Problemlösning och programmering under press

Sammanställt av Mikael Goldmann

27 augusti 2007

Förord

Det här kompendiet är baserat på föreläsninar och anteckningar på kursen *Problemlösning och programmering under press*. Att skriva föreläsningsanteckningar i LaTeX är ett moment som infördes 2005, dels för att åstadkomma ett kompendium och dels för att träna studenterna i att uttrycka sig i skrift.

Ett stort tack går till de som varit med om att utveckla kursen, eller som föreläst på den: Per Austrin, Alexander Baltatzis, David Rydh, Gunnar Kreitz, Fredrik Niemelä och Mattias de Zalenski. Dessutom har ett stort antal studenter bidragit till innehållet i detta kompendium: Johnne Adermark, Marcus Dicander, Erik Edin, Niclas Eklöw, Mikael Eliasson, Joakim Eriksson, Elias Freider, Patrik Glas, Jesper Hjertstedt, Mats Linander, John Lindberg, John Lindholm, Ulf Lundström, Nils Loodin, Jenny Melander, Dmitry Palagin, Henrik Sjögren, Anders Sjöqvist, Oscar Sundbom, Pehr Söderman,Oscar Tengwall, Alix Warnke, Per Wennersten, Henrik Ygge.

Vem skrev om komb. sökning 2005??

Innehåll

Kapitel 1

Dynamisk programmering och giriga algoritmer

Denna föreläsning började med en kort genomgång av domarsystemet Kattis som används i kursen och regler för samarbete vid laborationer och hemuppgifter. Sedan diskuterade vi algoritmkonstruktionmetoder såsom giriga algoritmer och dynamisk programmering.

1.1 Administration

1.1.1 Kattis

Domarsystemet Kattis finns på http://kattis.csc.kth.se. När man skickar in en lösning på ett problem till domaren kan man få följande svar:

- Accepted - Lösningen godkänd.
- Submission error - Problemet okänt eller felaktigt format på inskickningen.
- Compilation error - Kattis kunde inte kompilera filen.
- Illegal function - Vissa funktioner är inte tillgängliga i Kattis av säkerhetsskäl, t.ex. att läsa en fil från disken etc. Försöker man det får man detta fel.
- Runtime error - Fel vid körning.
- Memory limit exceeded - Programmet använder mer minne än vad som är tillåtet. Kan också visa sig som ett Runtime error.
- Output limit exceeded - Programmet genererar alldeles för mycket output till standard out.

- Time limit exceeded - Programmet hinner inte exekvera klart inom den förbestämda tidsgränsen. Observera att Kattis stänger av programmet i så fall.
- Wrong answer - Programmet svarar fel på testdata.

1.1.2 Betygsgränser

Observera att maximala antalet poäng på OVN1 och LAB1 är 80p respektive 108p. För betyg 3/4/5 krävs 24/36/48 poäng. Mikael var noga med att poängtera att även om maxpoängen är hög jämfört med betygsgränserna är det ändå inte lätt att få bra betyg i kursen, och att man inte behöver satsa på att ta alla poäng, men att det är viktigt att jobba kontinuerligt med kursen.

1.1.3 Samarbete

- Hemtal: Ska lösas individuellt. Att dela idéer är okej, men inte kod!
- Anteckningar: Skrivs i grupp med 2-3 personer.
- Problemsessioner: 2 pers/grupp, är det ojämnt antal studenter i tid till en session blir det en grupp med 3 personer, ingen ska alltså behöva göra en problemsession själv.
- Labbar: Löses i grupper om 1-3 personer, men alla måste skicka in en separat lösning där koden innehåller en kommentar som anger gruppmedlemmar.

Det finns en nyhetsgrupp för kursen där studenter kan diskutera. Den heter nada.kurser.popup06 och finns på inn.nada.kth.se.

1.2 Algoritmkonstruktion

1.2.1 Dekomposition

Går ut på att dela upp problemet i mindre delproblem, lösa dem och sedan sätta ihop resultatet. Exempel är Quicksort och Mergesort. Detta är förmodligen bekant från tidigare kurser.

1.2.2 Giriga algoritmer

Dessa är typiska för att hitta en lösning som är “billigast” eller “bäst”. Idén går ut på att utvidga en partiell lösning stegvis, lokalt optimalt, och se till att valet inte förstör möjligheten att utvidga till en optimal lösning på slutet. Ett exempel är Kruskals algoritm för *minimum spanning tree*. Den väljer i varje steg den billigaste kanten att utvidga som inte bildar en cykel med de redan valda kanterna.

Ett annat exempel är *täckning med intervall* [1].

Algoritm 1: Täckning med intervall
Input: $[L, R]$, $[a_1, b_1]$, $[a_2, b_2]$, ... , $[a_n, b_n]$; där $L, R, a_i, b_i \in \mathbf{Z}$, $a_i \leq b_i$ och $L \leq R$.
Output: En minimalt stor delmängd $A \subseteq \{1, 2, ..., n\}$ så att $\cup_{i \in A}[a_i, b_i] \supseteq \mathbf{Z} \cap [L, R]$. Mata ut "fail" om det inte finns någon sådan delmängd.

(1) Bland alla $[a_i, b_i]$ där $a_i \leq L \leq b_i$ välj det i där b_i är störst. Om det inte finns $[a_i, b_i]$ som täcker L **return** fail
(2) $A \leftarrow A \cup \{i\}$
(3) $L \leftarrow b_i + 1$
(4) **if** $L > R$
(5) **return** A
(6) goto (1)

För att övertyga oss om att algoritmen ovan är korrekt bevisar vi följande påstående: Första gången vi väljer intervall $[a_i, b_i]$ girigt väljs ett intervall som kan ingå i en optimal lösning.

Antag att vi väljer $[a_j, b_j]$ först och att det existerar en optimal lösning A som inte innehåller detta intervall. Då $\exists i \in A$ så att $L \in [a_i, b_i]$, alltså finns det ett annat intervall i i den optimala lösningen A som täcker L.

Betrakta

$$A' = (A \setminus \{i\}) \cup \{j\}$$

I och med att j har det största övre gränsen b av alla intervall som täcker L är det lätt att se att

$$[L, R] \cap [a_i, b_i] \in [L, R] \cap [a_j, b_j]$$

alltså förlorar vi ingenting på att välja $[a_j, b_j]$ framför $[a_i, b_i]$.

Vi kan lätt klara av det här i $O(n \log n)$ tid genom att först sortera alla intervall så att $a_1 \leq a_2 \leq ... \leq a_n$. Sen går vi igenom den sorterade listan och väljer det intervall med högst b_i där $a_i \leq L$. Sorteringen tar $O(n \log n)$ tid, och sedan stegar vi genom listan i $O(n)$ tid. Totalt alltså $O(n \log n)$ tid.

1.2.3 Dynamisk programmering

Idén med dynamisk programmering är att få ner den, ofta exponentiella, tiden som krävs för beräkna ett rekursivt uttryck genom att spara framräknade värden och använda dessa om och om igen istället för att beräkna dem på nytt.

[1] Ofta vill man täcka hela det reella intervallet $[L, R]$, inte bara heltalspunkterna, och då behöver man modifiera den givna algoritmen.

Ett exempel är att beräkna $\binom{n}{k}$. Formeln är

$$\binom{n}{k} = \begin{cases} 1 & \text{om } n = k \text{ eller } k = 0 \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{annars} \end{cases}$$

Den vanliga rekursiva lösningen är långsam. Den tar $O(\binom{n}{k})$ tid att exekvera. Vi kan göra samma sak på $O(nk)$ genom att använda dynamisk programmering.

Algoritm 2: Variant 1 - top-down med memoisering
Input: $n, k \in \mathbf{Z}$ där $n \geq k \geq 0$
Output: $\binom{n}{k}$
Bin(n, k)
  if $(n{=}{=}k \mid\mid k{=}{=}0)$
    return 1
  if (!computed[n,k])
    tab[n, k] $\leftarrow$ Bin$(n-1, k-1)$+Bin$(n-i, k)$
    computed[n, k] $=$ *true*
  return tab[n, k]

Algoritm 3: Variant 2 - bottom-up, vanlig dynamisk programmering
Input: $n, k \in \mathbf{Z}$ där $n \geq k \geq 0$
Output: $\binom{n}{k}$
Bin(n, k)
  for $i = 0$ **to** n
    tab[$i, 0$] $\leftarrow$tab[i, i] $\leftarrow$ 1;
  for $i = 2$ **to** n
    for $j = 1$ **to** $\min(i-1, k)$
      tab[i, j] $\leftarrow$tab[$i-1, j$]+tab[$i-1, j-1$];
  return tab[n, k];

Vilken version som lämpar sig bäst varierar från fall till fall. Memoisering är bättre om man inte tror att man kommer att fylla hela tabellen, eftersom den bara räknar ut de värden som kommer att behövas. "Bottom-up" fyller alltid hela tabellen, men gör det effektivare än memoisering så om hela eller nästa hela tabellen behövs så är "bottom-up" ofta snabbare. Vilken version som är lättast att koda varierar också från fall till fall så det får man ta med i beräkningen när man bestämmer sig för den ena algoritmen framför den andra.

Ett till exempel är *maximum sum* som går ut på att hitta den största summan av bredvidliggande tal i en array. Vi har $x_1, x_2, \ldots, x_n \in \mathbf{Z}$. Vi vill hitta

$$\max_{l \le r} f(l,r), \text{ där } f(l,r) = \sum_{i=l}^{r} x_i$$

Den naiva lösningen att beräkna enligt formeln ovan tar kubisk tid. Men om vi inför en vektor prefix[] enligt

$$\text{prefix}[r] = \sum_{k=1}^{r} x_k$$

så kan vi beräkna varje $f(l,r)$ i konstant tid

$$f(l,r) = \text{prefix}[r] - \text{prefix}[l-1]$$

och bästa summan kan beräknas i totalt kvadratisk tid.

Vi kan även lösa problemet i linjär tid genom att för varje r beräkna den största summan som slutar i r. Detta kan göras i linjär tid genom att beräkna

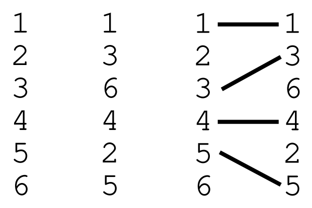
$$\begin{aligned} m(1) &= x_1 \\ m(r+1) &= x_{r+1} + \max(0, m(r)) \end{aligned}$$

Bästa summan är då $\max_{1 \le r \le n} m(r)$.

1.2.4 Längsta växande delsekvens

Två elektroniska kretsar ligger intill varandra och varje har n stycken kontakter. Dessa är numrerade, på den ena är de i ordning $1, 2, ..., n$ medans på den andra finns samma nummer men inte i ordning. Vi vill förbinda kontakterna med samma markering. Frågan är hur många förbindelser vi maximalt kan dra utan att de korsar varandra.

Betrakta följande exempel:



Det maximala antalet förbindelser är i detta fall 4 och det bästa sättet att förbinda chipen syns ovan.

Vi betecknar kontakterna på andra chipet som $x[i]$. Efter att ha tittat noga på bilden ovan konstaterar vi att det vi söker är egentligen längden av den längsta växande delsekvensen av $x[1...n]$.

	start	1	3	6	4	2	5
längden av delföljden	0	1	2	3	3	3	3
största tillagda element	$-\infty$	1	3	6	6	6	6

Det kan vara lockande att försöka lösa detta problem med en girig algoritm. Vi skulle kunna gå igenom $x[]$ och i varje steg lägga till $x[i]$ till delföljden om det är större än det största hittills tillagda elementet. Med denna metod får vi följande:

Detta ger som synes inte optimalt svar (maximala längden blir 3 istället för 4). Ett bättre tillvägagångssätt är att använda dynamisk programmering.

Vi skapar en vektor last[] där last$[i]$ är det lägsta värde som kan avsluta en växande sekvens av längd i. I början sätter vi last$[0] = -\infty$ och last$[i] = \infty$ för $i = 1, ..., n$. Vi går igenom $x[]$ och för varje i söker vi j så att last$[j-1] < x[i] \leq$ last$[j]$ och sätter last$[j] = x[i]$. För varje $x[i]$ kommer det att finnas en plats j eftersom vi initialiserat vektorn som vi gjorde. Så här ser det ut för vårt exempel:

	start	1	3	6	4	2	5
last[0]	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
last[1]	∞	1	1	1	1	1	1
last[2]	∞	∞	3	3	3	2	2
last[3]	∞	∞	∞	6	4	4	4
last[4]	∞	∞	∞	∞	∞	∞	5
last[5]	∞	∞	∞	∞	∞	∞	∞
last[6]	∞	∞	∞	∞	∞	∞	∞

Svaret är då största talet i last[] som inte är ∞. Att hitta rätt plats i last[] för $x[i]$ kan göras med binärsökning, eftersom last[] hela tiden är sorterad i stigande ordning. Antalet element i $x[]$ är n, och tidskomplexiteten för hela algoritmen blir då $O(n \log n)$.

