Föreläsning 8
Datastrukturer (DAT037)

Fredrik Lindblad[1]

2016-11-23

[1]Slides skapade av Nils Anders Danielsson har använts som utgångspunkt. Se
`http://www.cse.chalmers.se/edu/year/2015/course/DAT037`

Förra gången:

- grafer
  - kortaste väg, Dijkstra
  - minsta uppspännande träd
  - Prims algoritm

Idag:

- Kruskals algoritm
- djupet-först-genomlöpning
- topologisk sortering
- starkt sammanhängande komponenter

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 (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: 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 $\{$ `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.

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: 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).

Prims algoritm

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

Tidskomplexitet
(om viktoperationer tar konstant tid):

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

Kruskals algoritm

- För osammanhängande grafer är en minimal uppspännande skog en uppsättning fria träd som innehåller alla noder i grafen och och har minimal total vikt.
- Prims algoritm hanterar inte osammanhängande grafer.
- Istället Kruskals algoritm

Kruskals algoritm

Idé:

- Ha en partition av noderna i grafen och en MST för varje delmängd.
- Lägg till nästa minsta båge som går mellan två delmängder (inte inom en delmängd).
- Slå samman dessa två delmängder.
- Korrektheten följer ett liknande resonemang som för Prims.

Kruskals algoritm

```
skapa vektor v av listor av bågar med ett element för varje nod
sätt varje element till tomma listan
skapa en prioritetskö pq och placera alla bågar i grafen i denna
start loop yttre
  start loop inre
    om pq är tom avsluta yttre loop
    plocka nästa båge, e, ur pq
    om v[e.start] != v[e.slut] avsluta inre loop
  slut loop inre
  låt lk vara den kortare listan av v[e.start] och v[e.end]
  och ll vara den längre
  lägg till e och bågarna i lk till ll
  för alla noder som är sammanbundna av bågarna i lk,
   låt dess plats i v peka på ll
slut loop yttre
returnera de unika listorna av bågar (en lista för sammanhängande graf)
```

Djupet först-sökning

Djupet först-sökning (DFS)

- Metod för att gå igenom alla noder och kanter.
- Har tidigare sett bredden först-sökning (kortaste vägen för oviktade grafer).
- 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}   Θ(|V|)
            and all elements equal to false

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

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

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

Djupet först-sökning

Tidskomplexitet: $\Theta(|V| + |E|)$ (linjär).

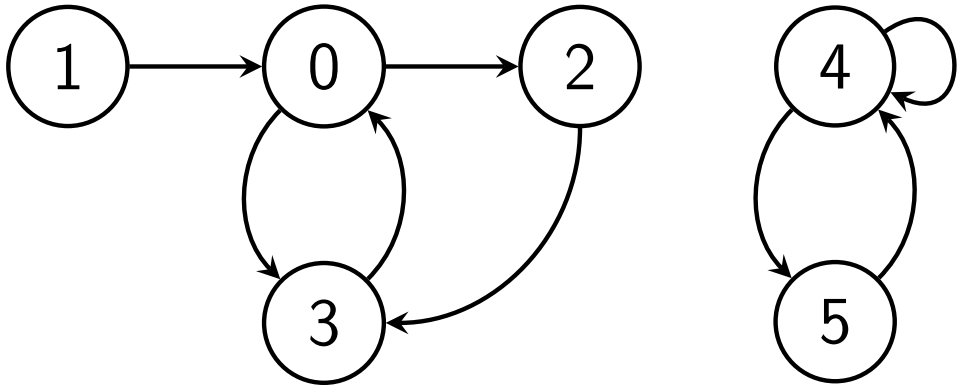
Uppspännande skog

Depth-first spanning forest:

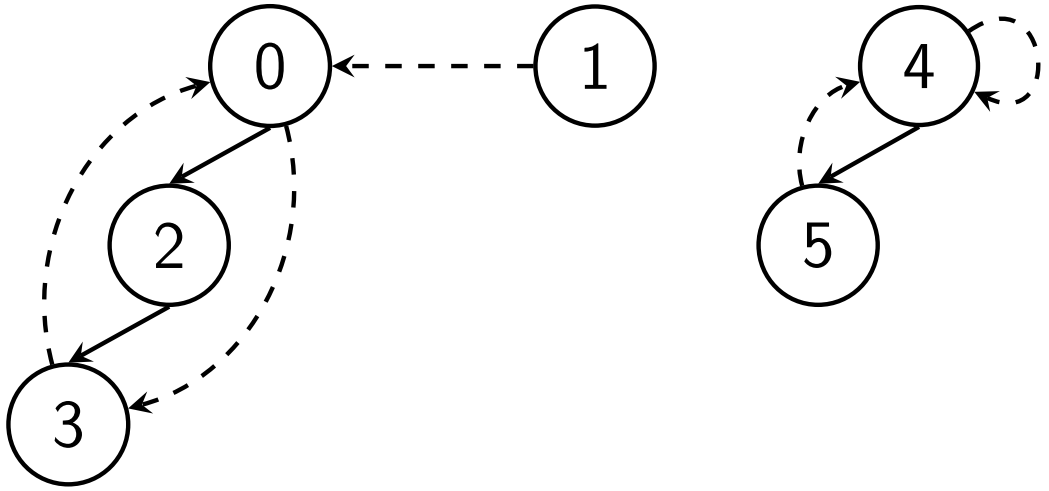
- Skog: Lista av träd.
- Ett träd för varje toppnivåanrop till `dfs`.
- Rekursiva anrop till `dfs` ger upphov till vanliga trädkanter.
- Om `visited[w]` = `true` får man istället en "speciell" kant:
  - Bakåtkant: Om kanten pekar uppåt (eller på samma nod).
  - Framåtkant: Om kanten pekar nedåt.
  -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

Uppspännande skog

Graf:



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



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)
}
```

Topologisk sortering

Topologisk sortering

Definition:

- Total ordning av V.
- Om det finns en väg från v_1 till v_2 så är $v_1 < v_2$.

Exempel:

- Förkunskapskrav $\Rightarrow$ giltig ordning av kurser.

Topologisk sortering

- Ointressant för oriktade grafer.
- Cykel $\Rightarrow$ ingen topologisk sortering.
- DAGs (riktade acykliska grafer) kan alltid sorteras topologiskt.
- Tillräckligt villkor för att vara cyklisk: Grafen innehåller minst en nod, och alla noder har ingrad > 0.

Topologisk sortering: enkel algoritm

```
r = new empty list

while V ≠ ∅ do
  if any v ∈ V with indegree(v) = 0 then
    r.add-last(v)
    remove v from G
  else
    raise error: cycle found

return r  // Nodes, topologically sorted.
```

Topologisk sortering: enkel algoritm

```
r = new empty list

while V ≠ ∅ do
  if any v ∈ V with indegree(v) = 0 then
    r.add-last(v)
    remove v from G
  else
    raise error: cycle found

return r  // Nodes, topologically sorted.
```

Kan vi undvika radering?

Topologisk sortering: enkel algoritm (2)

```
r = new empty list
d = map from vertices to their indegrees
    // null for nodes in r.

repeat |V| times
  if d[v] == 0 for some v then
    r.add-last(v)
    d[v] = null
    for each direct successor v' of v do
      decrease d[v'] by 1
  else
    raise error: cycle found

return r  // Nodes, topologically sorted.
```

Grafrepresentation

Pseudokod: Behöver mer information för tidskomplexitetsanalys.

Grafrepresentation (den här gången):

- Noder numrerade $0, 1, ..., |V| - 1$.
- Array `adjacent` med $|V|$ positioner.
- `adjacent[i]` innehåller grannlista (länkad lista) för nod `i`.

`r`: dynamisk array, `d`: array.

Pseudokod

Gärna mer detaljer och bättre namn på tenta.
Exempel:

```
// d är en map från nodindex till
// /virtuella/ ingrader, de ingrader
// respektive nod skulle ha om alla noder i
// r togs bort från grafen. Den virtuella
// ingraden för noder i r är null.
d = new array of size |V|

// Initialisera d.
for i in [0,...,|V|-1] do
  d[i] = 0
for i in [0,...,|V|-1] do
  for each direct successor j of i do
    d[j]++
```

Pseudokod

Gärna mer detaljer och bättre namn på tenta.
Exempel:

```
// d är en map från nodindex till
// /virtuella/ ingrader, de ingrader
// respektive nod skulle ha om alla noder i
// r togs bort från grafen. Den virtuella
// ingraden för noder i r är null.
d = new array of size |V|

// Initialisera d.
for i in [0,...,|V|-1] do                    O(|V|) ggr
  d[i] = 0                                   O(1)
for i in [0,...,|V|-1] do
  for each direct successor j of i do
    d[j]++
```

Pseudokod

Gärna mer detaljer och bättre namn på tenta.
Exempel:

```
// d är en map från nodindex till
// /virtuella/ ingrader, de ingrader
// respektive nod skulle ha om alla noder i
// r togs bort från grafen. Den virtuella
// ingraden för noder i r är null.
d = new array of size |V|

// Initialisera d.
for i in [0,...,|V|-1] do                    O(|V|) ggr
  d[i] = 0                                   O(1)
for i in [0,...,|V|-1] do                    O(|V|)
  for each direct successor j of i do        O(|E|) ggr
    d[j]++                                   O(1)
```

Topologisk sortering: enkel algoritm (2)

```
r = new empty list                                  O(1)
d = map from vertices to their indegrees            O(|V| + |E|)
    // null for nodes in r.

repeat |V| times                                    O(|V|) ggr
  if d[v] == 0 for some v then                        O(|V|)
    r.add-last(v)                                     O(1)
    d[v] = null                                       O(1)
    for each direct successor v' of v do              O(|E|) ggr
      decrease d[v'] by 1                             O(1)
  else
    raise error: cycle found                          O(1)

return r  // Nodes, topologically sorted.             O(1)
```

Totalt: $O(|V|^2 + |E|) = O(|V|^2)$.

Topologisk sortering med kö

```
r = new empty list
d = map from vertices to their indegrees
q = queue with all nodes of indegree 0

while q is non-empty do
  v = q.dequeue()
  r.add-last(v)
  for each direct successor v' of v do
    decrease d[v'] by 1
    if d[v'] = 0 then
      q.enqueue(v')

if r.length() < |V| then
  raise error: cycle found

return r  // Nodes, topologically sorted.
```

Analysera värstafallstidskomplexiteten.

- $\Theta(|V|)$.
- $\Theta(|E|)$.
- $\Theta(|V| + |E|)$.
- $\Theta(|V|^2)$.

Bonusövning

Vad händer om man använder en stack istället för en kö?

Topologisk sortering med kö

```
r = new empty list                                  Θ(1)
d = map from vertices to their indegrees            Θ(|V| + |E|)
q = queue with all nodes of indegree 0              O(|V|)

while q is non-empty do                             O(|V|) ggr
  v = q.dequeue()                                     Θ(1)
  r.add-last(v)                                       Θ(1)
  for each direct successor v' of v do              O(|E|) ggr
    decrease d[v'] by 1                               Θ(1)
    if d[v'] = 0 then                                 Θ(1)
      q.enqueue(v')                                   Θ(1)

if r.length() < |V| then                            Θ(1)
  raise error: cycle found                          Θ(1)

