Föreläsning
Datastrukturer (DAT036)

Nils Anders Danielsson

2013-11-18

Idag

Mer om grafer:

- Minsta uppspännande träd (för oriktade grafer).
  - Prims algoritm.
  - Kruskals algoritm.
- Djupet först-sökning.

Cykel i *oriktad* graf

Väg av längd ≥ 1 som använder
varje kant max en gång.

Träd (utan rot)

Sammanhängande, acyklisk, oriktad graf.

Uppspännande träd

Träd som är delgraf och innehåller alla noder.

Minsta uppspännande träd (MST)

Uppspännande träd vars totala (kant)vikt är $\leq$ vikten av alla andra uppspännande träd.

Minsta uppspännande träd

Några tillämpningar:

- Konstruktion av nätverk.
- Klusteranalys.
- Bildsegmentering.
- Och mycket annat.

Minsta uppspännande träd

Några egenskaper:

- Existerar omm grafen är sammanhängande.
- Lägger man till en kant får man en cyklisk graf med exakt en cykel.
- Tar man sedan bort en kant från cykeln får man ett uppspännande träd.

Prims algoritm (för icketom graf)

```
Done = new set containing arbitrary node s
ToDo = V \ { s }
T    = new empty set  // MST för noder i Done.

while ToDo is non-empty do
  if no edge connects Done and ToDo then
    raise error: graph not connected

  (u,v) = cheapest edge connecting Done and ToDo
          (u ∈ Done, v ∈ ToDo)

  Done = Done ∪ { v }
  ToDo = ToDo \ { v }
  T    = T    ∪ { (u,v) }

return T  // Bara kanterna.
```

Prims algoritm

Väldigt lik Dijkstras algoritm.

- Båda *giriga algoritmer*.
- Lägger till billigaste nya noden.
- Dijkstra: Minsta avståndet från startnoden.
- Prim: Minsta avståndet från gammal nod.

Prims algoritm: korrekthet

Algoritmen ger ett uppspännande träd eller,
om grafen ej är sammanhängande,
ett felmeddelande:

- Invariant: `T` är ett uppspännande träd för noderna i `Done`.
- Invarianten håller i början:
  Trädet med noden `s` och inga kanter spänner upp $\{ \texttt{s} \}$.
- Invarianten bevaras:
  Lägger alltid till ny nod, aldrig cykel.
  (Om felmeddelande: ej sammanhängande.)
- När vi kör `return T` så är `Done = V`.

Prims algoritm: korrekthet

Algoritmen ger ett MST (om det finns något):

- Invariant: $\mathtt{T}$ är ett delträd av något MST.
- Invarianten håller i början:
  Det tomma trädet är ett delträd av alla MST.
- Invarianten bevaras:
  Antagande: $\mathtt{T}$ är ett delträd av något MST.
  Visa (för $k = (\mathrm{u}, \mathrm{v})$):
  $\mathtt{T} \cup \{ k \}$ är ett delträd av något MST.

Prims algoritm: korrekthet

- Antagande: `T` delträd av MST M.
- Visa: $\texttt{T} \cup \{ k \}$ delträd av *något* MST.
- Klara om $k \in M$. Annars finns cykel C med $k \in C$ och $C \setminus \{ k \} \subseteq M$.
- Betrakta mängden $K = C \setminus (\texttt{T} \cup \{ k \})$.
- k förbinder `Done` och `ToDo`, `T` gör det inte, C är en cykel $\Rightarrow$ en kant $k' \in K$ förbinder `Done` och `ToDo`.
- k' är minst lika dyr som k.
- $\texttt{T} \cup \{ k \}$ delträd av $(M \setminus \{ k' \}) \cup \{ k \}$ (som är ett MST).

Hemuppgift för intresserade

Visa att Dijkstras algoritm är korrekt.