Kapitel 2

Debugging

Föreläsningen handlade om felsökning och hur man ska förebygga fel när man programmerar. Vi studerade tre olika typer av fel: Kompileringsfel, Kraschfel och Logiska fel.

2.1 Återskapa felet med hjälp av bra testfall

Om ditt program kraschar av någon oförklarlig anledning så behöver du testfall för att återskapa felet och därefter hitta och eliminera det. För att provocera fram en bugg behöver man en bra uppsättning testfall som om möjligt ska innehålla exempel från var och en av kommande kategorier.

2.1.1 Triviala fall

För att kontrollera att programmet klarar basfallen. Klassiska basfall är 0, 1, tomma strängen, tomma listan och null.

2.1.2 Små instanser

Små instanser är värdefulla eftersom man kan verifiera för hand att svaret är korrekt.

2.1.3 Stora instanser

Med stora instanser kan det vara svårt att avgöra om svaret är helt korrekt, men det bör gå att avgöra om svaret är rimligt. Dessutom kan man passa på att kontrollera om programmet uppfyller tidskravet.

2.1.4 Extremvärden

För att framkalla krasch eller oväntat beteende. Typiska extremvärden är maxint och minint. Ibland är det bara extremvärden som får programmet att krascha eller göra fel.

2.1.5 Corner case

Ett corner case uppkommer när flera variabler antar sina extremvärden samtidigt. Precis som vid extremvärden så kan det vara så att programmet bara gör fel just här.

2.1.6 Oväntad indata

En del program visar bara sina sårbarheter vid oväntade fall. Som exempel kan en dålig quicksort-implementation krascha när den sorterar en lista med tal som alla antar samma värde.

2.1.7 Hittar inte felet ändå?

Om man trots dessa brutala men nödvändiga testfall fortfarande inte kan hitta felet så kan det bero på något av följande:

- Glömt att synkronisera trådar (dock ovanligt i denna kurs eller på programmeringstävlingar)
- Glömt att initialisera variabler (kan orsaka oväntat beteende)
- Alignment-problem: Minne kan behöva allokeras på en adress som är en jämn multipel av 2 för att sedan läsas, detta problem är inte särskilt vanligt

2.2 Lokalisera felet utan debugger

Det gäller att här skilja på kompileringsfel (enklast att hitta och åtgärda), körningsfel (krascher), och logiska fel (Datorn gör exakt vad du sagt åt den att göra, men inte vad du vill att den ska göra). Vid kompileringsfel skriver kompilatorn ut på vilken rad felet ligger. Tänk på att läsa kompilatorns felmeddelanden uppifrån eftersom en del fel fortplantar sig nedåt. Vid körningsfel skriver Java ut vilken rad som orsakat kraschen. C och C++ gör det inte men vi kommer att titta på tekniker för att snabbt och definitivt ta reda på var i koden som programmet kraschade.

2.2.1 Binärsökning...

Binärsökning kan användas till att lokalisera fel genom att upprepade gånger halvera sökintervallet.

...med bortkommentering av kod

Binärsök genom koden genom att kommentera bort stora stycken. Kommentera bort hälften av koden och se om det fortfarande kraschar. I så fall, upprepa proceduren om och om igen.

...med spårutskrifter

Lägg in utskrifter på väl valda platser i koden (mitt i sökintervallet är väl valt) som gör det lätta att avgöra (1) var i koden det skrevs ut och (2) värden på kritiska variabler. Använd preprocessorn för att hantera spårutskrifter i C. Skriv detta längst upp i implementationsfilen.

```
#ifdef _DEBUG
#define DEBUG(X) X
#else
#define DEBUG(X)
#endif
```

Skriv spårutskrifter så här:

```
DEBUG(printf("LOOP: i = %d, j = %d\n", i , j));
```

Se till att flusha spårutskrifter när du jagar kraschbuggar. Om du inte gör det så kan programmet ha passerat en utskriftsrad trots att du inte ser den på skärmen. I C skriver du:

```
fflush(stdout);
```

Använd preprocessorn för att hantera spårutskrifter i C++. Skriv detta längst upp i implementationsfilen:

```
#ifdef _DEBUG
#define dout debug && std::cerr
#else
#define dout if(false) std::cerr
#endif
```

Skriv spårutskrifter så här:

```
dout << "LOOP: i = " << i << " j = " << j << std::endl;
```

Om _DEBUG inte är definierat kommer ovanstående rad att ersättas med ett if(false) i början. Eftersom if(false) aldrig kan vara sant så kommer raden inte att skrivas ut. För att flusha buffern skriver du:

```
std::cerr.flush();
```

och i Java skriver du långt uppe i klassen (ändra till true vid felsökning):

```
private final int debug=false;
```

Spårutskrifter på formen:

```
if(debug){
    System.err.println("i = " + i + " j = " + j);
}
```

Flusha genom att:

```
System.err.flush();
```

2.2.2 Assert

Använd *assert* för att kontrollera antaganden om variabelvärden och minnesåtgång. Är strängen kortare än en Shakespearepjäs? Är listan längre än ett schackparti mellan Kasparov och Carola? Är din signed int fortfarande inte negativ?
I C/C++ skriver du:

```
assert(villkoret)
```

Om *villkoret* inte är uppfyllt så stannar exekveringen på den raden.
I Java skriver du:

```
assert(villkoret)[:"felmeddelande"];
```

Om *villkoret* evalueras till false så skickas *felmeddelande* till ExceptionHandler.

2.3 Lokalisera felet med debugger

På skolan finns många debuggers tillgängliga, bland annat: gdb (för C/C++), ddd (grafiskt skal för gdb) och jdb (för Java). Nedan beskriver vi gdb som ett exempel på en debugger. Andra debuggers har likande kommandon.

För att kunna använda gdb måste ditt program kompileras med flaggan -g.

```
$ gcc -g <filename>.cpp
$ gdb a.out
```

Här är en lista på kommandon i gdb:

Kommando	Förtydligande
r (run)	
l (list)	l <radnummer>, l<filnamn>:<rad>
b (breakpoint)	b<rad>, b<funktionsnamn>
cont (continue)	kör koden tills den stöter på en breakpoint eller slutet av programmet
d/p (display/print)	d <x>, p <x> skriver ut värdet av x
und (undisplay)	und <x> slutar visa variablen x
step	step utför en rad instruktioner - om ett funktionsanrop, en rad inuti den
n (next)	next utför en rad instruktioner - om ett funktionsanrop, kör hela och returnerar
bt (backtrace)	Backar körningen till förra breakpoint
del (delete)	del N, där N är en breakpoint, endast del tar bort alla breakpoints
i (info)	info stack ger t.ex information om stacken
set	set variable x=12, tilldelning
cal (call)	call foo() anropar funktionen foo()
def (define)	define bar
h (help)	hjälp om andra kommandon
fin (finish)	avslutar gdb

Ofta vill man placera ut breakpoints i början av funktioner så att exekveringen stoppas när man nått så långt. Vill vi stanna före funktionen Foo() så skriver vi: *br Foo*. gdb kommer då ge feedback om var denna brytpunkt sattes in, exempelvis:
Breakpoint 2 at 0x2290: file main.cpp, line 30
Det går även alldeles utmärkt att istället för ett funktionsnamn ange en rad i koden, ex: *br 20* Prova nu att köra kommandot run, exekveringen kommer att stanna vid varje breakpoint. Vill du ta bort en breakpoint skriver du helt enkelt *delete N*, där *N* är id-numret på breakpointen som ska bort. Har du väldigt många breakpoints då ett id-nummer inte säger dig särskilt mycket kan du alltid skriva: *info break* för att få detaljer. En bra nybörjartutorial finner du på t.ex:
`http://www.cs.princeton.edu/~benjasik/gdb/gdbtut.html`

2.4 Ytterligare tips för att motverka fel

Naturligtvis vore det bästa att inte skriva fel från början, så vi börjar där.

2.4.1 Förebyggande

För att spara tid och slippa onödig felsökning under programmeringsfasen så kan du förbereda dig på flera sätt.

Design by contract

Design by contract är en programutvecklingsmetod där preconditions och postconditions kan ses som ett kontrakt som en metod eller funktion måste uppfylla. De variabler och datastrukturer som en metod eller funktion behandlar antas ha vissa egenskaper. Dessa egenskaper kan man ange explicit i form av assertions. Om någon av dessa antaganden inte har uppfyllts så har vi ett kontraktbrott och programexekveringen avbryts omedelbart och obönhörligen.

Använd en bra editor

En bra editor brukar ha tydlig syntax highlighting och stöd för kompilering och exekvering, men framför allt är en bra editor någon som du är van att arbeta med och som du känner till väl.

Vanliga fel

Lär dig vilka fel du brukar göra. Sök med Google efter vanliga fel i just ditt programmeringsspråk.

Post Mortem

Se till att förstå de problem som uppkom och hur du löste dem. Det är förebyggande inför nästa gång.

2.4.2 Under felsökningen

Buggar är illmariga och kan ha överlevt dina förebyggande insatser, men oroa dig inte - Vi har tips för även denna fas!

Läs dokumentationen

Bra dokumentation är skriven av kloka människor som av erfarenhet eller insikt lyckats förutspå många av de problem som kan uppstå. Det kan vara en idé att läsa den ordentligt för felet kan bero på att man missförstått en dokumenterad detalj.

Läs och förstå problemet samt indata och utdata

Ett välskrivet problem beskriver i detalj vilka antaganden man kan och inte kan göra om indata. De enklaste buggarna bygger på att man läser in eller skriver ut på fel sätt. Lite knepigare är om indata kan anta mycket stora värden. Mycket knepigare är om man har en algoritm som löser problemet men den är för generell och långsam, så programmet spräcker tidsgränsen. Eller åt andra hållet: Man kan ha gjort för många antaganden om indata så att algoritmen bara löser vissa specialfall av problemet.

Dialogmetoden

Sokrates förespråkade att använda dialoger för att lära ut abstrakta koncept. Tag lärdom av detta och försök förklara ditt program eller din idé noggrannt för någon annan. Det kan öka bådas förståelse för problemet. Många självklara fel kommer fram när man förklarar programmet för någon annan. Under förklaringen så kanske du själv upptäcker saker du inte riktigt har förstått eller antaganden du gjort som inte stämmer och då vet man var man ska rikta sin energi. Det är också viktigt att få bra feedback på sina förklaringar. Om man lär sig vilken typ av frågor som brukar ställas eller vad frågeställaren brukar påpeka så lär man sig förklara på ett bra sätt från början.

Papper och penna

Papper och penna är enkla, klassiska, beprövade och mycket effektiva verktyg för felsökning. Skriv ut koden. Många har lättare att förstå den från ett papper än från skärmen. Under tävlingspass innebär det också att datorn blir ledig för att lösa andra problem medans du jagar buggar. Stega igenom och räkna i huvudet. Gör bra anteckningar för att avlasta närminnet. Skriv misstänkta algoritmer i pseudokod för att kolla om de verkligen fungerar.

Djupare fel

Fastna inte i att ändra småsaker. Ibland kan programmet ha ett djupare problem (fel algoritm, fel idé). Spara undan den gamla filen under ett annat namn och testa att göra större ändringar. Om det senare visar sig att den gamla lösningen var bättre kan man enkelt gå tillbaka.

Ta pauser

Ibland blir man så låst att man inte kommer någonstans. Då kan det vara bra för såväl hälsa som problemlösningsförmåga att ta en paus och återvända till problemet senare. Hjärnan kan bearbeta problem trots att man inte anstränger sig för att tänka på dem.

Ändra bara en sak i taget

Om du utgår från ett program som du redan förstår någorlunda och sedan gör många ändringar innan du kompilerar om och testar det så kan det bli svårt att lokalisera nya fel som du råkade introducera. När man ändrat 5 saker som verkade helt galna så kanske programmet plötsligt gör fel på testfall som det tidigare klarade. När man letar svårfunna fel under tidspress är det sista man vill att introdcuera ännu fler svårfunna fel.

Kapitel 3

Strängmatchning

Denna föreläsning behandlar problemet strängmatchning. Den naiva ansatsen, ändliga automater och Knuth–Morris–Pratts algoritm gås genom och kortare översikter av några metoder för matchning av flera mönster i samma text ges.

3.1 Strängmatchning – naiv algoritm

Problemet är att givet en text $T[1...n]$ och ett mönster $P[1...m]$ hitta alla förekomster av P som delsträng till T. Den naiva algoritmen undersöker alla positioner i texten mellan 1 och $n - m$ för att se om mönstret börjar där. När en position är undersökt så skiftas mönstret ett steg åt höger i T. Tidskomplexiteten för algoritmen som kan ses nedan är $\Theta(mn)$.

