Algoritmid ja andmestruktuurid

- Ahned (*greedy*) algoritmid
- Graafialgoritmid: kattev puu, lühim tee

Markide paigutamise probleem

- Meil on kasutada 10, 20 ja 50 sendised margid. Leida millises kombinatsioonis tuleks marke panna mingi summa maksmiseks, et markide arv oleks minimaalne. Millise algoritmiga saab sobilikku markide komplekti valida?
- Ahne algoritm - võta suurimaid münte niipalju kui saad ja siis sellele järgneva suurusega jne
- Aga kui on kasutada 10, 40 ja 50 sendised margid?
  - Ahne algoritm ei tööta!

Ahned algoritmid

Lahenduseni jõutakse tehes igal sammul hetke parim valik.

- Kõik valikud on lokaalselt optimaalsed
- Tulemusena jõutakse globaalselt optimaalse tulemuseni
- Otsingupuu mõttes liigutakse alla mööda ühte haru
  - Ei kasutata tagasivõtmist
  - Ei võrrelda teiste harudega
  - Ahne kriteerium on oraakel, kes teeb õige valiku

Ahned algoritmid

- Annavad efektiivse lahenduse mitmetele keerulisena näivatele probleemidele.
  annavad enamasti O(n) - O(n^2) algoritmi
- Ei leidu (usutavasti) enamusele keeruliseks peetavatele probleemidele
- Keerukana näiva probleemi korral:
  - tasub alustada ahne algoritmi olemasolu otsimisest
  - püüda leida kontranäidet, mida ahne algoritmi kandidaat ei suuda lahendada ja kui ei suuda sellist leida siis
  - tõestada, et algoritm jõuab alati lahenduseni

Ahne algoritmi tsükkel

1. Valik
   - järgmise elemendi valik hulgast
   - tehakse lähtuvalt lokaalsest optimaalsuskriteeriumist

   N: valitakse kõige suurem olemasolev münt
2. Sobivuse kontroll
   - kontrollitakse kas selle elemendi valimisel on võimalik jõuda lahenduseni

   N: kontrollitakse, et selle mündi lisamisega ei mindaks üle lõppsumma, kui minnakse, siis võetakse järjekorras järgmine
3. Lahenduseni jõudmise kontroll
   - kontrollitakse kas ollakse juba lahenduseni jõudnud

Lokaalne optimaalsus

Kriteeriumit nimetatakse *lokaalseks* kui otsuse tegemine ei nõua tagasivõtmist *(backtracking)* ja ammendavat otsingut.

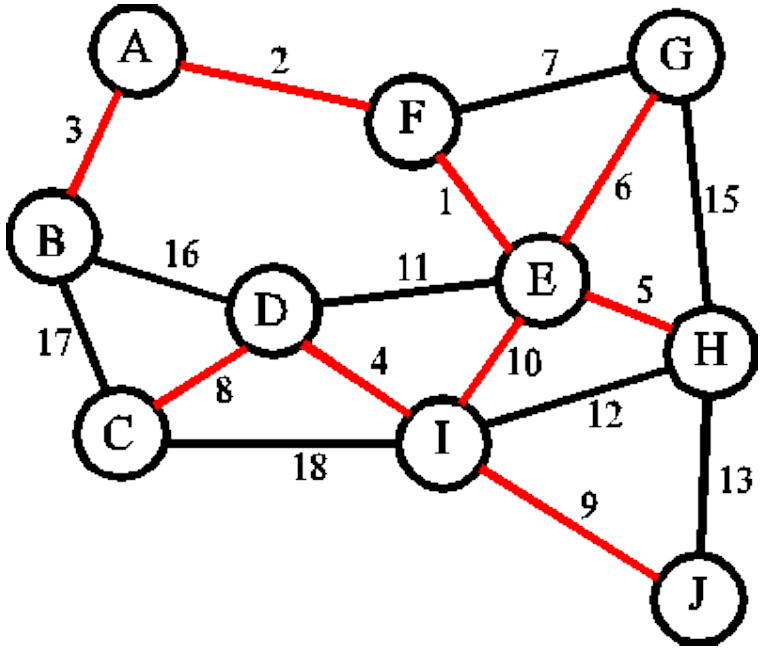
Lokaalne optimaalsus ei taga alati globaalset optimaalsust!

Iga probleemi ja valitud lokaalse optimaalsuse kriteeriumi kohta tuleb tõestada, et see tagab globaalse optimaalsuse!