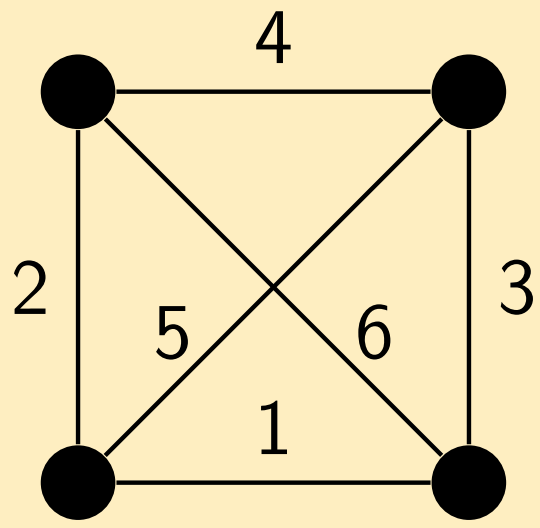
Prims algoritm

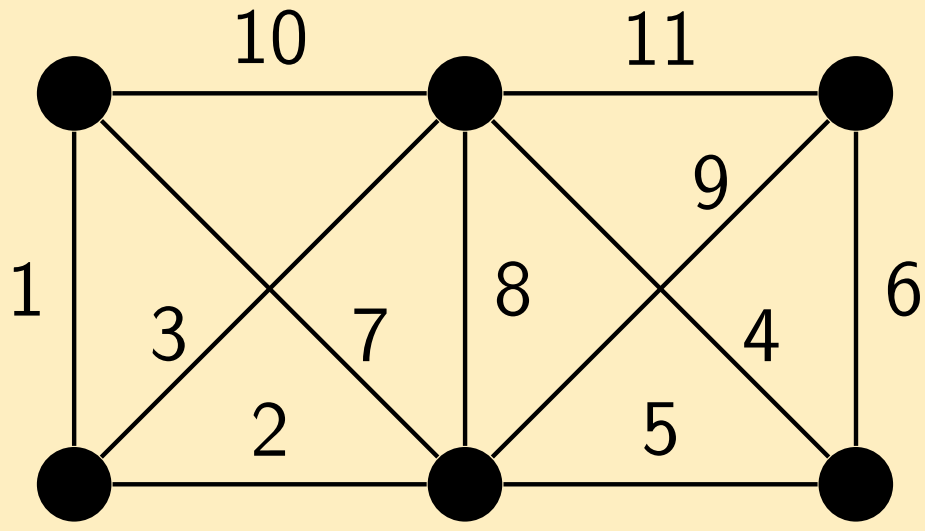
Kan implementera Prims algoritm
på ungefär samma sätt som Dijkstras.
(Glöm inte kontrollera att
grafen är sammanhängande.)

Tidskomplexitet:

- $O(|V|^2)$.
- $O(|E| \log |V|)$ (om $|V| = O(|E|)$).

Vad är den totala vikten av följande två grafers minsta uppspännande träd?





Kruskals algoritm

Välj hela tiden billigaste kanten
som inte skapar en cykel.

Kruskals algoritm

- Hur hittar vi billigaste kanten?
  Prioritetskö?
- Hur vet vi om kanten skapar en cykel?

Disjunkt mängd-ADTn

- Representerar partition av ändlig mängd.
- `new disjoint-set(n)`:
  Skapar $\{ \{ i \} \mid i \in \{ 0, 1, ..., \mathtt{n} - 1 \} \}$.
- `union`: Slår ihop två delmängder.
- `find`: Ger unik representant för delmängd.
  `find(i) == find(j)` omm
  `i` och `j` hör till samma delmängd.

Disjunkt mängd-datastruktur

En implementation (se boken):

- `new disjoint-set(n)`: $\Theta(\mathtt{n})$.
- `union`: *Nästan* konstant amorterad tid.
- `find`: *Nästan* konstant amorterad tid.

