Föreläsning
Datastrukturer (DAT036)

Nils Anders Danielsson

2012-11-19

Idag

Mer om grafer:

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

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 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.

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

Prims algoritm

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

Tidskomplexitet:

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

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.

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).

Övning

Visa att Dijkstras algoritm är korrekt.

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)`: $O(\mathtt{n})$.
- `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
```

Kruskals algoritm

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

Partition = new disjoint-set(|V|)                    O(|V|)
Edges     = new priority queue containing E,         O(|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
```

Minsta uppspännande träd

En tillämpning: klusteranalys.

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).

Övning

Konstruera en linjär algoritm som avgör om en *oriktad* graf är sammanhängande.

Övning

Konstruera en linjär algoritm som avgör om en *oriktad* graf är cyklisk.

Övning

Konstruera en linjär algoritm för topologisk sortering av *riktade* acykliska grafer.

Använd djupet först-sökning.

Sammanhängande?

Testa om DFS från godtycklig nod
besöker alla noder (utan omstart).

```
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)
}
```

Cyklisk?

Testa om DFS träffar på redan besökt nod.

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

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

Topologisk sortering (för acyklisk graf)

Pusha noderna på stack i postordning.

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

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

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.

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.

```
visited = new array with indices {0,…,|V|-1}
stack   = new empty stack containing node indices
sccs    = new empty list containing sccs
            (lists of node indices)

// Första djupet först-sökningen.
for v ∈ {0,…,|V|-1} do visited[v] = false
for v ∈ {0,…,|V|-1} do if not visited[v] then dfs1(v)

// Andra djupet först-sökningen.
for v ∈ {0,…,|V|-1} do visited[v] = false
while stack is non-empty do
  v = stack.pop()
  if not visited[v] then
    scc = new empty list of nodes
    sccs.add-last(scc)
    dfs2(scc, v)

return sccs
```

```
dfs_1(v) {
  visited[v] = true

  // Baklänges!
  for w ∈ {w | (w, v) ∈ E} do
    if not visited[w] then
      dfs_1(w)

  stack.push(v)
}
```

```
dfs_2(scc, v) {
  visited[v] = true

  scc.add-last(v)

  for w ∈ {w | (v, w) ∈ E} do
    if not visited[w] then
      dfs_2(scc, w)
}
```

SCC-algoritm

- Tidskomplexitet: $O(|V| + |E|)$.
- Notera att komponentlistan är omvänt topologiskt sorterad.

Sammanfattning

- Minsta uppspännande träd.
- Djupet först-sökning.

Nästa vecka:

- Sökträd.