Graafi minimaalne kattev puu (aluspuu, toes)

- Graafi G kattev puu (*spanning tree*) on servade hulk suurusega |V| -1, mis ühendab kõiki tippe
- Minimaalne kattev puu on kattev puu, mille kõigi servadega seotud kaalude summa on minimaalne

Minimaalse katva probleemi lahendus on oluline kaabelduse, torustike, raudteede jms planeerimisel



Minimaalse katva puu probleemi ahne lähenemine

while (ülesanne pole lahendatud)

{

  vali mingis mõttes lokaalselt parim serv

  kontrolli, et selle lisamine ei moodustaks tsüklit

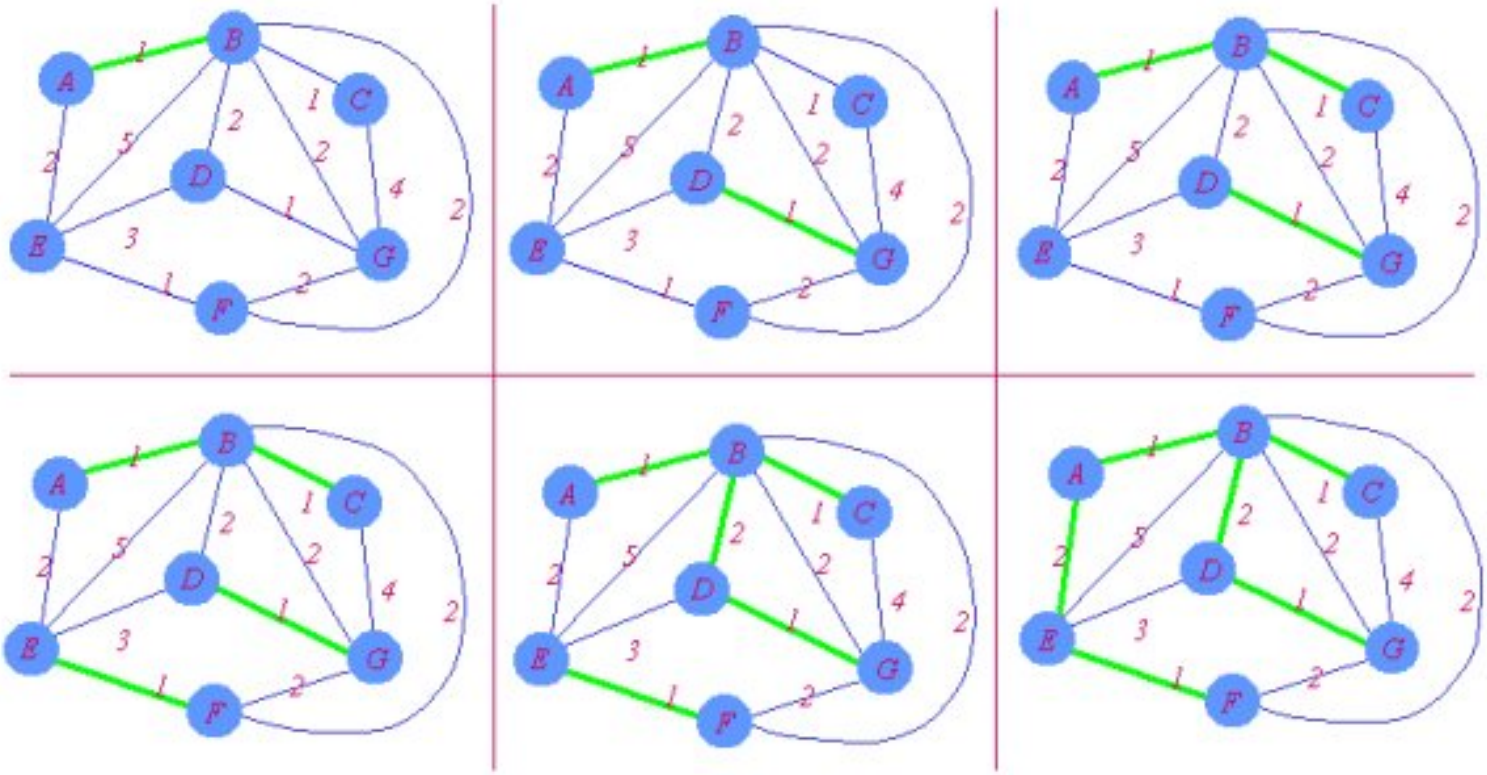
  kui tulemus on *kattev puu*, siis on ülesanne lahendatud

}

- mis on lokaalselt parim serv?
- kuidas kontrollida tsükli tekkimist?
- kuidas näidata, et tulemus on minimaalne kattev puu?