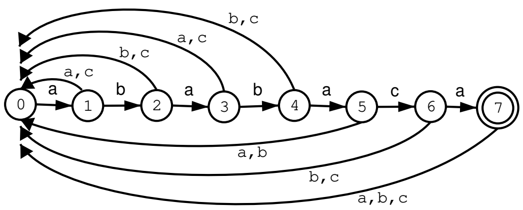
Algoritm 4: Strängmathcning - naiv algoritm
Input: Text $T[1...n]$, mönster $P[1...m]$.
Output: Positioner i T där P börjar.

```
STRING MATCHING(T[1...n], P[1...m])
    for j ← 1 to n − m
        for i ← 1 to m
            if P[i] ≠ T[i + j − 1] then break
            if i ≥ m then OUTPUT j
```

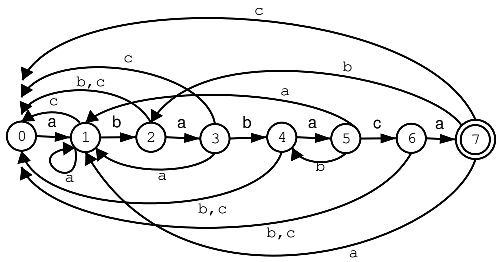
3.2 Strängmatchning – ändlig automat

En ändlig automat för strängmatchning är en riktad graf, vars noder och kanter kallas *tillstånd* respektive *övergångar*. Varje övergång i automaten är märkt med ett eller flera tecken ur alfabetet Σ. Automaten har ett *starttillstånd* och minst ett *accepterande tillstånd*.

Strängmatchning utförs genom att vandra i automaten, med början i starttillståndet, tills det accepterande tillståndet nås. I varje tillstånd väljs nästa tillstånd genom att ta ett nytt tecken x ur T och sedan låta den övergång som är märkt x ange nästa tillstånd. Betrakta följande automat som söker efter mönstret $P = ababaca$:



Man befinner sig i tillstånd q om de senaste q lästa tecknen i T matchar de första q tecknen i P. Som synes kan endast en förekomst av *ababaca* leda till ett accepterande tillstånd. Problem uppstår dock t ex om $T = abababaca$, ty efter de första fem tecknen hamnar vi i starttillståndet och missar därför förekomsten av P med början i tecken 2 i T. Lösningen på detta är att lägga till övergångar enligt följande figur:



Automaten hamnar nu i det accepterande tillståndet precis om P förekommer i T. Eftersom varje tecken i T undersöks en enda gång och övergångarna kan göras i konstant tid så blir tidsåtgången $\Theta(n)$.

För att göra övergångar i konstant tid kan vi använda en övergångsmatris $\delta[q, x] =$ nästa tillstånd, där q är nuvarande tillstånd och x är indatatecken. Problem uppstår vid stora alfabeten, t ex unicode, då matrisen kan bli orimligt stor och tidskrävande att konstruera.

3.3 Strängmatchning – Knuth–Morris–Pratt

Problemet med stora övergångsmatriser kan avhjälpas med Knuth–Morris–Pratt algoritm (KMP). Här använder vi oss av en prefixfunktion $\pi[1..m]$ sådan att $\pi[q]$ är längden av det längsta prefix av $P[1..q-1]$ som också är ett suffix av $P[1..m]$.

Algoritm 5: Strängmatchning – Knuth–Morris–Pratt
Input: Text $T[1...n]$, mönster $P[1...m]$.
Output: Positioner i T där P börjar.

```
KMP(T[1...n], P[1...m])
    π ← PREKMP(P)
    q ← 0
    for i ← 1 to n
        while q > 0 and P[q + 1] ≠ T[i]
            q ← π[q]
        if P[q + 1] = T[i] then q ← q + 1
        if q = m
            OUTPUT i − m + 1
            q ← π[q]
```

Algoritm 6: Prefixfunktionen π tillhörande KMP
Input: $P[1...m]$
Output: $\pi[1...m]$ så att
$\pi[q] = max\{k | k < q, P[1...k] \text{ suffix av } P[1...q]\}$

```
PREKMP(P[1...m])
    π[1] ← 0
    for q ← 2 to m
        k ← π[q − 1]
        while k > 0 and P[k + 1] ≠ P[q]
            k ← π[k]
        if P[k + 1] = P[q] then π[q] ← k + 1
                            else π[q] ← 0
```

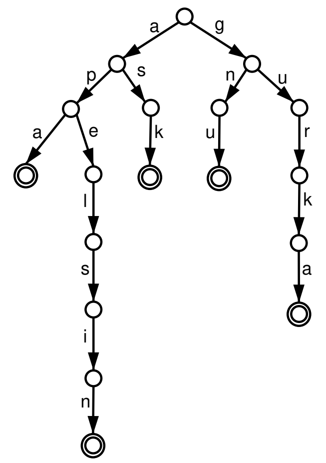
Då π kan förberäknas i tid $\Theta(m)$ blir tidsåtgången för KMP $\Theta(m + n)$. Eftersom $m < n$ rimligen gäller, så är algoritmen $\Theta(n)$ i tid.

3.4 Översikter av algoritmer för matchning av flera mönster i samma text

Under föreläsningen gavs även ett par mer övergripande genomgångar av några algoritmer för samtidig matchning av flera mönster i samma text. Upprepad användning av algoritmer som KMP fungerar givetvis men bättre metoder finns att tillgå.

3.4.1 Trie

En trie är ett sökträd som representerar en mängd S av ord över alfabetet Σ. Trädets kanter är märkta med tecken ur Σ och dess noder är märkta med 1 eller 0. En nod märkt 1 svarar mot att märkningen av kanterna, från trädets rot ner till den märkta noden, bildar ett ord i S. En nod märkt 0 betyder att motsvarande ord ej finns i S. Betrakta följande trie för $S = \{apa, apelsin, ask, gnu, gurka\}$:



Valet av datastruktur för representation av trädet beror på om låg minnesåtgång eller snabba sökningar skall prioriteras. Om hastighet är viktigast så kan det vara lämpligt att i varje nod hålla en array av storlek $|\Sigma|$ med pekare till barnnoderna. I varje nod kan då uppslagning av efterkommande nod göras snabbt, oavsett hur stort alfabetet är, så uppslag av ett m tecken långt ord går i tid $\Theta(m)$. Insättning av ett ord av längd m går i detta fall i tid $\Theta(m)$.

Om minnessnålhet prioriteras kan det vara mer lämpligt med dynamiska arrayer i noderna. Detta låter oss i varje nod lagra precis så många barnreferenser som behövs, istället för en barnreferens för varje tecken i alfabetet. Dessa arrayer hålls med fördel sorterade för snabbare uppslag. Skillnaden i tidsåtgång (och minnesåtgång) skiljer sig endast från det föregående fallet med en konsant faktor, så tidskomplexiteten är oförändrad.

3.4.2 Suffixträd

Om vi skall söka efter flera strängar P_i i T, så kan det löna sig att först konstruera ett suffixträd över T och sedan för varje P_i söka i suffixträdet. Själva suffixträdet är helt enkelt en trie innehållandes alla suffix till T. Om P_i är en delsträng till T så måste P_i vara prefix till något av T's suffix och kan därför i tid $\Theta(m)$ slås upp i suffixträdet. Själva konstruktionen görs naivt i tid $\Theta(n^2)$ men kan göras i tid $\Theta(n)$ (se litteratur för detaljer).

3.4.3 Suffixarray

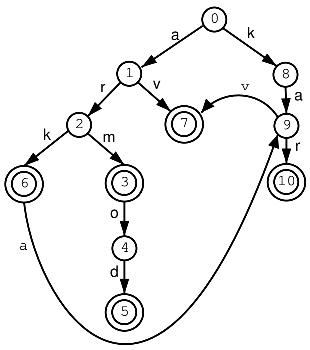
Ett alternativ till att konstruera suffixträd är att använda en suffixarray, vilket är en array vars element är alla suffix till T (eller referenser till dessa) sorterade i lexikografisk ordning. På samma sätt som för suffixträdet kan förekomster av ett mönster P i T finnas genom att söka efter de element i arrayen vars prefix är P. Sökningen kan göras genom binärsökning i tid $O(m \log n)$ där m och n är mönstret respektive textens storlek. Suffixarrayen kan konstrueras i tid $\Theta(n)$, t ex genom att först bygga ett suffixträd och sedan söka genom detta i lexikografisk ordning. I praktiken är det ofta, bland annat på grund av höga konstantfaktorer, bättre att använda andra metoder.

Den naiva ansatsen vore att sortera suffixen med en vanlig jämförelsebaserad algoritm. Eftersom suffixens längd är $O(n)$ blir då tidskomplexiteten $O(n^2 \log n)$. Genom att utnyttja att varje suffix till ett suffix förekommer som prefix till något annat suffix, kan vi relativt enkelt förbättra den naiva ansatsen.

Låt k gå från 0 till $\log n$ och sortera i varje steg suffixen med avseende på de första 2^k tecknen. I den första iterationen inspekterar vi varje suffix första tecken och sorterar efter dessa. I efterföljande iterationer utnyttjar vi att den första hälften av de 2^k tecknen som skall ses till redan är i ordning och att de återstående 2^{k-1} tecknen är prefix till något annat suffix och som sådant har ordnats i föregående iteration. Sorteringssteget kan reduceras till ett konstant antal jämförelser för varje suffixpar och kan alltså göras i tid $O(n \log n)$. Sorteringen utförs i alla de $\log n$ iterationerna, så tidskomplexiteten blir $O(n(\log n)^2)$.

3.4.4 Aho–Corasick

Aho–Corasicks algoritm är en generalisering av KMP som söker efter förekomster av flera strängar P_i samtidigt. Algoritmen konstruerar en trie innehållandes P_i, lägger till ett antal återlänkar och behandlar sökträdet som en ändlig automat med starttillstånd i roten och accepterande tillstånd i de noder som är märkta 1. Följande figur illustrerar hur en sådan automat, med en majoritet av övergångarna borttagna, skulle kunna se ut för $P_i \in \{$av,arm,ark,armod,kar$\}$.



I praktiken använder Aho–Corasicks algoritm, liksom KMP, inte den ändliga automaten direkt, utan konstruerar istället en generalisering av KMP:s prefixfunktion.

Kapitel 4

Kombinatorisk sökning

Den här föreläsningen handlade om kombinatoriska sökningar, en typ av problemlösning som ofta resulterar i en totalsökning. Föreläsaren berättade bland annat om spelen Pentatriss och 15-pusslet samt andra planeringsproblem och olika tillvägagångssätt för deras lösningar.

4.1 Innehåll

- Kombinatorisk Sökning
- Sökstrategier
- Heuristik/Approximationer
- Trick

4.2 Kombinatorisk sökning

Kombinatoriska sökproblem räknas allmänt till de svårare problemen att lösa beräkningsmässigt och detta beror på att lösningarna ofta resulterar i uttömmande sökningar. Typiskt för dessa problem är att man söker efter något kombinatoriskt eller matematisk objekt med vissa egenskaper.

- M-färgning av en graf
- Kappsäcksproblemet med önskad optimal lösning.
- Rubiks kub (en kub uppdelad i sektioner där man vrider sidorna för att få samma färg på var sida.)

Gemensamt för dessa och andra typer av kombinatoriska sökproblem är att det ofta inte finns någon effektiv lösning till problemen och då får man försöka så gott man kan.

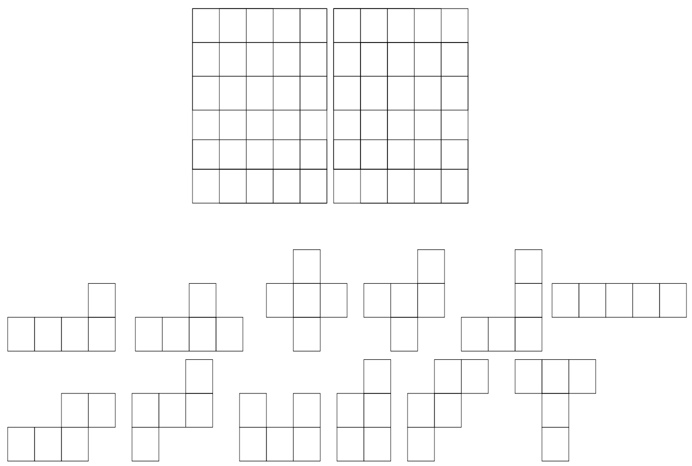
För små instanser går problemen snabbt att lösa med en totalsökning men när instanserna växer blir det svårare och för stora instanser kommer det att ta lång tid. Därför är det viktigt att vara noggrann med den implementation man väljer och det finns vissa saker man bör tänka på.

