v končnem številu korakov
- težava le pri zankah, zato definiramo
  - dobro utemeljeno množico D (delna urejenost brez neskončnih padajočih zaporedij)
  - zančno spremenljivko $t \in D$
- dokažemo, da se t v zanki manjša

```
// t ∈ D
while (Pogoj(y)) {
  // t ∈ D ∧ Pogoj(y)}
  S ;
 // t' ∈ D ∧ t' < t
}
```


Primer dokazovanja pravilnosti – izračun potence

```
public static int pot(int a, int n)
{
    int p = 1 ;
    int i = 0 ;
    while (i != n)  {
        i  = i + 1 ;
        p= a*p ;
    }
    return  p ;
}
```

Primer dokazovanja pravilnosti – pogoji in invarianta

- začetni pogoj $\Phi(a,n) \equiv a > 0 \wedge n \geq 0$
- končni pogoj $\Psi(a,n,p) \equiv p=a^n$
- parcialna pravilnost
  - zančna invarianta $I(a,p,i) \equiv p=a^i$
- totalna pravilnost
  - dobro utemeljena množicaa $D = N_0$
  - zančna spremenljivka $t = n-i$

parcialna pravilnost

```
public static int pot(int a, int n)
{
    // Φ(a,n) ≡ a >0 ∧ n ≥ 0
     int p = 1 ;
     int i=0 ;
     // a >0 ∧ n ≥ 0 ∧ p=1 ∧ i=0 ⇒ I(a,p,i) ≡ p=a^i (1=a^0) velja
      while (i != n) {
          // a >0 ∧ n ≥ 0 ∧ i <n ∧ p=a^i
          i  = i + 1 ;
          // a >0 ∧ n ≥ 0 ∧ i-1 <n ∧ p=a^(i-1)
          p= a*p ;
          // a >0 ∧ n ≥ 0 ∧ i-1 <n ∧ p = a*a^(i-1) =  a^i
      }
      // a >0 ∧ n ≥ 0 ∧ ¬ i != n ∧ p = a^i ⇒ I = n ∧ p =  a^n
      return  p ;
      // p = a^n ⇒ Ψ(a,n,pot)
}
```

totalna pravilnost

(parcialno smo že dokazali)

```
public static int pot(int a, int n)
{
    int p = 1 ;
    int i = 0 ;
    // D = N0 ∧ t = n-i ∧ n ≥0 ∧ i = 0 ⇒ t ∈ D
    while (i != n) {
        // t  = n-i ∈ D
        i  = i + 1 ;
        p= a*p ;
        // t'  = n - (i+1)  = n - i – 1 = t-1 < t
    }
    return p ;
}
```


Vaja dokazovanja pravilnosti

```
double sum(double a[], int n) {
    int s = 0.0 ;
    int i = 0 ;
    while (i < n)  {
        s = s + a[i] ;
        i  = i + 1 ;
    }
    return  s ;
}
```