Föreläsning 1. Introduktion och sökning i graf

Vad är en algoritm?

Först: Vad är ett problem?
Består av **indata** och ett **mål**.
Indata: [En beskrivning av en struktur.]
Mål: [Kan vara Ja/Nej, ett tal eller en struktur.]
Exempel:

Indata: Ett heltal n.
Mål: Är n ett primtal?
Vi kallar problemet PRIMTAL.

Indata: En lista L med tal.
Mål: En lista L^* som är L i sorterad ordning.
Vi kallar problemet SORTERING.

Indata: En graf G.
Mål: Storleken på den längsta vägen i G.
Vi kallar problemet LÄNGSTA VÄG.

Algoritmer

En algoritm löser ett problem på ett *mekaniskt* sätt. (En algoritm är som ett dataprogram.)
Tre egenskaper hos en idealiserad algoritm är
1. De är plattformsoberoende.
2. De är helt mekaniska och ointelligenta.
3. De ger alltid korrekt resultat.

Kursmål

1. Att lära oss konstruera bra algoritmer.
2. Att lära oss identifiera problem som inte kan lösas effektivt.
3. Att lära oss metoder för att ändå kunna hantera svåra problem.

Fallstudie

Vi studerar en del grundläggande begrepp genom att titta på ett speciellt problem: GENOMSÖKNING AV GRAF.
Indata: En riktad graf G och en startnod s.
Mål: Hitta alla noder som kan nås från s via en riktad väg.
Vi skall beskriva en lösning på problemet med hjälp av *pseudokod*.

Pseudokod

Pseudokod är ett semiformellt programspråk.
Den utgör en mall för ett riktigt program.
Den lyfter fram huvudidéerna i en algoritm.
Den kan användas på valfri detaljnivå.

Vi ger nu en pseudokod för GENOMSÖKNING AV GRAF. Vi tänker oss att grafen har en nodmängd V och en kantmängd E.

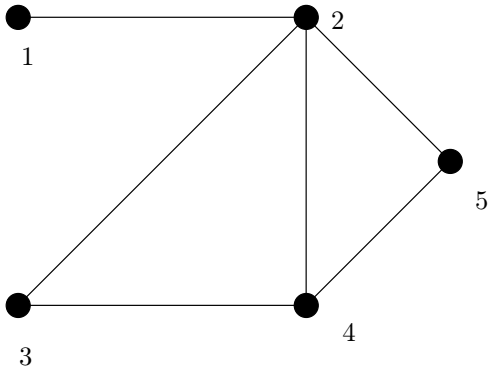
```
GRAFSÖKNING(G, s, t)
(1)     U ← {s}
(2)     while t ∉ U och det finns kant (v, w) med v ∈ U
        och w ∉ U
(3)         U ← U ∪ {w}
(4)     return U
```

Mängden U ger alla noder som kan nås från s.

Det finns två stora frågor kring denna enkla pseudoalgoritm. Den första är hur graferna skall representeras. Den andra är i vilken ordning kanterna i algoritmen skall väljas ut.

Representation av grafer

Vi visar hur *oriktade* grafer kan representeras. Tekniken för riktade grafer är i stort sett samma.



Grannmatris:

	1	2	3	4	5
1		1	0	0	0
2			1	1	1
3				1	0
4					1
5					

Grannlista:

1	2			
2	1	3	4	5
3	2	4		
4	2	3	5	
5	2	4		

Representationen *kantmatris* förekommer också: Det är en $|V| \times |E|$-matris där element (i, j)är 1 om $v_i \in e_j$, 0 annars.

Täta och glesa grafer

En *tät* graf har $\Theta(|V|^2)$ kanter och representeras med fördel med en grannmatris. En *gles* graf har $\Theta(|V|)$ kanter och representeras lämpligen med någon form av grannlistor. Glesa grafer är vanliga i tillämpningar och prestanda för en algoritm på sådana kan i praktiken vara lika viktigt som tidskomplexiteteni värsta fallet. *Graden* hos ett hörn är antalet kanter sominnehåller hörnet medan gradtalet hos en grafär det högsta gradtalet för något hörn i grafen.

