Algoritmid ja andmestruktuurid

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

Ahne algoritmi tsükkel

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

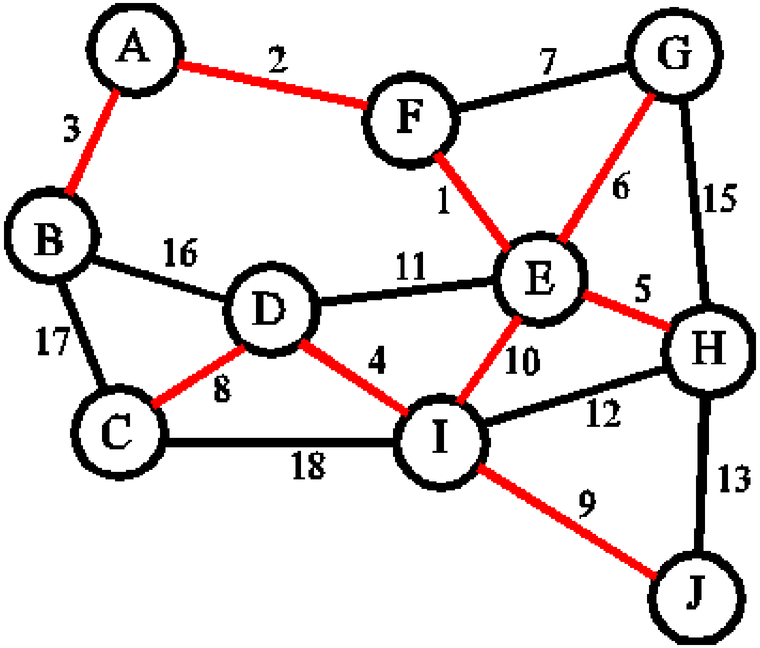
   N: valitakse kõige suurem olemasolev münt
2. Sobivuse kontroll
   - kontrollitakse kas selle elemendi valimisel on võimalik jõuda lahenduseni

   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

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



[Demo]

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

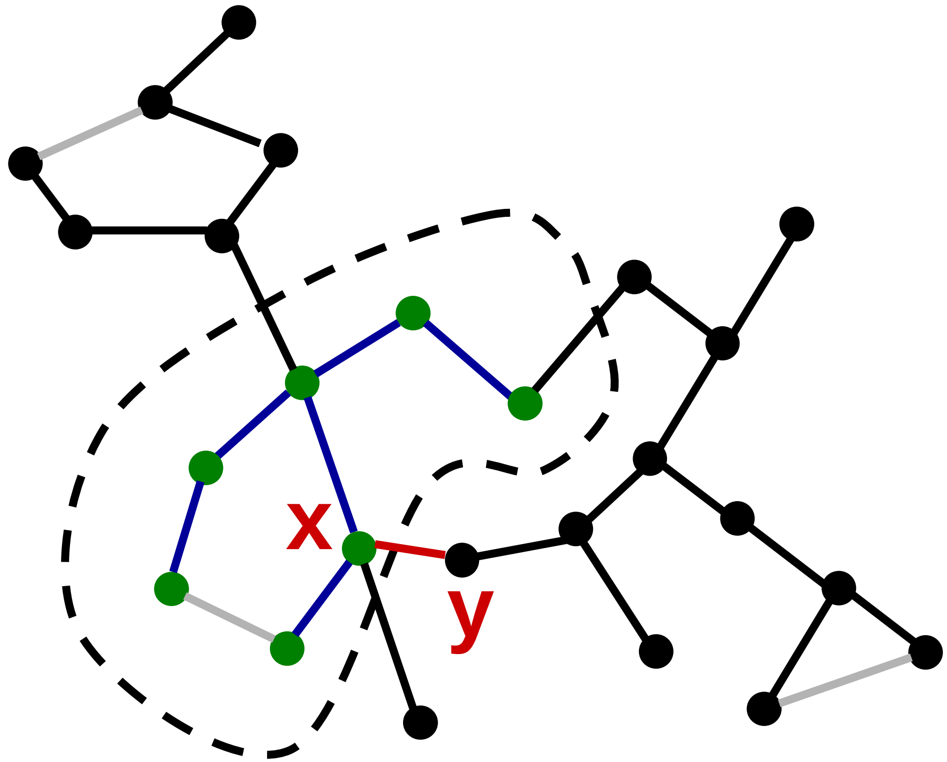
- Algoritm lisab *n-1* serva, ilma tsüklit loomata. Seega loob ta sidusale graafile katva puu. (*tõestage!*).

On see **minimaalne** kattev puu?

Oletame, et ei ole.