Kruskal’i algoritmi idee

- Ühendame servad, mis ei moodusta tsüklit, pikkuse järjekorras
- Tsükli tekkimise kontrolliks kasutame *mittelõikuvaid alamhulki*
- Animation



Kruskal'i algoritm

```
edges kruskal(int n, int m, Edges E, Vertices V)
   index i, j
   Set P, Q
   Edge e
   Sort the m edges in E by weight;  // or use priority queue
   F = ∅
   forall(v in V)
      MakeSet(v)
   while( number of edges in F is less than n-1)
      e = next edge (with the least weight) from sorted E
      (i, j) = e   // vertices connected by e
      P = Find(i); Q = Find(j)
      if (! Equal(P,Q))
         Union(P, Q)
         add e to F
  return F;
```

Kruskal'i algoritmi keerukus

n - tippude arv graafis

m - servade arv graafis

Servade sorteerimine	$W(m) \in O(m \lg m)$
Initsialiseerimine	$W(n) \in O(n)$
While loop	$W(m) \in O(m \lg n)$

$W(m, n) \in O(m \lg m)$

täielikus graafis:

$m = n(n - 1) \in O(n^2)$

$W(m, n) \in O(n^2 \lg n)$

Mittelõikuvad alamhulgad

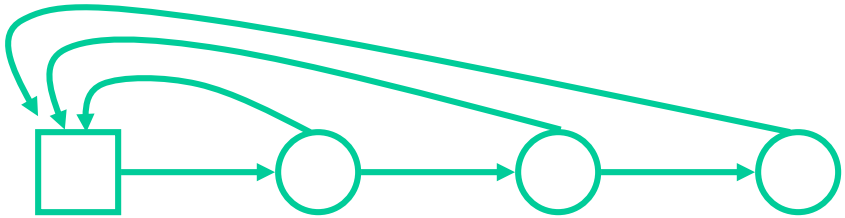
- Mittelõikuvad alamhulgad (*disjoint subsets*)
  - meil on fikseeritud hulk U elementidega X_i
  - U jaguneb mittelõikuvateks alamhulkadeks S1, S2 , S3 , … Sk
  - $Si \cap Sj = \emptyset$, kui $i \neq j$
  - $S1 \cup S2 \cup S3 \cup \ldots \cup Sk = U$

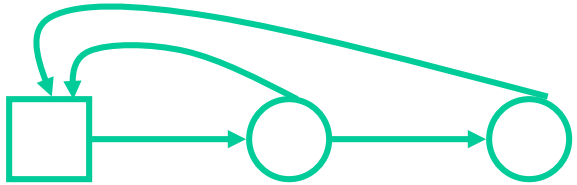
- Operatsioonid andmestruktuuril mittelõikuvad alamhulgad
  - **MakeSet(X):** Väljasta hulk, milles on ainult element **X**
  - **Union(S,T):** Väljasta **S** $\cup$ **T**, mis asendab hulki **S** ja **T**
  - **Find(X):** Leia alamhulk **S** nii et **X** $\in$ **S**

  !!! Pole operatsiooni, mis leiaks alamhulga kõik liikmed

Mittelõikuvate alamhulkade naiivne implementatsioon

- Kasutame iga hulga esitamiseks lingitud listi:

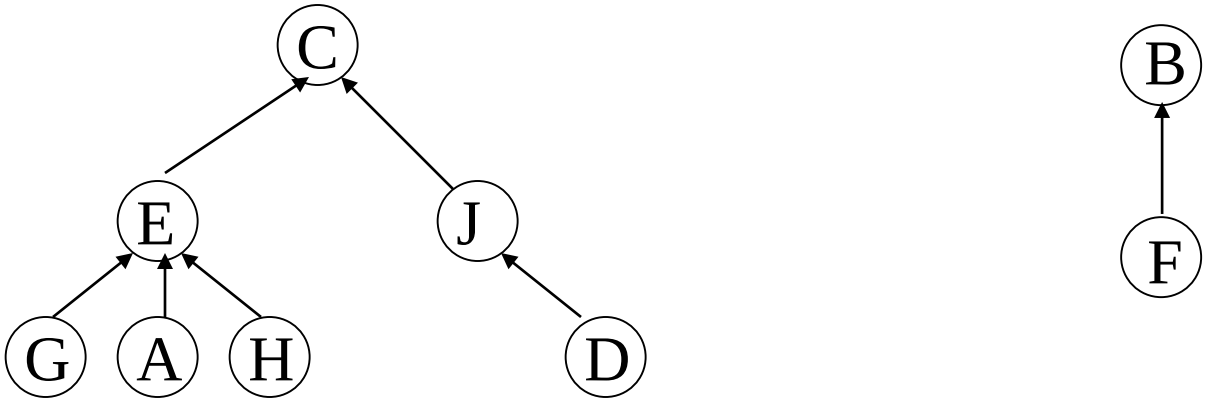




- MakeSet(): O(1) keerukus
- Find(): O(1) keerukus
- Union(A,B): “kopeerime” elemendid hulgast A hulka B: O(A) keerukus