- Undvik att utforska fruktlösa dellösningar.
- Välj så bra algoritm som möjligt. Det är skillnad på $n!$, 2^n och $2^{\frac{n}{2}}$.
  - $n!$ går att lösa i rimlig tid med $n = 12$ då: $12! = 4.8 \cdot 10^8$.
  - 2^n ger oss bättre resultat, $n = 24$: $2^{29} = 5.4 \cdot 10^8$
  - $2^{\frac{n}{2}}$ kan vi lösa med $n = 58$: $5.4 \cdot 10^8$

4.3 PentaTriss

Pentatriss är ett spel där man skall pussla ihop bitar som tillsammans bildar ett bräde med två delar av storleken 5×6 rutor. Varje bit består av 5 rutor ihopsatta på olika sätt och för att pussla ihop spelet får man vrida och vända på bitarna hur man vill.

I spelet kan man avgränsa totalsökningen för probleminstansen genom att iaktaga att storleken av varje ny samling av lediga rutor som erhålles när man placerar in en pusselbit på brädet måste vara delbart med 5 efterssom alla pusselbitar täcker 5 rutor var. Dessutom kan man undvika att testa fall som redan testats genom att inse att vissa av pusselbitarna har symmetrisk form (Exempelvis pusselbiten som ser ut som ett kors, som varken behöver roteras eller speglas).



Figur 4.1: En bild föreställande alla olika bitar i ett Pentatriss.

För att kunna beräkna alla fall så snabbt som möjligt behövs ett sätt att snabbt kunna lägga till och ta bort pusselbitar och att kontrollera om en bit får plats på brädet på en viss position. Detta kan uppnås genom 'bitmasker' där de två 5×6 rutorna i brädet och även pusselbitarna kan representeras av 30 bitar expemelvis i en integer. Man låter då de första 6 bitarna representera varje ruta i översta raden och följande 6 bitar i andra raden osv. En etta betyder att rutan är upptagen och en nolla att den är ledig. Sedan kan man hitta snabba bitoperationer för att kontrollera när en pusselbit kan läggas till eller ej.

4.4 Sökstrategier

4.4.1 Planeringsproblem

För planeringsproblem finns olika kriterier som man ska tänka på när man bestämmer hur ens kombinatoriska problemet ska lösas.

- Det finns oftast en startposition från vilken man utgår.
- Vill nå en målkonfiguration, men det kan eventuellt finnas flera möjliga.
- Ska kunna påverka konfigurationen med enkla operationer.
- Ibland vill man optimera vägen dit, ibland bara hitta målet.

Som exempel på planeringproblem kan nämnas det klassiska spelet 15-pusslet som består av 15 siffror på brickor på ett bräde tillsammans med en tom ruta. Målet för spelet är att flytta dessa brickor så att en viss siffersekvens bildas på brädet. Men man får bara flytta brickor till den tomma platsen och dessa brickor måste angränsa denna för att få flyttas dit.

4.4.2 8-pusslet

En mindre version av 15-pusslet är 8-pusslet med 8 brickor och en tom ruta. Antalet möjliga konfigurationer för spelet är $\frac{9!}{2}$. Där är sökrymden tillräckligt liten för att den skall vara beräkningsbar i rimlig tid. Så hur går vi nu till väga för att lösa problemet?

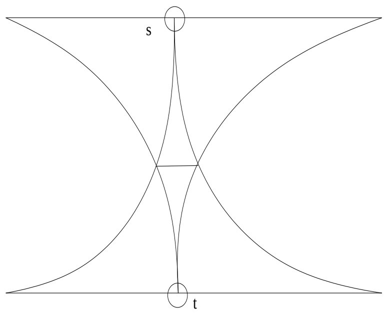
Vad man kan göra är att använda sig av ett sökträd och genom att söka igenom detta försöka hitta lösningar till problemet. Det vi har är ett träd där varje nod är nåbar från rot-noden och representeras av ett tillståd på brädet och där varje kant representerar den handling vi utförde för att komma dit från dess förälder. Sättet som trädet söks igenom beror på val av sökalgoritm som i sin tur beror på vilka fördelar man vill ha respektive nackdelar man får.

- BFS - Breadth First Search. BFS söker igenom hela trädet och expanderar alla noder på samma nivå. Nackdelen med BFS är att man riskerar att få en alltför stor kö och många noder att hålla reda på, vilket leder till stor minnesanvändning. BFS garanterar dock en optimal lösning.

- DFS - Depth First Search. Söker på djupet i sökträdet. Använder betydligt mindre minne men är inte optimal. Man kan således råka hitta en lösning långt ned i trädets grenar som är mycket sämre än den bästa.

- ID-DFS - Iterative Deepening Depth First Search. Är en variant av DFS som begränsar djupet i en vanlig DFS. Sedan ökas djupet i omgångar tills man nått tillräckligt djupt och en lösning kan hittas. Detta kan verka lite långsamt men faktum är att i ett sökträd finns ofta de flesta noderna långt ned i trädet. Genom detta kommer vi hitta en optimal lösning utan att behöva undersöka noder långt ned i grenar i trädet som kan innehålla dåliga lösningar. Komplexiteten blir då $\theta(b^d)$ där b är greningsfaktorn och d är djupet.

Det går att starta en sökning från starttillståndet, måltillståndet eller från båda hållen så att de möts på mitten, även kallat Bidirectional Search. Fördelen med att söka från båda hållen är att man minskar den totala mängden noder som måste genereras. Komplexiteten för dubbelriktad sökning, om greningsfaktorn är b och djupet är d, blir då $\theta(2 * b^{\frac{d}{2}}) = \theta(2 * b^{\frac{d}{2}})$ som med andra ord är snabbare än en sökning från bara ena hållet med kvadratroten. En nackdel kan vara att vi med en bidirektionell sökning måste kontrollera ifall vi har stött på någon nod från andra sidan så att man vet när man har krockat på mitten.



Figur 4.2: Dubbelriktad sökning, man söker från båda håll tills man möts. Den sökrymd man nu behöver kolla begränsar sig till att endast vara överlappningen mellan den över och undre sökrymden på bilden.

4.5 Heuristiker

Vid en heuristisk sökning använder man en skattning av kostnaden att nå målet i ett sökningsproblem från en viss plats där man befinner sig. För sökning i trädstrukturer kan heuristiker användas för att välja bra vägar längs grenar i sökträdet och på så sätt undvika grenar där det kan finnas dåliga lösningar. Vilken heuristik man väljer beror på problemet, enda kravet är att heuristiken **inte** får överskatta kostnaden för att nå en lösning. Det vill säga, heuristiken får tro att det finns en bättre lösning längs en viss stig än vad som sedan visar sig vara fallet. Det finns flera algoritmer som använder sig av heuristik för att hitta lämpliga vägar, exempelvis:

- Best First Search
- Branch-and-Bound

4.5.1 Best-First-Search

I Best First Search väljs varje gång den bästa vägen som heuristiken säger åt den att ta. Detta beskrivs genom evalueringsfunktionen f där det bästa valet för att nå målet bestäms av den heuristik som används på den plats x där man befinner sig.

A*

Ett exempel på en Best-First-Search algoritm är A* vars evalueringsfunktion räknar med antal steg man gått från starten till den plats där man för tillfället befinner sig g, plus det uppskattade antalet steg kvar till målet h. Evalueringsfunktionen får då utseeendet:

$f(x) = g(x) + h(x)$ // summan av tillryggalagd väg och väg kvar till målet.
$g(x) \geq dist(start, x)$ // g är en överskattning
$h(x) \leq dist(x, goal)$ // h är en underskattning

Problematik

Ett problem som kan uppstå om man använder denna algoritm är att man riskerar att få en stor kö Q, exempelvis om vi har en stor sökrymd.

Algoritm 7: A* search algoritmen
Input: Ett sökträd bestående av noder och barn.
Output: Reultatet i form av success, failure.

```
A*(Ett träd)
    Q ← {start}
    while Q ≠ {}
        finn x ∈ Q som har minst f(x)
        if isGoal(x)
            process(x)
            return success
        else
            L ← getChildren(x)
            foreach y ∈ L
                if y obesökt
                    markera y besökt
                    lägg y till Q
    return failure
```

IDA*, Iterative deepening A*

Det kan vara rimligt att ha ett bestämt träddjup och att man sedan med en iterativt ökande algoritm ändrar djupet efter hur heuristiken ändras. Vad som skiljer algoritmen nedan med ID-DFS är att IDA* ibland ökar maxdjupet vi får besöka i trädet med mer än 1, och i en jämförelse med A* slipper vi nu också administrera en kö Q.

15-Spelet med IDA*

Vad skall man ha för heuristik $h(x)$ till 15-spelet? Man bör välja mellan snabb beräkning och god approximation.

- $h(x) = \sum_{i=1}^{15} manhattanavst.$ för bricka i från aktuell plats till rätt plats
- $h(x) = 0$ omm x är målet.

Om den algoritm man testade blev långsam kan man testa att:

- Ställa upp delmål, ex försök klara första raden i 15-spelet, sedan nästa osv. Här offrar man optimalitet men sparar mycket tid.
- Undvik samma konfiguration flera gånger längs samma stig upp till rot-noden. I det här fallet visar det sig vara mer effektivt att enbart se till att man inte går tillbaka till samma nod som man just kom ifrån istället för att kolla hela vägen upp till roten. Detta kan innebära att

Algoritm 8: IDA* search algoritmen
Input: x roten till ett sökträd bestående av noder och barn, *depth* det djup i trädet vi gått hittils, *limit* en gräns satt för hur djupt vi max vill gå.
Output: Reultatet i form av success -1, eller den ökning *newlimit* vi måste göra för att nå målet enligt vår heuristik.
IDA*$(x, depth, limit)$

$newlimit \leftarrow depth + h(x)$ // g(x) + h(x)
if $\text{ISGOAL}(x)$
  $\text{PROCESS}(x)$ **return** -1 // vi är klara och nöjda.
else if $newlimit \leq limit$
  $L \leftarrow \text{GETCHILDEREN}(x)$
  foreach $y \in L$
    $tmp \leftarrow \text{IDA}*(y, depth + 1, limit)$
    $newlimit \leftarrow \text{MIN}(tmp, newlimit)$
return newlimit

Algoritm 9: Härifrån anropas IDA* algoritmen.

$limit \leftarrow 0$
while $limit \geq 0$
  $limit \leftarrow \text{IDA}*(start, 0, limit)$

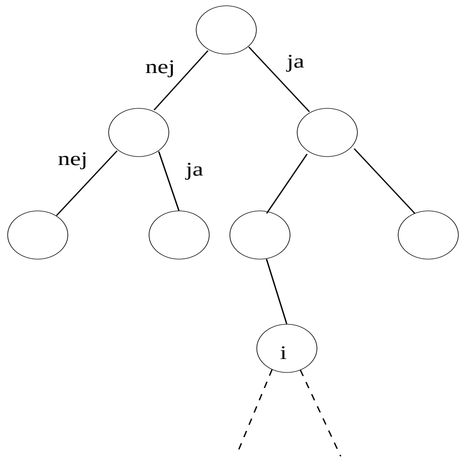
man får upprepade konfigurationer men man tjänar ändå tid eftersom det är mycket snabbare att testa.

- Profilera koden och försök se vad det är som tar mest/mycket tid. Om exempelvis heuristikberäkningen tar tid, försök byta ut denna till en annan som går snabbare. För 15-spelet gäller exempelvis att om x är föregående och x' är aktuell konfiguration så är $h(x')$ lätt att beräkna givet $h(x)$ och senaste draget.

4.5.2 Branch and Bound exempel

Tanken med Branch and Bound är att man kan upptäcka vid en nod i sökrymden, om dess barn i sökningen är värda att undersöka. Detta styrs av ett villkor som talar om utifall vägen kommer kunna ge en bättre lösning än den bästa hittills. Om villkoret ej uppfylls backtrackar man och testar andra vägar. Man kan använda Branch and bound för att beräkna en lösning till kappsäcksproblemet. K är är kappsäckens maximala kapacitet, C hittills packad vikt, V hittills packat värde och M är hittills bästa hittade packning.

Man börjar med att lägga alla saker i en lista och sortera dem så att $\frac{v_1}{w_1} \geq \frac{v_2}{w_2}$ osv. där v_i och w_i är värde respektive vikt för sak i



Figur 4.3: Sökträd vid branch and bound, där varje nivå i trädet motsvarar nästa sak i listan.

Vi börjar då plocka saker ur den sorterade kön, får första saken plats går vi till höger (säger ja), får nästa sak plats fortsätter vi till höger ända tills vi hittar en sak som ej får plats då går vi till vänster (säger nej). Om $\frac{vi}{wi}(K - C) + V \leq M$ kan vi strunta i noden eftersom vägen ej kan ge bättre resultat än M. När vi sökt igenom alla vägar så är M värdet av den bästa packningen.

Detta ger en totalsökning men man kommer ofta att kunna skära bort ganska många vägar eftersom villkoret vi jämför med ofta kommer bli falskt långt upp i trädet och på det sättet kan branch and bound effektivisera sökningen en hel del.

Kapitel 5

Grafer Del 1

Den här föreläsningen var den första av två om grafer och grafalgoritmer. Det som togs upp var olika sätt att representera grafer samt algoritmer för att finna minimala spännande träd, starkt sammanhängande komponenter, och minsta avstånd i en graf.