Kruskals algoritm

```
if |V| <= 1 then return empty set

Partition = new disjoint-set(|V|)
Edges     = new priority queue containing E,
            prioritised by edge weight
T         = new empty set

while Edges is non-empty do
  (u,v) = Edges.delete-min()

  if Partition.find(u) != Partition.find(v) then
    Partition.union(u,v)
    T = T ∪ { (u,v) }
    if |T| == |V| - 1 then return T

raise error: graph not connected
```

Vad är värstafallstidskomplexiteten för Kruskals algoritm? (Anta att `Edges` är en binär heap och `T` en lista. Anta inte att $|V| = O(|E|)$.)

- $\Theta(|V| + |E|)$.
- $\Theta(|E| \log |V|)$.
- $\Theta(|V| + |E| \log |V|)$.
- $\Theta((|V| + |E|) \log |V|)$.
- $\Theta(|V||E|)$.

Hemuppgift för intresserade

Visa att Kruskals algoritm är korrekt.

Kruskals algoritm

```
if |V| <= 1 then return empty set

Partition = new disjoint-set(|V|)                    Θ(|V|)
Edges     = new priority queue containing E,         Θ(|E|)
            prioritised by edge weight
T         = new empty set

while Edges is non-empty do                          O(|E|) ggr
  (u,v) = Edges.delete-min()                          O(log |V|)

  if Partition.find(u) != Partition.find(v) then
    Partition.union(u,v)
    T = T ∪ { (u,v) }
    if |T| == |V| - 1 then return T

raise error: graph not connected
```

Djupet först-sökning

Djupet först-sökning (DFS)

- Metod för att gå igenom alla noder och kanter.
- Har tidigare sett bredden först-sökning.
- Fungerar för oriktade och riktade grafer.

Djupet först-sökning: mall

```
visited = new array with indices {0,…,|V|-1}
            and all elements equal to false

// Startar/startar om sökning.
for v ∈ {0,…,|V|-1} do
  if not visited[v] then
    dfs(v)

// Utför sökning.
dfs(v) {
  visited[v] = true

  for w ∈ {w | (v, w) ∈ E} do
    if not visited[w] then
      dfs(w)
}
```

Djupet först-sökning: mall

```
visited = new array with indices {0,…,|V|-1}   O(|V|)
            and all elements equal to false

// Startar/startar om sökning.
for v ∈ {0,…,|V|-1} do                          O(|V|) ggr
  if not visited[v] then
    dfs(v)

// Utför sökning.
dfs(v) {                                        O(|V|) ggr
  visited[v] = true

  for w ∈ {w | (v, w) ∈ E} do                   O(|E|) ggr
    if not visited[w] then
      dfs(w)
}
```

Djupet först-sökning

Tidskomplexitet: $O(|V| + |E|)$ (linjär).

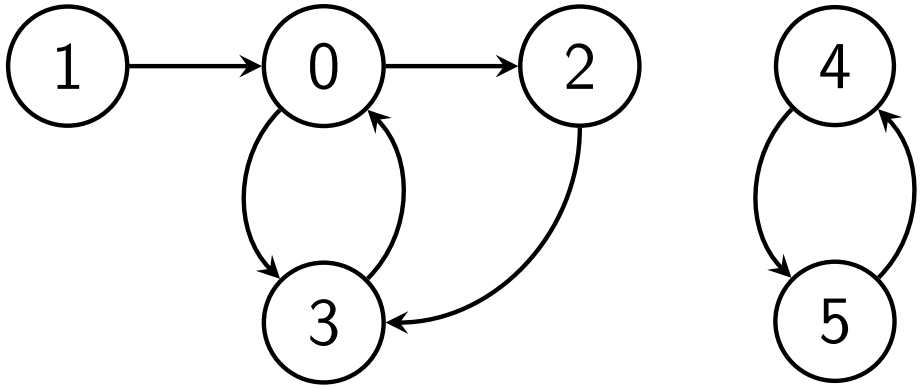
Uppspännande skog

Depth-first spanning forest:

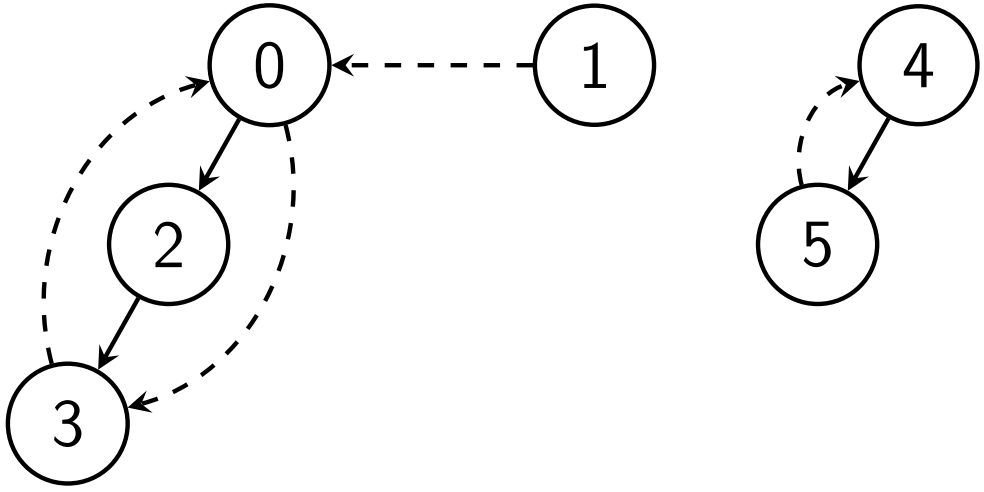
- Skog: Lista av träd.
- Ett träd för varje toppnivåanrop till `dfs`.
- Rekursiva anrop till `dfs` ger upphov till vanliga trädkanter.
- Om `visited[w]` = `true` får man istället en "speciell" kant:
  - Bakåtkant: Om kanten pekar uppåt.
  - Framåtkant: Om kanten pekar nedåt.
  - Tvärkant: I övriga fall ("åt vänster").

Uppspännande skog

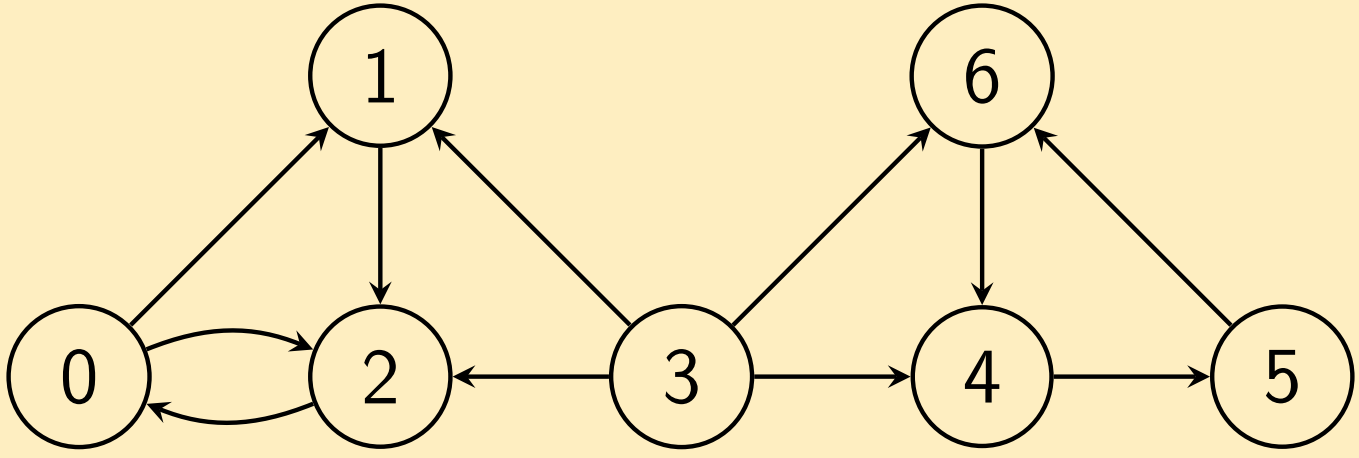
Graf:



Skog
(om vi söker från 0, 1 och 4, och väljer 2 före 3):



Hur många bakåtkanter innehåller
DFS-skogen för följande graf, givet att vi,
då vi kan välja mellan flera noder,
alltid väljer den lägst numrerade noden?



Djupet först-sökning

- DFS kan användas för att implementera andra algoritmer.
- Exempel:
  Är en oriktad graf sammanhängande?
  - Testa om DFS från godtycklig nod besöker alla noder (utan omstart).

Sammanhängande?

```
if |V| = 0 then return true

visited = new array with indices {0,…,|V|-1}
            and all elements equal to false
dfs(0)
for v ∈ {1,…,|V|-1} do
  if not visited[v] then return false
return true

dfs(v) {
  visited[v] = true
  for w ∈ {w | (v, w) ∈ E} do
    if not visited[w] then
      dfs(w)
}
```

Djupet först-sökning

- Återanvändning genom kopiering?
- Mer modulärt:

```
dfs :: Graph -> Forest

type Forest = [Tree]
type Vertex = Integer

data Tree = Root Vertex [Edge]
data Edge = Regular Tree
          | Forward Vertex
          | Back    Vertex
          | Cross   Vertex
```

- Olika implementationer av `dfs` för oriktade och riktade grafer.

Djupet först-sökning

- Modulär implementation:

```
connected :: Graph -> Bool
connected g = length (dfs g) <= 1
```

- Kanske mindre effektiv.
- Är grafen cyklisk?

```
cyclic :: Graph -> Bool
cyclic g = hasBackEdge (dfs g)
```

Träd: traverseringsordningar

Kan gå igenom ett träds noder i olika ordning:

Preorder Roten, barnträden från vänster till höger, vart och ett ”i preorder”.

Postorder Barnträden från vänster till höger, roten.

Inorder För binära träd:
vänster barnträd, roten, höger barnträd.

Kan generaliseras från träd till skogar
(vänster till höger).

Topologisk sortering

Kan sortera DAG topologiskt genom att utföra DFS och pusha noderna på stack i postordning:

```
-- Precondition: Grafen måste vara acyklisk.
topologicalSort :: Graph -> [Vertex]
topologicalSort g = postorderPush (dfs g)
```

Starkt sammanhängande komponenter

För riktade grafer:

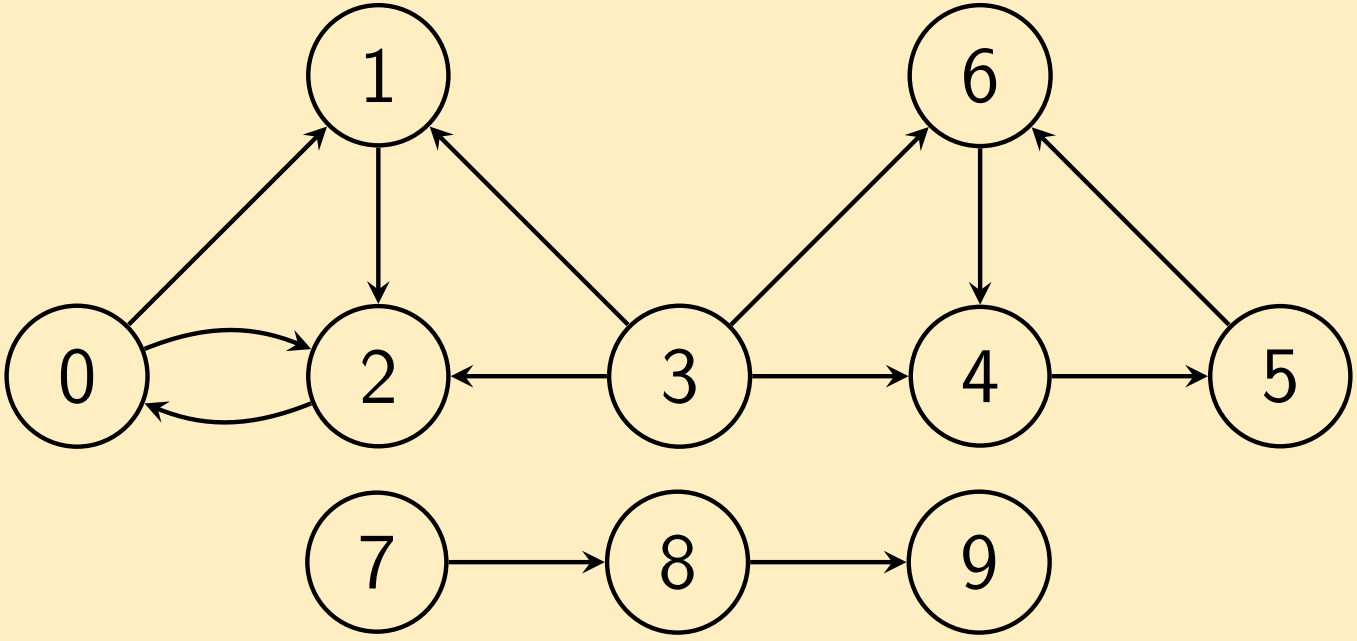
- Starkt sammanhängande graf:
  Om $u, v \in V$ så finns det en väg från u till v
  (och vice versa).
- Starkt sammanhängande komponent (SCC):
  Maximal starkt sammanhängande delgraf.
- Om varje SCC byts ut mot en ny nod
  (en per SCC) får vi en "multi-DAG".

Starkt sammanhängande komponenter

Exempel:

- Noder: Funktioner.
- Kanter: Anrop från en funktion till en annan.
- Funktioner i en SCC är ömsesidigt rekursiva.

Hur många starkt sammanhängande komponenter innehåller följande graf?



SCC-algoritm

Kan vi hitta alla SCCer?

- Kan hitta alla noder i ”löv-SCC” genom DFS (utan omstart) från någon av dem.
- Kan sedan ta bort (ignorera) löv-SCCn och fortsätta med annan löv-SCC.
- Men hur hittar man löv-SCCer?

SCC-algoritm

Utför DFS, lägg noder på stack i postordning.

- Översta noden (om någon)
  måste höra till en ”rot-SCC”.
- Tar man bort alla noder som hör till den SCCn
  så hör översta noden (om någon)
  återigen till en rot-SCC.
- Och så vidare...

För löv-SCC:
Utför DFS *baklänges* (på *reverserad* graf).

SCC-algoritm

1. Utför DFS baklänges,
   pusha noder i postordning.
2. Utför DFS framlänges,
   börja hela tiden med
   översta obesökta stacknoden.
   Varje sökning ger en SCC.

SCC-algoritm

```
-- Börja varje sökning i den första obesökta
-- listnoden.
orderedDFS :: Graph -> [Vertex] -> [Tree]

-- Reverserar grafen.
opposite :: Graph -> Graph

-- Resultatet är omvänt topologiskt sorterat.
sccs :: Graph -> [[Vertex]]
sccs g = map verticesInTree (orderedDFS g stack)
  where
  stack = postorderPush (dfs (opposite g))
```

SCC-algoritm

Tidskomplexitet: $O(|V| + |E|)$.

Sammanfattning

- Minsta uppspännande träd.
- Djupet först-sökning.

Nästa vecka:

- Sökträd.

Ge valfri feedback på hur kursen fungerar