Parem implementatsioon: Up-Tree

- Omadused
  - igal sõlmel on viit oma vanemale; tipus on see viit tühi
  - igal sõlmel võib olla piiramatu hulk järglasi
  - hulkasid tähistab juurtipp



Disjoint Subsets: C = {A,C,D,E,G,H,J}

B = {B,F}

Up-Tree: **MakeSet** ja **Find**

- **MakeSet(X):** loo üheelemendiline puu
  - keekukus O(1)
- **Find(X):** millisesse hulka element kuulub? mine mööda viitasid puu tipuni
  - Juur-element ongi hulga nimi, esindaja
  - keerukus O(puu-sügavus)

- Kahe elemendi samasse alamhulka kuulumise kontroll:
  - tee mõlemale Find ja vaata kas on sama hulk (juurtipp)

Up-Tree: **Union**

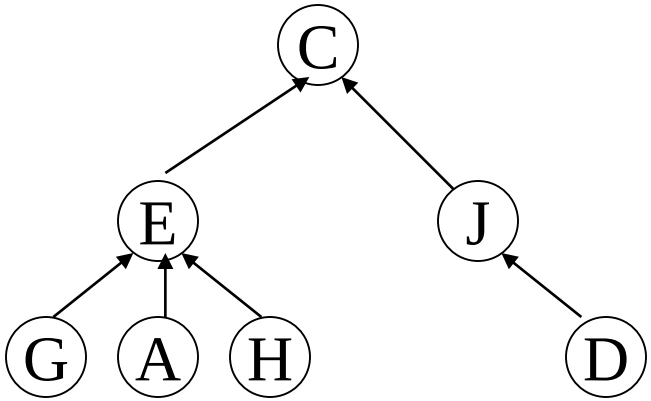
- **Union(S, T):** Paneme ühe hulga juure viitama teise hulga juurele.
  Kui paneme **S** juure viitama **T** juurele, siis lisame hulga **S** hulka **T**
- **Optimiseerimine**: Et vältida puu kasvamist lineaarseks listiks – lisame “väiksema” puu suuremasse
  - kõrguse järgi – halvima juhu optimiseerimine
  - suuruse (tippude arvu) järgi – keskmise juhu optimiseerimine

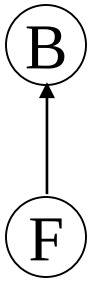
Up-Tree: **Union**

Balanseeritud ühendamise strateegia

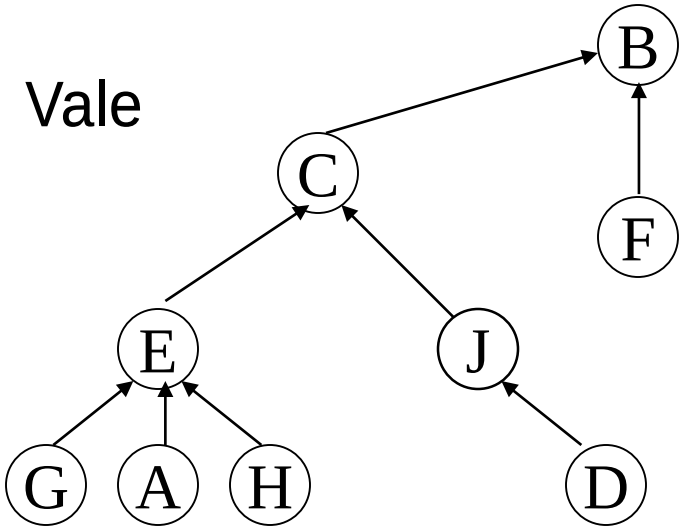
- Miks vältida puu sügavuse kasvu?
  - **Find** keerukus on proportsionaalne puu sügavusega
- Realisatsioon: Igal tipul on lisaks **Count** väärtus, puu juurel märgib see väärtus puus olevate tippude arvu (või puu sügavust)
- **Union** keerukus: **O(1)** kui **S** ja **T** on juurtipud, muidu Find(X) keerukus