return r  // Nodes, topologically sorted.          Θ(1)
```

Topologisk sortering

Kan sortera DAG topologiskt genom att:

- Utföra DFS.
- Gå igenom träden i uppspännande skogen i preordning, med skillnaden att (barn)träden gås igenom från höger till vänster.

Starkt sammanhängande komponenter

Starkt sammanhängande komponenter

För riktade grafer:

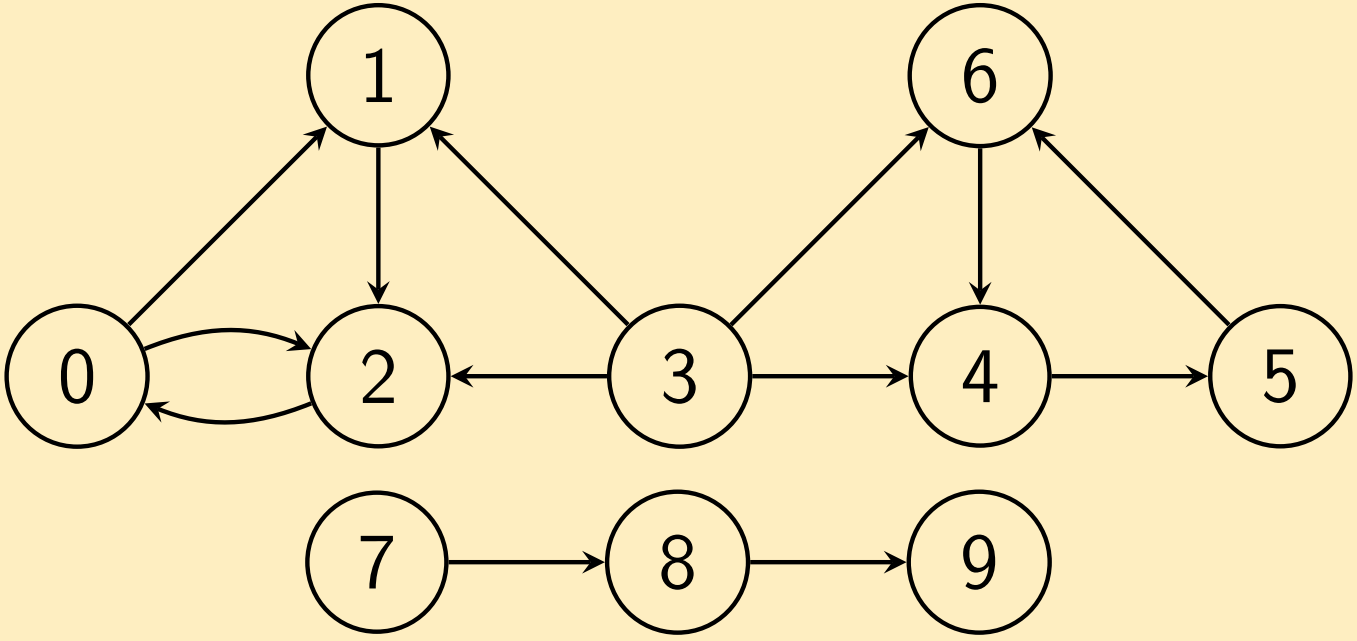
- Starkt sammanhängande graf:
  Om $u, v \in V$ så finns det en väg från u till v
  (och vice versa).
- Starkt sammanhängande komponent (SCC):
  Maximal starkt sammanhängande delgraf.
- Om varje SCC byts ut mot en ny nod
  (en per SCC) får vi en acyklisk multigraf.

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, gå igenom uppspännande skogen i preordning, från höger till vänster, som i topologisk sortering:

- Första noden (om någon)
  måste höra till en ”rot-SCC”.
- Tar man bort alla noder som hör till den SCCn så hör nästa nod (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.
2. Skapa en nodlista med `preorderRL`.
3. Utför DFS framlänges,
   börja hela tiden med
   första obesökta listnoden.
   Varje sökning ger en SCC.

Tidskomplexitet: $\Theta(|V| + |E|)$.