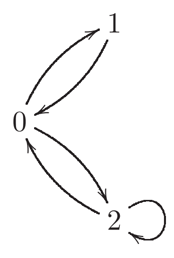
5.1 Grafer

En graf G är en tupel (V, E), där V är en mängd av noder och E en mängd av par $\{(u, v) : u, v \in V\}$ som betecknar kanter mellan noderna.

5.2 Representation

5.2.1 Grannmatris

En grannmatris är en matris $A = \{a_{ij}\}$ där $a_{ij} = 1$ om $(i, j) \in E$, annars 0.



$$\begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

$\boxed{0}$ – $[(0,1),(0,2)]$
$\boxed{1}$ – $[(1,0)]$
$\boxed{2}$ – $[(2,0),(2,2)]$

Figur 5.1: En riktad graf och dess representation som grannmatris respektive kantlistor.

5.2.2 Kantlistor

För varje hörn $v \in V$ finns en lista $adj[v]$ av alla kanter ut från v. D.v.s.

$$(u, v) \in E \Leftrightarrow (u, v) \in adj[u]$$

5.2.3 Jämförelse

Grannmatriser har fördelen att de är marginellt lättare att implementera. Kantlistor blir å andra sidan mindre för glesa grafer vilket gör att de tar mindre minne och kan vara snabbare att söka igenom. Det finns fall när man kan vilja använda båda representationer, t.ex. om man vill filtrera bort multipla kanter i indata, vilket lätt görs med en grannmatris, men sedan vill använda en algoritm som fungerar bättre med kantlistor.

5.3 Oriktade grafer

Oftast kan vi omvandla en oriktad kant (u, v) till två riktade kanter (u, v) och (v, u). Detta fungerar dock inte för alla algoritmer (t.ex. Eulercykel, där vi går längs alla kanter).

5.4 Vanliga fallgropar

Vanliga fallgropar när man löser grafproblem är man missar att tänka på självloopar, multipla kanter och osammanhängande grafer. Skriver man i C++ bör man tänka på att maps av typen

```
map<pair<int,int>,int>
```

inte är lämpliga att använda för grafer med multipla kanter då dessa kanter kommer att skriva över varanda i map:en.

5.5 Grafalgoritmer

5.5.1 Minsta Spännande Träd

Exempel 5.1. *Vi har ett antal namngivna städer, ett antal existerande vägar och ett antal offerter på nya vägar. Vårt problem är att koppla ihop alla städer (indirekt) så billigt som möjligt.*

Modell 5.2. *Vi skapar en graf G med en nod för varje stad, en kant med vikt noll för varje existerande väg och en kant med priset som vikt för varje offert. Vi vill gärna jobba med numrerade noder sådana att $V = \{0, \ldots, n-1\}$. Vi* *kan uppnå detta genom att tilldela varje stadsnamn ett nummer med en index map*[1].

Lösning 5.3. *Problemet är nu att hitta ett Minsta Spännande Träd (MST) i vår graf G och vi kan lösa det med Prims eller Kruskals algoritm, dock går vi bara igenom Prims.*

Algoritm 10: Prims algoritm
Beskrivning: Vi börjar i godtycklig nod, markerar den som besökt och utökar sen trädet med den billigaste kanten som går från en besökt nod till en obesökt.
Input: En graf G med viktade kanter
Output: Minsta Spännande Träd för G

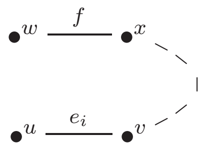
(1) $V = \{0, \ldots, n-1\}$
(2) **foreach** $u \in V$
(3) $P[u] \leftarrow -1$ /* Förälder till u */
(4) $d[u] \leftarrow \infty$ /* Kostnad för att lägga till u */
(5) $vis[u] \leftarrow false$ /* Sann om u är besökt */
(6) $d[0] \leftarrow 0$
(7) **for** $i = 1$ **to** $|V|$
(8)   Hitta en nod u s.a. $vis[u] = false$, $d[u]$ minimal och $d[u] \neq \infty$
(9) $vis[u] \leftarrow true$
(10)   **foreach** $(u, w) \in E$
(11)   **if** not $vis[w]$ and $d[w] > wt(u, w)$
(12) $d[w] \leftarrow wt(u, w)$
(13) $p[w] \leftarrow u$

Det är inte självklart hur vi bör implementera rad (8).

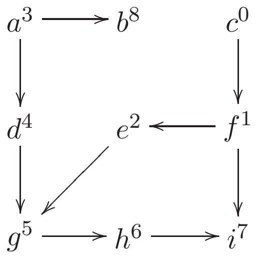
- *(I) Antingen linjärsöker vi genom alla hörn vid (8). Algoritmen går då i tid* $\Theta(|V|^2)$.
- *(II) Smartare är då att i (8) välja första elementet i en prioritetskö med alla obesökta noder u (för vilka* $d[u] \neq \infty$*), sorterade efter ökande kostnad. I (12) måste vi då uppdatera prioritetskön, vilket går att göra i tid* $log|V|$. *Vi får då en tidskomplexitet på* $O(|E|log|V|)$ *för hela algoritmen om vi använder kantlistor*[2].

[1] Vi översätter index till stadsnamn genom att ha en vektor med alla stadsnamn i, en översättning går i konstant tid. För att översätta stadsnamn till index kan vi använda en hashtabell, för vilken översättningar går i förväntad konstant tid, eller en map (binärt sökträd), för vilken översättningar går i $O(log|V|)$. För varje ny stad vi läser in lägger vi till den sist i vektorn.

[2] Notera att om grafen är tät (med $|E| \approx |V|^2$) så är altertaiv (I) snabbare.



Figur 5.2: Noderna u, w och alla noder till vänster om dessa är besökta, noderna v, x och alla noder till höger om dessa är obesökta.



Figur 5.3: Topologisk sortering av noderna i en graf

Bevis. Vi vill visa att det träd som väljs i varje steg kan utökas till ett optimalt träd. Vi visar detta med induktion.
Basfall: Från början finns inga kanter så vi har ett delträd till det optimala.
Induktionsantagande: Mängden $E_{i-1} = \{e_1, \dots, e_{i-1}\}$ av kanter i G kan utökas till ett MST.
Induktionssteg: Vi har valt E_{i-1}, enligt induktionsantagandet finns ett optimalt träd T s.a. $E_{i-1} \subseteq E(T)$, där $E(T)$ är mängden av kanter i T.

- Om $e_i \in T$ är vi klara.
- Annars så innehåller $\{e_i\} \cup T$ en cykel som i sin tur innehåller en kant f som går mellan en besökt och en obesökt nod (se Figur 5.2). Vi har att $wt(f) \geq w(e_i)$, annars skulle algorimen valt f istället för e_i. Då följer att $T' = (T \cup \{e_i\}) \setminus f$ också är ett träd och att $wt(T') \leq wr(T)$ men då T är optimalt har vi att $wt(T') = wt(T)$. Alltså går $E_i = \{e_1, \dots, e_i\}$ att expandera till ett optimalt träd.

□

5.5.2 Topologisk sortering av Riktad acyklisk graf (DAG)

Vi vill tilldela alla noder u ett nummer, $topnr[u]$, som beskriver i vilken ordning vi kan besöka dem om vi utgår från en specifik startnod. Mer specifikt har vi för $u, v \in V$ att $topnr[u], topnr[v] \in \{0, \dots, n-1\}$ s.a. om det existerar en stig från u till v så gäller $topnr[u] < topnr[v]$. Vi löser detta enligt Algoritm 5.5.2.

Algoritm 11: Topologisk sortering
Beskrivning:
Input: En DAG, G
Output: En vektor med nummer för alla noder i G

```
(1)   for u = 0 to n − 1
(2)       vis[u] ← false /* Sant om u är besökt */
(3)       topnr[u] ← ∞ /* Topologisk nummer för u */
(4)   nr ← n − 1
(5)   for u = 0 to n − 1
(6)       if not vis[u]
(7)           nr ← DFS(u, nr, vis, topnr)
(8)
(9)
(10)  DFS(u, nr, vis, topnr)
(11)      vis[u] ← true
(12)      foreach (u, v) ∈ E
(13)          if not vis[v]
(14)              nr ← DFS(v, nr, vis, topnr)
(15)          else
(16)              if topnr[v] = ∞
(17)                  CYKEL!!
(18)      topnr[u] ← nr
(19)      return nr − 1
```

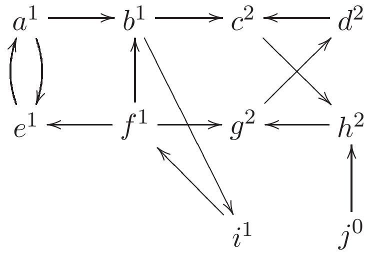
Bevisskiss. För att visa att algoritmen är korrekt måste vi visa att för varje kant (u, v) gäller att $topnr[u] < topnr[v]$. Detta kan vi göra genom att notera följande:

- Använder vi kanten (u, v) i den DFS-sökning som besöker nod v följer det att att $topnr[u] = topnr[v] - 1$.
- Annars så är nod u inte besökt (för då skulle även nod v vara det) och då kommer DFS-sökningen som besöker nod v startas med $nr < topnr[v]$.

Algoritmen har tidskomplexitet $O(|V| + |E|)$ då vi i loopen i (5) fortsätter tills alla noder är besökta, aldrig besöker en redan besökt nod, och för varje nod vi besöker undersöker alla utgående kanter.

5.5.3 Starkt Sammanhängande Komponenter

En starkt sammanhängade komponent (strongly connected component, SCC) C är en maximal delmängd noder s.a. om $u, v \in C$ så finns det en stig $u \rightsquigarrow v$ och en stig $v \rightsquigarrow u$. Att hitta alla SCC:er i en graf $G = (V, E)$ går att göra i



Figur 5.4: En graf med markerade starkt sammanhängande komponenter (markerade 0, 1 och 2)

tid $\Theta(|V| + |E|)$ enligt följande: Först kör vi en toplogisk sorting på grafen, men ignorerar eventuella cykler. Sedan skapar vi grafen G^T som är identisk med G fast med alla kanter vända. D.v.s.

$$\begin{aligned} V(G^T) &= V(G) \\ E(G^T) &= \{(v,u) : (u,v) \in E(G)\} \end{aligned}$$

Till sist loopar vi över alla noder i G^T enligt den tidigare sorteringen och gör, för varje obesökt nod, en DFS-sökning. De noder som besöks i en DFS-sökning kommer vara en unik SCC.
Bevisskiss. Vi visar inte att algoritmen är korrekt men ger ett Lemma som hjälper en bit.

Lemma 5.4. *Låt C, C' vara olika SCC i $G = (V, E)$. Antag att $(u, v) \in E^T = E(G^T)$ där $u \in C$, $v \in C'$. Då kommer första noden i C' före alla noder i C i den topologiska sorteringen.*

Detta medför att C' räknas ut före C, och DFS-sökningen som startas på en nod i C kommer då inte besöka någon nod i C' då de redan är besökta.

5.5.4 Hitta avstånd i en graf

Vi vill hitta det kortaste avståndet i en graf G från en given nod s till alla andra noder. Vi börjar med att betrakta grafer där kanterna har ickenegativa vikter.

Dijkstra

Dijkstras algoritm (Algoritm 5.5.4) för att hitta minsta avstånd i en graf med ickenegativa vikter bygger på Prims algoritm för MST men istället för att i innersta loopen uppdatera kostnader för att lägga till en nod i ett MST uppdaterar vi avståndet till noden.
Algoritmen har samma tidskomplexitet som Prims algoritm (d.v.s $O(|V|^2)$

eller $O(|E|log|V|)$ beroende på implementation), då funktionen RELAX, precis som rad (12) i Prim kan kräva en uppdatering av en prioritetskö vilket tar tid $O(log|V|)$.
Vi kan visa att algoritmen fungerar med induktion på de besökta noderna.