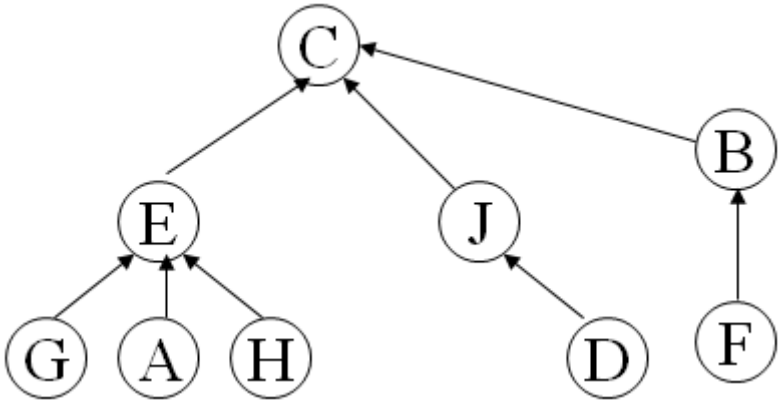
Up-Tree: **Union**





Vale





Up-Tree optimiseerimine

- **Find(X)** võtaks vähem aega madalas puus (põõsas)
- kasutame balanseeritud ühendamisstrateegiat kõrguse järgi
  → kindlustab kõrguse logaritmilise sõltuvuse tippude hulgast
- Kuna suvaline arv sõlmi võib omada sama vanemat, siis reorganiseerime puu ümber “põõsamaks”

Up-Tree optimiseerimine

- **Teekonna lühendamine: Find** operatsiooni käigus paneme kõik vahepealsed tipud viitama otse juurtipule.
- Kõik järgnevad **Find** operatsioonid võtavad nüüd vähem aega, kuna nad on tipule “lähemal”.
- Väike“probleem” kombineerituna hulkade võrdlemisega kõrguse järgi
  - võtame kõrgusi kui ebatäpseid hinnanguid

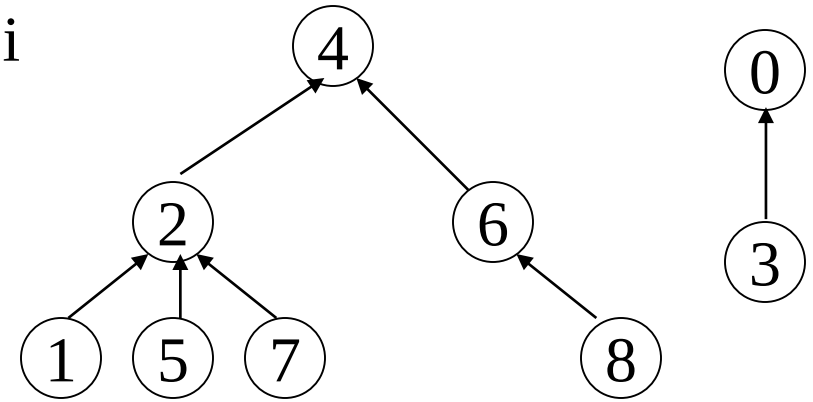
Up-Tree: **Massiivesitus**

- Tähistame elemendid on täisarvudega 0 kuni N
- Võime esitada Up-Tree massiivina suurusega N

a	-1	2	4	0	-1	2	4	2	6
	0	1	2	3	4	5	6	7	8

a[i] = *parent of* i

a[root] = -1



Mittelõikuvate alamhulkade näide

- Makeset (1 … 6)
- Union (2,3)
- Union (1,4)
- Union (2,6)
- Union (4,6)
- Find(5) = Find(6)?

Prim'i algoritmi idee

```
F = ∅;                                  // servade hulk
Y = {v1};                               // tippude hulk
while (pole lahendatud)
{ vali tipp hulgast V\Y mis on lähim Y-le;
  lisa tipp Y-le;
  lisa serv F-le;
  if (Y == V)
    lahendatud;
}
```

Animation

- Kuidas välditakse tsüklite tekkimist?