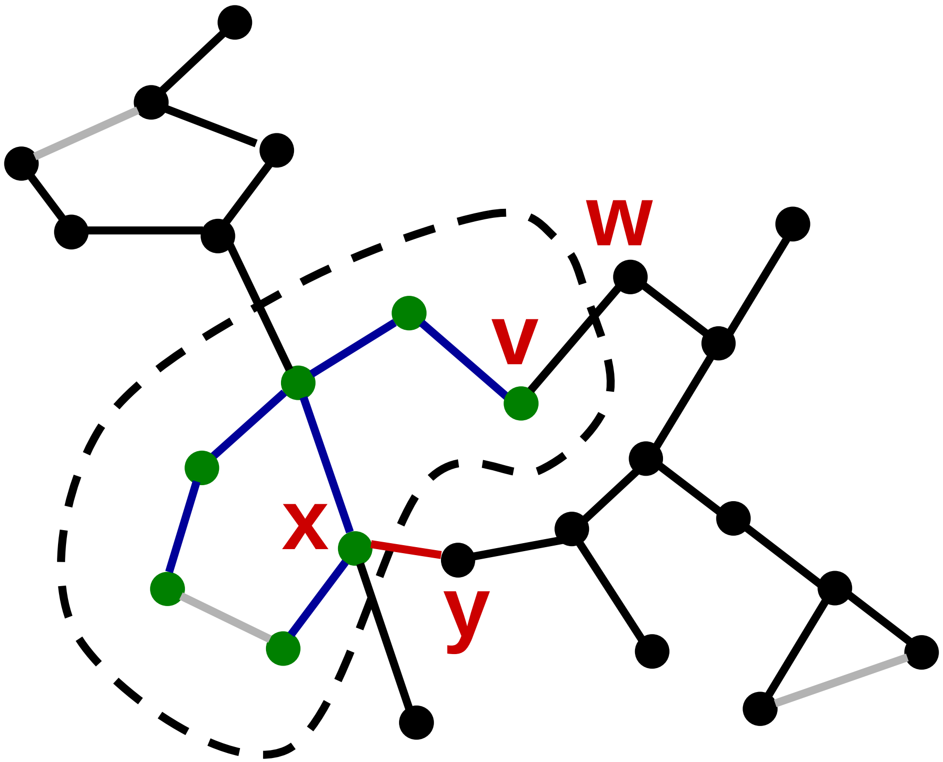
- Kuskil algoritmi töös peab olema hetk, kus tekib viga. Peab olema serv, mille lisamisel ei ole tekkinud puu enam **minimaalne** kattev puu valitud tippude jaoks.

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**).

Dijkstra lühimate teede algoritm

- Leiab lühimad teed ühest väljavalitud tipust igasse teise tippu (eeldusel, et graaf on sidus)
- Sarnane Prim'i katva puu algoritmile
  - kauguste massiivis peetakse meeles kaugust algsest tipust, mitte lähimast tipust
- Keerukus $O(n^2)$, kus *n* on tippude arv graafis
  - analoogselt Prim algoritmile on olemas ka efektiivsem prioriteetjärjekorral põhinev implementatsioon hõredate graafide jaoks

Dijkstra algoritm - idee

```
F = ∅;                  // servade hulk
Y = {v1};               // tippude hulk
while (pole lahendatud)
{ vali tipp hulgast V-Y millest on lühim
  tee tippu v1 kasutades ainult Y tippe
   lisa tipp Y-le;
   lisa kaar F-le;
   if (Y == V)
      lahendatud;
}
```

Dijkstra algoritmi animatsioon

Dijkstra algoritm

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

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

Eeldame, et W[i][j] = ∞, kui graafis pole serva (i,j)

Dijkstra algoritm

```
repeat (n-1 times)
{ min = ∞;
  for (i=2; i <= n; i++)
    if (distance[i] < min & !selected[i])
    { min = distance[i];
      vnear = i;
    }
  e = edge(touch[vnear], vnear);
  add e to F;
  for (i=2; i <= n; i++)
    if (distance[vnear]+W[vnear][i] < distance[i])
    { distance[i] = distance[vnear]+W[vnear][i];
      touch[i] = vnear;
    }
  selected[vnear]=1;
}}
```