Bevis. För varje nod v kommer avståndet till noden fixeras först när noden är besökt, vi måste alltså visa att ingen nod kan besökas medan den fortfarande kan få ett lägre avstånd tilldelat.
Detta kan visas med induktion enligt följande:
Basfall: Noden s ligger alltid på avstånd 0 från sig själv och besöks alltid först, alltså är dess avstånd fixerat när den besöks.
Induktionsantagande: Låt $l(u \rightsquigarrow v)$ beteckna längden av kortaste stigen från nod u till nod v. Låt u vara den obesökta nod som har minimalt $d[u]$. För alla noder v för vilka $l(s \rightsquigarrow v)$ är mindre än $l(s \rightsquigarrow u)$, gäller att v är besökt och att $d[v]$ därmed är fixerat till just $l(s \rightsquigarrow v)$. För alla noder w, som har en ingående kant från en besökt nod, har vi dessutom att $d[w]$ är satt till det minsta avstådet från s via besökta noder.
Induktionssteg: För att $d[u]$ för den nod u som vi besöker skall uppdateras med ett lägre värde efter att den besökts måste det finnas en obesökt nod $w \neq u$, med en kant (v, w) från en besökt nod v sådan att stigen $s \rightsquigarrow v \rightarrow w$ är optimal bland de stigar från s till w som endast går via besökta noder. Vi har då enligt induktionsantagandet att $d[w] = d[v] + wt(v, w) = l(s \rightsquigarrow v) + wt(v, w)$. Dessutom måste det finnas en stig $w \rightsquigarrow u$ och det måste gälla att $d[u] > l(s \rightsquigarrow v) + wt(v, w) + l(w \rightsquigarrow u)$ men eftersom $d[w] = l(s \rightsquigarrow v) + wt(v, w)$ och $l(w \rightsquigarrow u)$ är positiv har vi att $d[u] > d[w]$ vilket innebär att algoritmen skulle valt w och inte u. Alltså är avståndet till u optimalt.
Fixerar vi $d[u]$ och markerar den som besökt måste induktionsantagandet gälla även för den nod u' som nu har minimalt $d[u']$ bland de obesökta noderna. Det vill säga:

- För alla noder v för vilka $l(s \rightsquigarrow v) < l(s \rightsquigarrow u')$ är v besökt och $d[v] = l(s \rightsquigarrow v)$, detta gäller sen tidigare för alla $v \neq u$ och nu även för u då $d[u]$ är optimalt enligt ovan.

- För alla noder w, där $(v, w) \in E$ är $d[w] = min(\{l(s \rightsquigarrow v) + wt(v, w) : v \text{ besökt}\})$ vilket gäller för alla w, för vilka $(u, w) \notin E$, sen tidigare. För alla w, för vilka $(u, w) \in E$, gäller nu att $d[w] = min(d[u] + wt(u, w), d'[w])$ där $d'[w]$ är det förra värdet på $d[w]$. Om sista noden i den kortaste stigen via besökta noder till w är u så är $d[u]+wt(u, w) < d[w]$ då $d[u]$ är optimal och $d[w]$ är optimal bland de stigar som bara går via $\{v : v \text{ besökt}, v \neq u\}$, $d[w]$ blir då $d[u]+wt(u, w)$ vilket är optimalt då $d[u]$ är det. Annars behåller $d[w]$ sitt värde vilket fortfarande är optimalt.

$\square$

Algoritm 12: Dijkstra
Input: En graf G med viktade kanter och en startnod s
Output: Minsta avståndet från s till alla andra noder i G

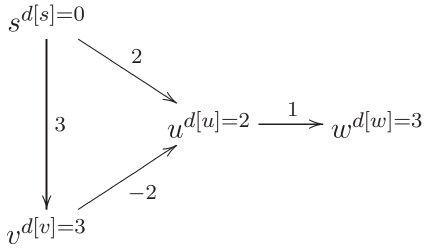
```
V = {0, ..., n − 1}
foreach u ∈ V
    p[u] ← −1 /* Förälder till u */
    d[u] ← ∞ /* Avståndet från s till u */
    vis[u] ← false /* Sann om u är besökt */
d[s] ← 0
for i = 1 to |V|
    Hitta en nod u s.a. vis[u] = false, d[u] minimal
    och d[u] ≠ ∞
    vis[u] ← true
    foreach (u, w) ∈ E
        if not vis[w]
            RELAX(u, w)


RELAX(u,w)
    if d[w] > d[u] + wt(u, w)
        d[w] ← d[u] + wt(u, w)
        p[w] ← u
```

Bellman-Ford

Om vi däremot har negativa vikter på vissa kanter fungerar inte Dijkstra (se Figur 5.5). Då kan vi istället använda Bellman-Ford (Algoritm 5.5.4) som fungerar så länge det inte finns cykler med negativ vikt. Om det finns negativa cykler kan den upptäcka detta. Då vi här inte behöver en prioritetskö (till skillnad från Dijkstra) går RELAX i konstant tid. Vi anropar RELAX $O(|V||E|)$ gånger vilket då också är komplexiteten för hela algoritmen.

Vi vill visa att algoritmen är korrekt.
Bevisskiss. Om $s \to u_1 \to u_2 \to \ldots \to u_r$ är den kortaste stigen till u_r så har vi efter i varv den kortaste stigen till u_i klar. Det högsta antalet kanter i den kortaste stigen till någon nod (utan negativa cykler) är $|V| - 1$.
Har vi en negativ cykel som kan nås från startnoden kan vi alltid uppdatera avstånden längs den cykeln i all oändlighet så om vi, när vi har räknat ut avstånd för alla noder (utan att bry oss om dessa cykler), fortfarande kan minska något avstånd måste vi ha en negativ cykel (under antagandet att algoritmen annars är korrekt). Kan cykeln inte nås från startnoden och alltså ligger på oändligt avstånd, kommer alla uppdateringar att återigen ge



Figur 5.5: Ett exempel på när Dijkstra inte fungerar, $d[u]$ sätts till 2 när den egentligen borde bli 1 och $d[w]$ sätts till 3 när den borde vara 2. Att uppdatera även intilliggande besökta noder när en nod besökts skulle ge rätt värde på $d[u]$ men $d[w]$ skulle behålla sitt felaktiga värde.

samma avstånd[3] och den negativa cykeln kan då alltså inte hittas.

Algoritm 13: Bellman-Ford
Input: En graf G med viktade kanter och en startnod s
Output: Minsta avståndet från en nod s till alla andra noder i G

```
for u = 0 to |V| − 1
    d[u] ← ∞
    p[u] ← −1
d[s] ← 0
for i = 1 to |V| − 1
    foreach (u, w) ∈ E
        RELAX(u,w)

/* Leta efter negativa cykler */
foreach (u, v) ∈ E
    if d[w] > d[u] + wt(u, w)
        NEGATIV CYKEL!!
```

[3]Väljer vi att representera oändlighet som ett stort tal kan vi även detektera onåbara negativa cykler men vi har då istället svårare att skilja oändlighet från ett giltigt värde.

Kapitel 6

Grafer del 2

Den här föreläsningen var den andra av två om grafer och grafalgoritmer. Det som togs upp var Floyd-Warshall och Johnson's algoritmer för att hitta kortaste stigen mellan alla par av hörn i en graf, en algortim för att hitta Eulercykler, Ford-Fulkerssons metod för maxflöde i en graf, och några exempel på tillämpningar av maxflöde för problemlösning.

6.1 Floyd-Warshall

Floyd-Warshall används för att hitta kortaste stigar mellan alla nodpar (x,y) i en graf.

En naiv lösning är att köra $|V|$ st Dijkstra eller Bellman-Ford. Det ger komplexitet $O(|V||E|log(|V|))$ respektive $O(|V|^2|E|)$, men notera att Dijkstra endast fungerar för icke-negativa kantvikter.

Floyd-Warshall är en bättre lösning som bygger på dynamisk programmering.

Definition 6.1. *Låt $D^k_{x,y}$ beteckna minimiavståndet från nod x till nod y där stigen endast får innehålla noderna 0...k-1 förutom start- och slutnoden.*

Låt på samma sätt $P^k_{x,y}$ beteckna den sista noden som besöktes innan slutnoden i en stig med minimalt avstånd.

$D^k_{x,y} \leftarrow \infty$
$D^k_{x,x} \leftarrow 0$
$P^k_{x,y} \leftarrow -1$
foreach $(x, y) \in E$
    $D^0_{x,y} \leftarrow wt(x, y)$
    $P^0_{x,y} \leftarrow x$

Initialisering

Huvudalgoritmen

for $k = 1$ **to** $|V|$
    for $x = 0$ **to** $|V| - 1$
        if $D^{k-1}_{x,k-1} < \infty$
            for $y = 0$ **to** $|V| - 1$
                if $D^{k-1}_{x,k-1} + D^{k-1}_{k-1,y} < D^{k-1}_{x,y}$
                    $D^k_{x,y} \leftarrow D^{k-1}_{x,k-1} + D^{k-1}_{k-1,y}$
                    $P^k_{x,y} \leftarrow P^{k-1}_{k-1,y}$
                else
                    $D^k_{x,y} \leftarrow D^{k-1}_{x,y}$
                    $P^k_{x,y} \leftarrow P^{k-1}_{x,y}$

Analys

Vid första anblick tycks algoritmen kräva tid och minne $\Theta(|V|^3)$, men om vi studerar loopen noggrannare ser vi att endast föregående värde på k används, så om vi bara lagrar nuvarande och föregående värden klarar vi oss med minne $\Theta(|V|^2)$. Faktum är att i just Floyd-Warshalls algoritm kan vi klara oss med att bara lagra ett $D_{x,y}$ och skippa k-indexeringen helt och hållet. Denna typ av lösning är vanlig inom dynamisk programmering, det lönar sig ofta att se hur mycket av information man faktiskt behöver ha kvar i varje steg.

En annan iakttagelse är att vi enkelt kan upptäcka negativa cykler med Floyd-Warshall, om $D^k_{x,x}$ är negativ så ingår x i en negativ cykel.

6.2 Johnsons algoritm

Vi gör en observation: upprepad Dijkstra går i $O(|V||E|log|V|)$, vilket är snabbare än Floyd-Warshalls $O(|V|^3)$ i glesa grafer. En idé är att patcha Dijkstra så att den fungerar för negativa vikter.

Potentialfunktionen

Vi inför en potentialfunktion $\pi : V \to \mathbf{R}$. Sedan viktar vi om alla kanter i grafen enligt $wt'(u,w) = wt(u,w) + \pi(u) - \pi(w)$. För att kontrollera denna lösning kan vi utreda vikten $wt'(P)$ för en stig P.

$$P = u_0, u_1, \ldots, u_k$$

$$wt'(P) = \sum_{i=1}^{k} wt(u_{i-1}, u_i) + \pi(u_{i-1}) - \pi(u_i)$$

Vi ser att detta är en teleskopsumma: alla potentialer utom den första och sista förekommer både som positiv och negativ term, så uttrycket kan förenklas till

$$wt'(P) = wt(P) + \pi(u_0) - \pi(u_k)$$

Eftersom $\pi(u_0)$ och $\pi(u_k)$ är konstanta för alla stigar mellan u_0 och u_k så kommer den nya viktningen ge samma kortaste stig som den gamla.

Så, det kvarstår att hitta en potentialfunktion sådan att $wt(u,w)+\pi(u)-\pi(w)$ är ickenegativ för alla tänkbara kanter. Detta kan åstadkommas genom att välja $\pi(u)$ som avståndet från ett fixt hörn till u. Triangelolikheten ger då

$$\pi(w) \leq \pi(u) + wt(u,w)$$

omskrivning ger

$$wt(u,w) + \pi(u) - \pi(w) \geq 0$$

vilket var precis det som söktes. Så, vilket fixt hörn ska vi använda som utgångspunkt? Det visar sig vara bra att lägga till ett nytt hörn med kanter av längd 0 till alla andra hörn, och använda avståndet till detta nya hörn som π.

Färdig algoritm

(1) $G' \leftarrow$ lägg till hörn s med kanter med avstånd 0 till alla hörn i G.
(2) Räkna ut π m.h.a Bellman-Ford
(3) **if** $\exists$ negativ cykel
(4)   avbryt
(5) **foreach** $v \in V$
(6)   DIJKSTRA(G, v, wt')

6.3 Parentes om Bellman-Ford

Bellman-Ford detekterar som vi vet negativa cykler som kan nås från startnoden, men hur är det om de inte kan nås från startnoden? Då beror det plötsligt på hur oändligheten implementeras. Om vi sätter oändligheten till ett stort tal, så kommer cyklerna ändå att upptäckas när vi gör den vanliga sökningen efter negativa cykler. Om vi däremot specialbehandlar oändligheten så att $\infty - wt(u, w) = \infty$, går cyklerna inte att upptäcka. Det är viktigt att tänka efter när man använder stora tal för att representera oändligheten. De måste vara större än alla riktiga värden vi kan få, men också tillräckligt små för att vi aldrig ska göra någon addition där summan orsakar overflow - det kan leda till elaka buggar.

6.4 Eulercykler

En Eulercykel är en cykel som besöker alla kanter i en graf exakt en gång. Det finns tillräckliga och nödvändiga villkor för existens av Eulercykler i riktade och oriktade grafer. En oriktad graf har en Eulercykel omm $deg(u)$ är jämnt för alla u och grafen är sammanhängande. En riktad graf har en Eulercykel omm $indeg(u) = utdeg(u)$ för alla u och grafen är sammanhängande. Notera här att en graf kan ha en Eulercykel trots att den har isolerade hörn, det är bara kanterna som måste besökas. Alltså, när vi kräver att grafen är sammanhängande, bortser vi från isolerade hörn.

Så, hur hittar vi en Eulercykel? En enkel algoritm består av två delar, en stigfinnare och en kontrollant.