Prim'i ja Kruskal'i keerukus

n - tippude arv graafis

m - servade arv graafis

täielikus graafis $m = n(n - 1) \in O(n^2)$

hõredas graafis $m = O(n)$

- Prim keerukus sõltuvalt andmestruktuurist

  massiivdega $O(n^2)$,

  binaarkuhjaga $O(m \lg n)$ - täielikus graafis $O(n^2 \lg n)$,

  fibonacci kuhjaga $O(m + n \lg n)$ - täielikus graafis $O(n^2)$

- Kruskal

  $W(m, n) \in O(m \lg m)$

  täielikus graafis

  $W(m, n) \in O(n^2 \lg n)$

Prim'i algoritmi idee

```
F = ∅;                                  // servade hulk
Y = {v1};                               // tippude hulk
while (pole lahendatud)
{ vali tipp hulgast V\Y mis on lähim Y-le;
  lisa tipp Y-le;
  lisa serv F-le;
  if (Y == V)
    lahendatud;
}
```

Animation

- Kuidas välditakse tsüklite tekkimist?

Prim'i algoritmi idee

```
F = ∅;                                  // servade hulk
Y = {v1};                               // tippude hulk
while (pole lahendatud)
{ vali tipp hulgast V\Y mis on lähim Y-le;
  lisa tipp Y-le;
  lisa serv F-le;
  if (Y == V)
    lahendatud;
}
```

Kuidas esitada hulka Y, nii et oleks lihtne leida minimaalse kaugusega tippude paari (x, y), $x \in V \setminus Y$, $y \in Y$?

Prim'i algoritm - massiividega

Sisendiks on graaf naabrusmaatriksina

$$W[i][j] = \begin{cases} \texttt{serva kaal} & \text{kui } (v_i, v_j) \in E \\ \infty & \text{kui } (v_i, v_j) \notin E \\ 0 & \text{kui } i = j \end{cases}$$

Hoiame vahetulemusi kahes massiivis

nearest[i]	v_i-le lähim tipp Y-s
distance[i]	kaugus v_i ja *nearest*[i] vahel, kui $v_i \in Y$
	-1, kui $v_i \notin Y$

Prim'i algoritm

```
void prim (int n, const number W[][], set_of_edges& F)
{ index i, vnear;
  number min;
  edge e;
  index nearest[2...n];
  number distance[2...n];

  F = ∅;
  for (i=2; i <= n; i++)      // massiivide algväärtustamine
  { nearest[i] = 1;
    distance[i] = W[1][i];
  }
```

Prim'i algoritm

```
repeat (n-1 times)          // iga korraga lisame ühe tipu (ja serva)
{ min = ∞;
  for (i=2; i <= n; i++)    // otsime juba valitutele lähima tipu
    if (0 <= distance[i] < min)
    { min = distance[i];
      vnear = i;            // lähim on i
    }
  e = edge(vnear, nearest[vnear]); // saame teada lisatava serva
  add e to F;
  distance[vnear] = -1;     // valitud tipp on nüüd juba valitute hulgas
  for (i=2; i <= n; i++)    // arvutame ümber kõigi vnear naabrite kauguse
    if (W[i][vnear] < distance[i])
    { distance[i] = W[i][vnear];  // kauguseks kaugus vnear-ist
      nearest[i] = vnear;         // lähimaks valitud naabriks vnear
    }}}
```

Keerukus: $T(n) = 2(n-1)(n-1) \in O(n^2)$, kus *n* on tippude arv

Prim'i algoritm – alternatiivne teostus prioriteetjärjekorraga

```
F = ∅;                  // valitud servade hulk
Y = {1};                // valitud tippude hulk
J = ∅;                  // tippe sisaldav prioriteetjärjekord
lisa tipud prioriteetjärjekorda J //prioriteet=kaugus Y-st
iga tipu kohta graafis salvesta kaugus ja naaber Y-s
while (pole valitud n-1 tippu)
{ võta järjekorrast J järgmine tipp u;
  lisa tipp u hulka Y;
  lisa seda ühendav serv hulka F;
  forall( v in u naabrid, kes ei ole Y-s);
     uuenda v kaugus ja lähim naaber Y-s; // decrease(v)
}
```

Keerukus: binaarkuhi: O(m lg n), Fibbonacci kuhi: O(m + n lg n)

Prim'i algoritmi korrektsus

Ahne algoritmi korrektsust tuleb tõestada - tuleb näidata, et lokaalse optimaalse valiku tegemine tagab globaalselt optimaalse tulemuse!

Invariant (omadus, mis kehtib peale iga sammu):

- Igal sammul lisame serva (***u,v***) s.t. (***u,v***) kaal on **minimaalne** servade seas kus ***u*** on puus ja ***v*** ei ole
- Iga samm annab minimaalse katva puu tippudele, mis on selle ajani valitud
- Kui kõik tipud on valitud, on meil kogu graafi minimaalne kattev puu!

Prim'i algoritmi korrektsus

- Algoritm lisab *n-1* serva, ilma tsüklit loomata. Seega loob ta sidusale graafile katva puu. (*tõestage!*).