En förbättrad algoritm

Vi använder så kalla Djupet-Först-Sökning (DFS). Vi utgår ifrån att grafen är representerad med grannlistor.
DFS går igenom de hörn som kan nås från starthörnet s.Detta görs genom att gå så långt det går i grafen utan att återkomma till något hörn och sedan stega tillbaka tills något nytt hörn går att besöka.

```
DFS(V, E, s)
(1)     foreach u ∈ V
(2)         vis(u) ← 0
(3)         p[u] ← NULL
(4)     DFS − Visit(s)
```

```
DFS-Visit(u)
(1)     vis(u) ← 1
(2)     foreach granne v till u
(3)         if vis(v) = 0
(4)             p[v] = u
(5)             DFS-Visit(v)
```

Efter körning av algoritmen innehåller vis information om vilka noder vi lyckades besöka. Pekarna p ger information om hur sökvägarna såg ut.

Tidskomplexitet

Tidskomplexiteten är antalet operationer $T(n)$ som krävs för att köra algoritmen då *indatas storlek* är n.
Vi använder ofta s.k. *ordoklasser* för att beskriva storleken på komplexitet.

Tillväxt hos funktioner

Komplexiteten beskrivs ofta i $O(\cdot)$-notation. Den bortser från konstanta faktorer och anger vad som händer då $n \to \infty$.

- $f(n) \in O(g(n))$ om $f(n)$ växer högst lika snabbt som $g(n)$.

  D.v.s. det finns $c > 0$ och n_0 så att $f(n) \leq cg(n)$ för alla $n \geq n_0$.
- $f(n) \in \Theta(g(n))$ om $f(n)$ och $g(n)$ växer lika snabbt.
- $f(n) \in \Omega(g(n))$ om $f(n)$ växer minst lika snabbt som $g(n)$.

Minneskomplexitet

Ett annat komplexitetsmått är **minneskomplexitet**:
Storleken på minnet som krävs för att utföra en beräkning.
Tidskomplexitet anses vara det viktigaste komplexitetsmåttet.

Tre problem med definition av komplexitet

1. Tiden är proportionell mot antalet **operationer**. Vad räknas som en operation?

Ex: Beräkna $521 \cdot 394 = 205274$ med "vanlig" algoritm. Hur många operationer krävs?
Två möjliga svar:
a. En operation!
b. 521 har 9 bitar. 394 har 8 bitar. Totalt krävs $9 \cdot 8 = 72$
De två typerna av kostnad är:

Enhetskostnad:

- Varje grundläggande operation tar en tidsenhet.
- Varje variabel upptar en minnesenhet.

Bitkostnad:

- Varje operation på en enskild bit tar en tidsenhet.
- En bit upptar en minnesenhet.

Bägge dessa mått på kostnaden används.

2. Indatas storlek.
Antag att ett komplicerat objekt är indata. Vad är då indatas storlek? I princip är det storleken lika med antalet bitar i den effektivaste representationen av objektet i en datastruktur. I praktiken kan det ibland vara svårt att avgöra vad det är.

3. Medelvärde/Värsta fall.
Ex: Quicksort är en känd algoritm för att sortera en lista tal. I medeltal kräver den $O(n \log n)$ operationer. I värsta fall kan den dock kräva $O(n^2)$ operationer. Vanligen används **värsta fall** som mått.

Skäl: Det är ofta svårt att känna till sannolikhetsfördelningen på möjliga indata för en korrekt medeltalsberäkning. Värsta fall är enklare att beräkna och är i många fall ungefär lika med medeltal.

Definition av effektiv algoritm

En algoritm som arbetar i tid $O(n^k)$ för något k. Kallas för **polynomiell** algoritm.

Vår algoritm för Djupet-Först-Sökning har komplexitet $O(|V|+|E|)$. Talet $|V|+|E|$ är ett naturligt mått på indatas storlek och algoritmen är därför effektiv.

Ex: En algoritm för att testa om m är ett primtal.

```
PRIMTAL(m)
(1)     for i ← 2 to √m
(2)         if i|m
(3)             return Inte primtal
(4)     return Primtal
```

Denna algoritm har komplexitet $O(\sqrt{m})$. Det går att inse att det är för långsamt för stora tal. Det naturliga måttet på indatas storlek är inte m utan $n = \log m$. Det som önskas är en algoritm som går i tid $O((\log m)^k)$ för något k.