Stigfinnare

Starta i godtyckligt hörn u. Traversera grafen så långt det går, utan att passera samma kant flera gånger. När det tar stopp är vi tillbaka i u, men det kan finnas obesökta kanter.

Kontrollant

Följ stigen från u som stigfinnaren har hittat tills vi kommer till ett hörn w med en obesökt kant. Starta stigfinnaren från w, för att hitta en stig av obesökta kanter som leder tillbaka till w. Skjut in hela den nya stigen där w besöktes i den gamla. Upprepa sökningen på samma sätt från w, längs den nya stigen. När vi kommer tillbaka till u är vi klara.

6.5 Flöde och Ford-Fulkerson

Vi tänker oss en uppgift som går ut på att ställa upp vägspärrar mellan Stockholm och Göteborg för att stoppa all biltrafik. Vi vill veta minimalt antal vägspärrar som behöver placeras ut, och var de ska ställas.

För att ta reda på hur många vägspärrar som krävs kan man representera vägnätet med en graf och låta alla kanter ha kapacitet 1. Vi vill sedan ta reda på vad det maximala flödet genom grafen från Stockholm till Göteborg är, och hur man uppnår det. Att lösa maxflödesproblemet kan man göra med hjälp av en metod som kallas Ford-Fulkerson.

Ford-Fulkerson

Grundién i Ford-Fulkersson är ganska enkel

(1) Sätt flödet till 0
(2) **while** $\exists$ utökande stig
(3) öka flödet längs stigen

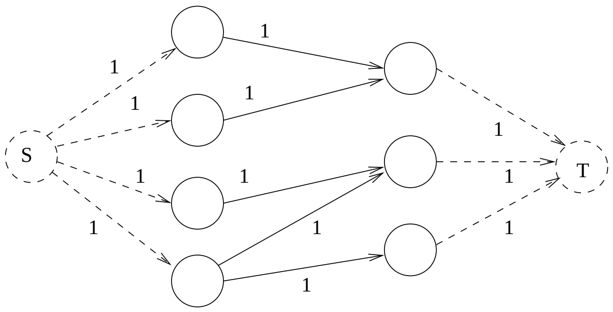
En utökande stig är en stig från starthörnet till sluthörnet där vi kan få igenom ett flöde. När vi letar efter stigen använder vi oss av en residualflödesgraf, där residualkapaciteten är ursprunliga kapaciteten $-$ nuvarande flödet. Vi ser också till att $f(u, w) = -f(w, u)$, alltså att vi har negativa flöden åt motsatt riktning, för att representera att vi kan "ångra" det flödet som tidigare gått genom kanten. För att åstadkomma detta inför vi för varje kant (u, w) en kant (w, u) med kapacitet 0, om det inte redan fanns en kant (w, u). När vi letar efter en utökande stig måste alltså alla ingående kanter ha en positiv residualkapacitet. Ett bra sätt att hitta utökande stigar är med en vanlig breddenförst-sökning, det kallas Edmonds-Karps algoritm. Det ger komplexitet $O(|V||E|^2)$. Om man istället använder en djupetförst-sökning riskerar man att få en väldigt dålig körtid på elaka exempel.

Var sätter vi vägspärrarna?

Vi börjar med att göra en sökning från startnoden i residualflödesgrafen, och markerar alla hörn vi når. Startnoden är nu markerad, medan slutnoden inte är det, eftersom vi då skulle ha hittat en utvidgande stig. Sedan sätter vi vägspärrar på alla vägar som går från markerade noder till omarkerade. Flödet genom vägarna vi nu har spärrat måste ha varit lika med kapaciteten på vägarna, annars hade vi kunnat ta oss genom kanten i residualflödesgrafen och markerat båda ändarna. Flödet längs de spärrade vägarna kan inte heller vara större än flödet från s till t, eftersom varje flöde från s till t går över från markerade till omarkerade noder högst en gång (om det går flöde från u till w och w är markerad så är även u markerad, eftersom vi kan gå baklänges längst flödet i residualgrafen).

6.6 Flödesexempel: Bipartit matchning

En bipartit graf är en oriktad graf sådan att hörnen kan delas in i två disjunkta mängder så att alla kanter i grafen har ett hörn i varje mängd. En matchning är ett val av kanter sådant att max en utvald kant går till ett givet



Figur 6.1: Illustration av bipartit matchning med hjälp av flöde.

hörn. Vi kan hitta en maximal bipartit matchning i en graf genom att ge alla kanter kapacitet 1 och rikta dem från ena mängden till den andra. Sedan lägger vi till ett hörn som är start och ett hörn som är mål, och sätta kanter från starten till alla hörn i ena mängden, och kanter från alla hörn i andra mängden till målet. De ursprunliga kanterna som har ett positivt flöde ingår i en maximal matchning.

Om kanterna dessutom har en kostnad kan vi hitta en maximal matchning av minimal kostnad om vi hela tiden hittar den billigaste utökande stigen i Ford-Fulkersson. Eftersom kantvikterna kan vara negativa görs detta lämpligen med Bellman-Fords algoritm.

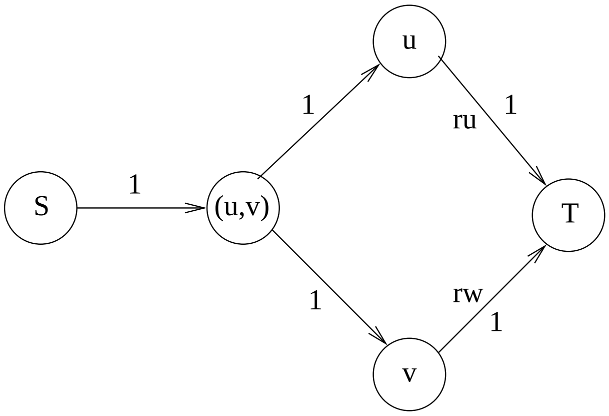
6.7 Flödesxempel: Bus Tour

Det här är ett exempel från NWERC 2003. En buss åker runt i en stad där några gator är enkelriktade, och frågan är om det existerar en Eulercykel. När det finns både oriktade och riktade kanter i en graf har vi inget enkelt sätt att avgöra om en Eulercykel existerar, men vi kan istället se det som att vi ska rikta alla oriktade kanter på ett sådant sätt att det sedan finns lika många in- och utkanter från varje hörn.

Betrakta ett hörn u som har deg_+ kanter ut, deg_- kanter in och deg_0 oriktade kanter. Vi vill rikta r_u kanter in mot hörnet så att

$$r_u + deg_- = (deg_+ + deg_- + deg_0)/2$$

När vi funnit r_u för alla hörn kan vi konstruera en graf för att lösa problemet med hjälp av flöde. Skapa ett hörn i den nya grafen för varje hörn och oriktad kant i den gamla, och låt kanter med kapacitet ett gå från hörnen som representerar kanter till båda hörnen som kan nås från kanten. Låt sedan kanter med kapacitet 1 gå från ett starthörn till alla hörn som representerar kanter. Slutligen, skapa kanter med kapacitet r_u från alla hörn som representerar ett hörn u i ursprungsgrafen till ett sluthörn.



Figur 6.2: Illustration av lösning av Bustour med hjälp av flöde

En enhet av flöde från hörnet (u, w) till hörnet u representerar att kanten (u, w) i ursprungsgrafen riktas mot hörnet u. Om maxflödets storlek är lika med $\sum r_u$ så existerar en Eulercykel i den ursprungliga grafen.

Kapitel 7

Syntaxanalys

7.1 Syntaxanalys

Den här föreläsningen tar upp tekniker som kan behövas för att utföra en syntaxanalys: lexikalanalys, grammatiker, rekursiv medåkning och reguljära uttryck.

7.1.1 Exempel på tillämpning

Beräkna molekylvikten för en molekyl givet dess kemiska formel (och kända atomvikter). Exempelvis har H_2O molekylvikten 18 eftersom H har atomvikt 1 och O har atomvikt 16.

En annan, lite mer komplicerad molekyl som vi kommer att använda vidare i senare exempel är nitroglycerin: $C_3H_5(NO_3)_3$ vilken har molekylvikt 227 (C har atomvikt 12 och N har atomvikt 14).

7.1.2 Lexikalanalys

För att kunna göra en syntaxanalys måste man först göra en lexikalanalys som omvandlar indatan till en ström av *tokens*. I det här fallet är indatan en textsträng, som för nitroglycerin har formen ”C3H5(NO3)3$”. $ symboliserar här ”slut på indata” och kan exempelvis vara en radbrytning.

Genom att analysera indatan kan man komma fram till att de tokens som förekommer i det här fallet är:

atom en atom (Läses in som en versal bokstav följd av noll eller fler gemener.)

num ett tal.

lpar en vänsterparentes.

rpar en högerparentes.

$ som ovan.

Motsvarande för ett vanligt språk skulle vara att ha ord och skiljetecken som tokens.

Att få in en tokenström som: **atom num atom num lpar atom . . .**

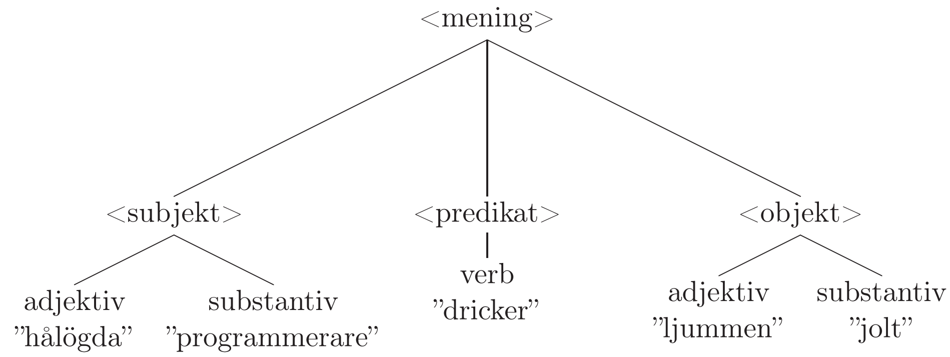
är i det här och många andra fall dock inte tillräckligt. Därför kan man knyta data till tokens som innehåller mer information och istället få något som:

atom	num	atom	num	lpar	atom	...
”C”	”3”	”H”	”5”		”N”	...

För att utföra lexikalanalysen kan man lämpligen använda reguljära uttryck, exempelkod för det kommer senare i anteckningarna.

7.1.3 Grammatiker

En grammatik används för att bygga ett syntaxträd. För en vanlig mening skulle till exempel ett syntaxträd kunna se ut så här:



Orden inom $<\ldots>$ kallas för ickeslutsymboler och orden inom ”. . .” kallas för lexem. Orden som är direkt knutna till ett lexem kallas för slutsymboler. Indatan som trädet byggts av kallas fras.