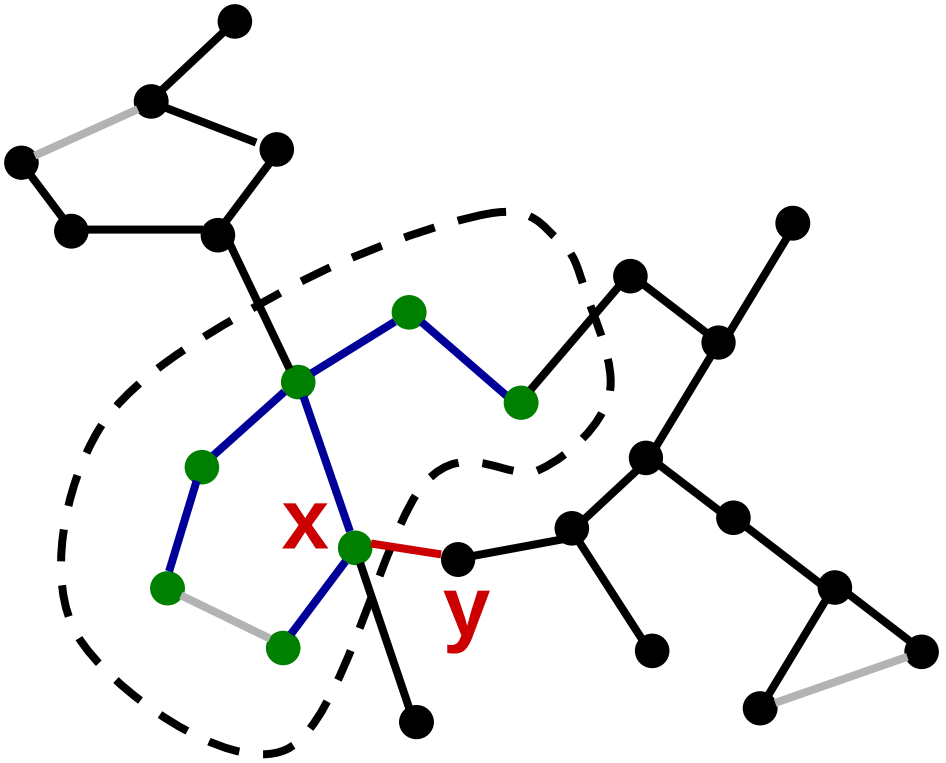
On see **minimaalne** kattev puu?

Oletame, et ei ole.

- Kuskil algoritmi töös peab olema hetk, kus tekib viga. Peab olema serv, mille lisamisel ei ole tekkinud puu enam **minimaalne** kattev puu valitud tippude jaoks.

Prim’i algoritmi korrektsus

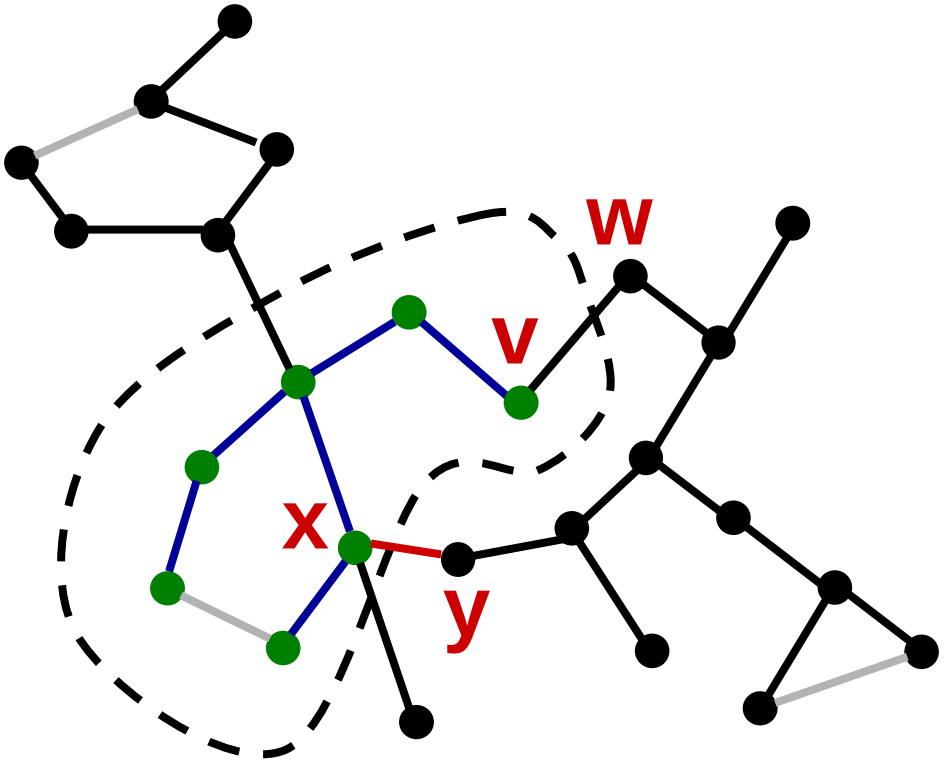
- Olgu ***G*** sidus, suunata graaf
- Olgu **S** Prim’i algoritmiga valitud servade hulk *enne* vale serva **(*x*,*y*)** valimist