Dijkstra'i algoritm – alternatiivne idee prioriteetjärjekorraga

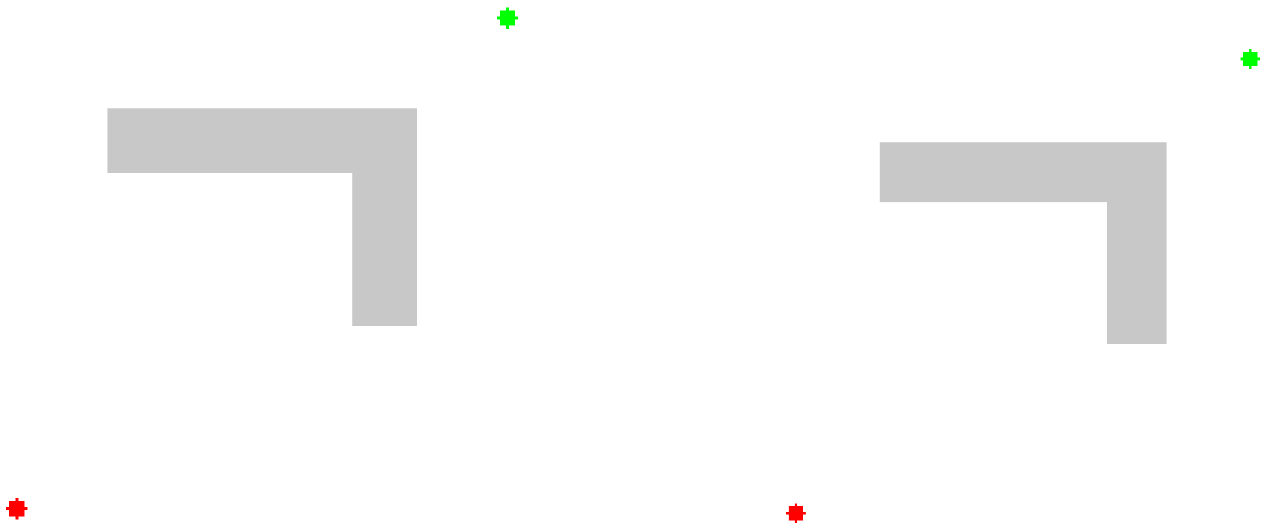
```
F = ∅;                      // valitud servade hulk
Y = {v1};                   // valitud tippude hulk
J = ∅;                      // tippe sisaldav prioriteetjärjekord
lisa tipud prioriteetjärjekorda J //prioriteet=kaugus v1 -st
iga tipu kohta graafis salvesta kaugus ja naaber v1
while (pole valitud 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 v1-st ja lähim naaber Y-s;
}
```

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

Lühim tee tipust A tippu B

- Dijkstra leiab lühimad kaugused kõigisse tippudesse sisuliselt laiuti otsinguga (lähemad enne)
  - Prioriteetjärjekorras on tipu F prioriteediks teadaolev lühim kaugus A-st
- A* algoritm leiab ühest punktist teise sisuliselt parim-enne otsinguga
  - Prioriteetjärjekorras on tipu F prioriteediks teadaolev lühim kaugus A-st + konservatiivne hinnang tipuni B
  - Konservatiivne hinnang ei või olla suurem kui tegelik lühim tee F-st B-ni. Geograafilisi ühendusi kajastavas graafis näiteks linnulennuline kaugus

Dijkstra ja A* animatsioonid



Konservatiivne (*admissible*) hinnang

- Sihifunktsioon f(n) on teepikkuse ja hinnangu summa

  g(n) – lühim teepikkus algtipust t_0 tipuni t

  h(n) – hinnang tipust tee lõpptipuni t_f

$$f(n) = g(n) + h(n)$$

- Hinnang on konservatiivne kui $f(n_f) >= g(n_f)$

A* algoritm

```
function A*(start, goal)
  closedSet := {}
  openSet := {start}
     // For each node, which node it can most efficiently be reached from.
     // If a node can be reached from ma
  cameFrom := an empty map
  gScore := map with default value of Infinity
  gScore[start] := 0
  fScore := map with default value of Infinity
  fScore[start] := heuristic_cost_estimate(start, goal)
```

https://en.wikipedia.org/wiki/A*_search_algorithm

A* algoritm

```
while openSet is not empty
    current := the node in openSet having the lowest fScore[] value
    if current = goal
        return reconstruct_path(cameFrom, current)
    openSet.Remove(current)
    closedSet.Add(current)
    for each neighbor of current
        if neighbor in closedSet
            continue        // Ignore the neighbor which is already evaluated.
        if neighbor not in openSet   // Discover a new node
            openSet.Add(neighbor)
        tentative_gScore := gScore[current] + dist_between(current, neighbor)
        if tentative_gScore >= gScore[neighbor]
            continue        // This is not a better path.
        cameFrom[neighbor] := current
        gScore[neighbor] := tentative_gScore
        fScore[neighbor] := gScore[neighbor] + cost_estimate(neighbor, goal)
return failure
```

Lühima tee ahned algoritmid

- Laiuti läbimine
  - Lühim tee kaaludeta graafis
- Dijkstra
  - Lühim tee kaaludega graafis
- A*
  - Lühim tee punktist punkti

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.