- Olgu **V'** tippude hulk, mida seovad servad hulgas **S**
- Olgu **T** graafi ***G*** MKP sisaldades kõiki servi **S**, aga mitte serva **(*x*,*y*)**.

Prim'i algoritmi korrektsus

- Serv **(*x*,*y*)** ei ole hulgas **T**, seega peab olema hulgas **T** tee punktist ***x*** punkti ***y*** kuna **T** on sidus.
- Serva **(*x*,*y*)** lisamine hulka **T** looks tsükli



- Selles tsüklis on lisaks servale **(*x, y*)** ainult üks serv, millel on ainult üks tipp hulgas **V'**, nimetame selle servaks **(*v*,*w*)**

Prim'i algoritmi korrektsus

- Kuna Prim'i algoritm valis **(*v*,*w*)** asemel **(*x*,*y*)**, siis w(**v**,**w**) >= w(**x**,**y**).
- Võime luua uue MKP **T'** vahetades puus **T** kaare **(*v*,*w*)** kaarega **(*x*,*y*)** (*tõestage, et see on kattev puu*!).
- w(**T'**) ei ole suurem kui w(**T**)
- See tähendab, et **T'** on MKP
- …kuigi sisaldab kõiki kaari hulgas **S** ja lisaks kaart **(*x*,*y*)**

...Vastuolu

- T ja T' saavad olla MKP-d ainult siis kui w(**v**,**w**) = w(**x**,**y**).

Ahnete algoritmide tihti kasutatav struktuur

Saab kasutada kui kõik valitavad elemendid M ja nende “headus” h(m) on ette teada
k(m) on lahenduse korrektsuse kriteerium

Näiteks Kruskal’i algoritm

Greedy (M, w)

$A = \varnothing$

sorteeri M funktsiooni h(m) järgi

for ($x \in M$ vastavalt järjestusele h(m) järgi)

if $A \cup \{x\} \in k(m)$

then $A = A \cup \{x\}$

return A

Ahnete algoritmide tihti kasutatav struktuur

Saab kasutada kui valitavad elemendid M selguvad töö käigus

h(m) on lahenduse headuse kriteerium
k(M) on lahenduse korrektsuse kriteerium

Näiteks Prim'i algoritm

Greedy (M, w)

Lisa m_1 prioriteetjärjekorda J

for (kuni lahendus on leitud)

võta prioriteetjärjekorrast x

if k($A \cup \{x\}$) **then** $A = A \cup \{x\}$

foreach (x naaber)

lisa (või muuda prioriteeti) järjekorda J

return A

Ahne algoritmi loomine

- Tuleb leida lokaalse optimaalsuse kriteerium
- Tuleb tõestada, et see lokaalne kriteerium tagab ka globaalse optimaalsuse

  tõestus on tavaliselt vastuväiteline

  – pakume välja invariandi
  – näitame, et lõpliku arvu sammude järel järeldub invariandist korrektsus
  – oletame, et mõni algoritmi tehtud valik oli vale (rikkus invarianti)
  – näitame, et alternatiivse valiku korral ei oleks tulemus tulnud parem

Kokkuvõtteks

- Ahne algoritm võimaldab efektiivselt lahendada mitmeid optimiseerimisülesandeid
- Kui ülesandel on olemas ahne algoritm, siis on see üldjuhul efektiivsem kui teised algoritmid
- Ahne algoritm teeb lõplikke valikuid hetkeseisust lähtudes
- Probleemi võimalike olekute (valikute) puus oskab ahne kriteerium valida õige haru ilma teisi harusid läbi vaatamata.
- Ahne algoritmi kandidaadi korrektsust tuleb tõestada, st näidata et lokaalsest optimaalsus-kriteeriumist järeldub globaalne optimaalsus.