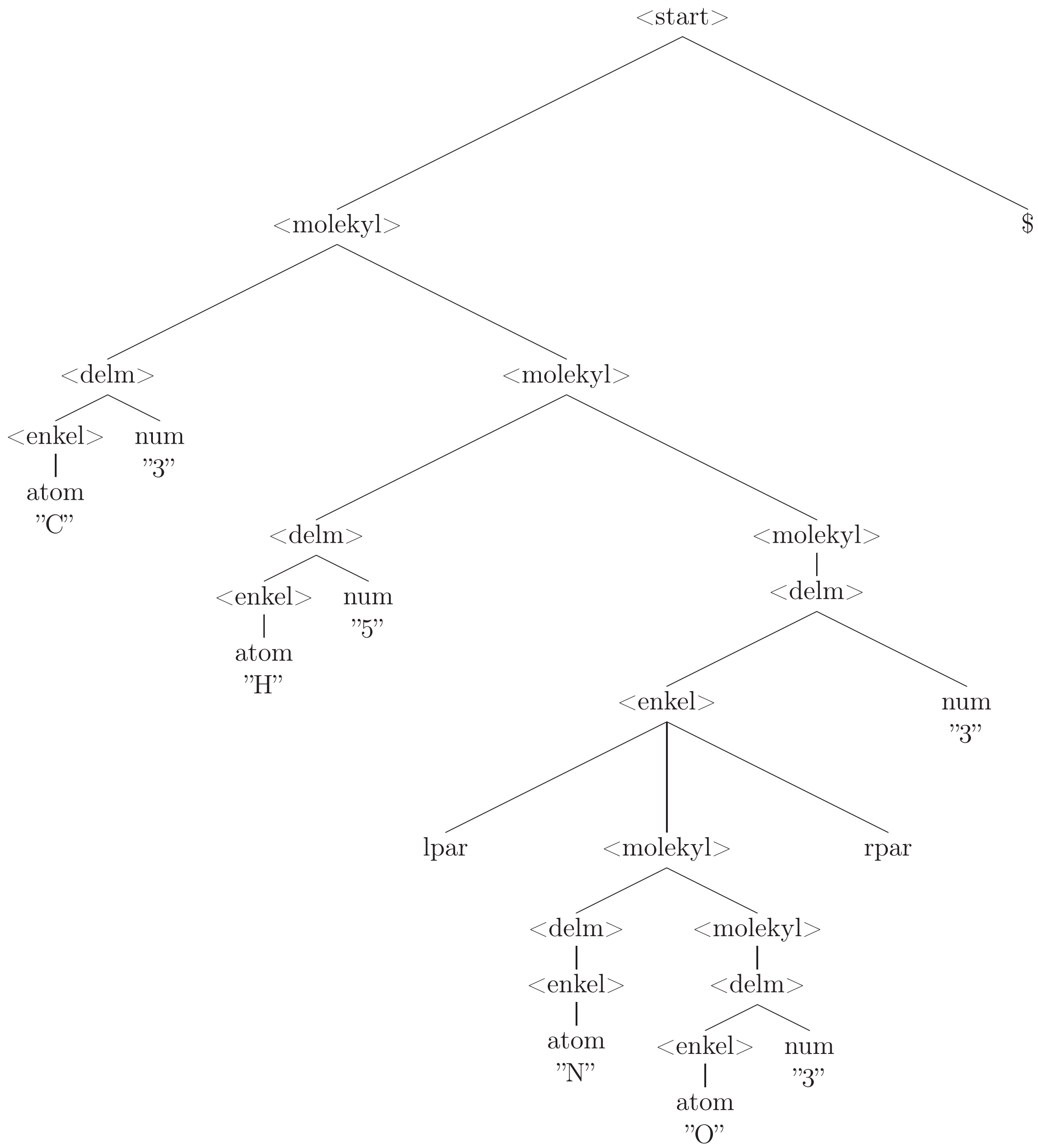
En grammatik har fyra beståndsdelar och kan uttryckas som en tupel $<\Sigma, N, <\mathit{start}>, R>$. Σ är mängden av slutsymboler, vilket är samma sak som kallades tokens i lexikalanalysen. N är mängden av ickeslutsymboler, $<\mathit{start}>$ är startsymbolen $\in N$ (<mening> i exemplet ovan) och R är en uppsättning produktioner (eller regler) för hur ickeslutsymbolerna ser ut.

Produktionerna har formen $<a> ::= \beta \in (\Sigma \cup N)^*$ (där A^* betyder sträng av noll eller flera element ur A).

För molekylerna blir R så här:

<start>	::=	<molekyl>$
<molekyl>	::=	<delm><molekyl>
<molekyl>	::=	<delm>
<delm>	::=	<enkel>num
<delm>	::=	<enkel>
<enkel>	::=	atom
<enkel>	::=	lpar<molekyl>rpar

Syntaxträdet för nitroglycerin ser ut så här:



7.2 Rekursiv medåkning

Rekursiv medåkning (recursive descent) är ett sätt att bygga upp syntaxträd från en grammatik. Det fungerar inte för all grammatiker, men i de fall där det inte fungerar blir det ofta så komplicerat att man rekomenderas att använda ett färdigt verktyg för att generera koden som behövs.

Idén är att man skriver en metod för varje ickeslutsymbol som rekursivt anropar varandra. Den datastruktur som innehåller de tokens som man fick från lexikalanalysen behöver bara funktionalitet för att se ett aktuellt token (`peekToken()`) och för att byta till nästa token (`nextToken()`). För ett token innehåller `type` tokentypen och `val` den data som vi knutit till vissa tokens.

Förutsättningen för att det här ska fungera är att vi, givet vilket token vi står på och vilket som är nästa token, bara kan följa en produktion.

Observera att följande algoritm inte bygger trädet explicit utan beräknar molekylvikten direkt.

START()
$w \leftarrow$ MOLEKYL()
$t \leftarrow$ PEEKTOKEN()
if $t.type \neq \$$
    ERROR()
else
    output w

MOLEKYL()
$w \leftarrow$ DELM()
$t \leftarrow$ PEEKTOKEN()
if $t.type \in \{atom, lpar\}$
    $w \leftarrow$w+ MOLEKYL()
else if $t.type \notin \{rpar, \$\}$
    ERROR()
return w

DELM()
$w \leftarrow$ ENKEL()
$t \leftarrow$ PEEKTOKEN()
if $t.type = num$
    $w \leftarrow w \cdot t.val$
    NEXTTOKEN()
return w

```
ENKEL()
    t ← PEEKTOKEN()
    if t.type = atom
        w ← LOOKUPWEIGHT(t.val)
        NEXTTOKEN()
    else if t.type = lpar
        NEXTTOKEN()
        w ← MOLEKYL()
        t ← PEEKTOKEN()
        if t.type ≠ rpar
            ERROR()
        else
            NEXTTOKEN()
    else
        ERROR()
    return w
```

7.3 Allmäna grammatiker

Anta att vi har vår grammatik G på Chomsky normalfom (CNF). Detta innebär att den endast består av regler på nedanstående former:

<a> ::= x
<a> ::= <b><c>

Grammatiker med regler på annan form går att omvandla till CNF. Exempelvis:

<a> ::= <b>x
↓
<a> ::= <b><ny>
<ny> ::= x

<a> ::= <b><c><d><e>
↓
<a> ::= <b><ny1>
<ny1> ::= <c><ny2>
<ny2> ::= <d><e>

Givet indata $X_1...X_n$, kan G generera $X_1...X_n$ från startsymbolen <a_0>? Genom att använda regler på formen <a_0>::=<a_1><a_2>, kan vi dela upp problemet i delproblem: $X_1...X_i \mid X_{i+1}...X_n$. Där vi vill att <$a_1$> ska matcha första delen och <a_2> andra.

Detta lämpar sig väl att lösas mha dynamisk programmering. För detta behöver vi en tredimensionell boolesk array: gen[j, i, k] ↔ <a_j> kan generera

$X_i ... X_{i+k-1}$. Matrisen blir ofta gles. En idé kan vara att istället göra en rekursiv implementation och memoisera.

DPCNF()
$gen[j,i,k] \leftarrow false \ \forall \ i,j,k$
if $X_i = b$ **and** $\exists$ regel $<a_j>::=b$
    $gen[j,i,1] \leftarrow true$
for $l = 2$ **to** n
    for $i = 0$ **to** $n-1$
        for $r = 1$ **to** $l-1$
            if $\exists$ regel $<a_j>::=<a_s><a_t>$ så att $gen[s,i,r]$ **and** $gen[t,i+r,l-r]$
                $gen[j,i,l] \leftarrow true$
return $gen[0,1,n]$

7.4 Reguljära uttryck

Reguljära uttryck (över alfabetet Σ) skapas av följande:

R		$L_R \subseteq \Sigma^*$
a	$a \in \Sigma$	$L_a = \{a\}$
ε	”tomma strängen”	$L_\varepsilon = \{\varepsilon\}$
$(R_1)\|(R_2)$	union	$L_{R_1} \cup L_{R_2}$
$(R_1)(R_2)$	konkatenering	$L_{(R_1)(R_2)} = \{\alpha\beta \mid \alpha \in L_{R_1}, \beta \in L_{R_2}\}$
$(R)^*$	Kleene-slutning	$L_{(R)^*} = L_\varepsilon \cup L_R \cup L_{(R)(R)} \cup ... \cup L_{(R)^k}$

Av operatorerna ovan har Kleene-slutningen högst precedens, följt av konkatenering och slutligen union. Detta leder till att man ofta kan utelämna många parenteser. Många implementationer av reguljära uttryck tillåter också formler enl. tabellen nedan. Dessa gör inte att man kan representera fler grammatiker med reguljära uttryck, men det gör uttrycken mycket lättare att skriva.

Formel(exempel)	Innebörd
[abc]	Något av a, b eller c
[a-z]	Något av tecknen i intervallet (ofta efter teckenkod)
[ˆabc]	Något tecken som varken är a, b eller c

7.4.1 Reguljära uttryck i Java

Nedan följer ett exempel på hur man kan använda reguljära uttryck i Java.

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;
```

```
import java.util.Scanner;

public class pat2 {
    public static void main(String[] as)
    {
        String atom   = "[A-Z][a-z]?";
        String paren  = "[()]";
        String number = "[1-9][0-9]*";
        String s = as[0];
        Matcher m =
            Pattern.compile(atom + "|" + paren + "|" + number).matcher(s);
        int pos=0;
        while (m.find(pos)) {
            System.out.println(m.group());
            pos=m.end();
        }
    }
}
```

Kapitel 8

Aritmetik och stora heltal

Den här föreläsningen handlar om beräkningar med flyttal och behandling av heltal som inte ryms i grundläggande datatyper.

8.1 Flyttalsaritmetik

Flyttal beter sig inte alltid som man tror.

Exempel 1 - Avrundning

```
double x = 0.29;
int y = (int)(100*x);
print(y); //Skriver ut 28
```

Exempel 2 - Jämförelse

```
for(double x = 0.0; x!=1.0; x+=0.1)
  print("Hello"); //forts\"atter i all o\"andlighet
```

8.1.1 Flyttalsrepresentation

Anledningen till beteendena vi stötte på ovan är att alla reella tal inte kan representeras exakt med grundläggande datatyper. Detta beror på hur flyttalen representeras i minnet.

I C/C++/Java representeras ett flyttal x enligt följande:

$x = S \cdot M \cdot 2^E$

$S -$ teckenbit $\in \{1, -1\}$

$M -$ mantissan (heltal)

$E -$exponenten (heltal) Storleken på mantissan och exponenten varierar med datatypen enligt tabell 1.

flyttalstyp	storlek	S	E	M
float	32	1	8	24
double	64	1	11	53
long double	128	1	15	113

Tabell 8.1: Antal bitar för S, E, M och totalt för olika flyttalstyper. Siffrorna för long double gäller sun-datorerna på Nada och kan variera mellan plattformar.

Som man ser i tabell 1 så är summan av antal bitar för de olika delarna ett mer än totala antalet bitar. Detta beror på att man använder normaliserad representation där den mest signifikanta biten i mantissan alltid är satt. Detta fungerar eftersom man alltid kan justera exponenten för att "kompensera" för en större mantissa. Om den mest signifikanta biten inte skulle vara satt kan man i stället representera talet med $S(2M)2^{E-1}$. Därför behöver inte den högsta biten lagras.

En närmare titt på double

En double x representeras alltså på följande sätt:

$$x = (-1)^s(1 + M \cdot 2^{-52}) \cdot 2^{E-1023}$$

där $0 \leq M < 2^{52}, 0 \leq E < 2^{11}$.

Vissa värden på E är reserverade för speciella tal som inte kan representeras enligt denna formel.

En double med $E = 0$ representerar i stället

$$x = (-1)^s(M \cdot 2^{-52}) \cdot 2^{-1023}$$

där ettan från föregående formel tagits bort för att kunna representera ± 0.

$E = 2047, M = 0$ betyder $\pm\infty$ beroende på s.

I C++ ger med många kompilatorer en varning om man skriver t.ex. 1.0/0.0 för att få doublevärdet för ∞. För att slippa varningen kan man istället skriva

```
std::numeric_limits<double>::infinity()
```

$E = 2047, M \neq 0$ representerar "Not a number", NaN. Detta är t.ex. resultatet av 0.0/0.0 och ∞/∞. NaN har den speciella egenskapen att $NaN == NaN$ är falskt. NaN kan man få utan kompileringsfel genom

```
std::numeric_limits<double>::quiet_NaN()
```

8.1.2 float vs. double

I de allra flesta fall där man behöver flyttal bör man använda double. Det ger en betydligt högre precision är float, men går ändå nästan lika fort att räkna med. Float kan dock vara bra om man har ont om minne, eller om avrundningsfel inte har någon betydelse, som t.ex. i datorgrafiksammanhang.

8.1.3 Varför blir det fel?

Exemplen i början har enkla förklaringar om man känner till datorns flyttalsrepresentation.

Exempel 1

0.29 kan inte representeras exakt med en double. Datorn lagrar då istället det doublevärde som ligger närmast 0.29 vilket råkar vara ungefär $0.28999999999999998 = 0.29 - 2 \cdot 10^{-18}$. När detta multipliceras med 100 och trunkeras till ett heltal får vi då 28.

Exempel 2

0.1 kan inte heller representeras exakt med en double. Det kan däremot 1. När vi har adderat 0.1 tio gånger kommer vi därför få ett tal som inte är exakt 1 och loopens slutkriterium kommer därför inte att uppfyllas.

8.1.4 Hur löser vi problemen?

Det finns ett antal lösningar på problemen i båda exemplen i början.

Exempel 1

Lösning 1: Avrunda till närmsta istället för nedåt vid omvandling till heltal. Detta görs enklast genom `y=(int)(x*100+0.5)`. Detta funkar om x inte är ohyggligt stort.
Lösning 2: Skriv ut talet som en double med 0 decimaler.
Lösning 3: Problemet uppstår normalt vid inläsning av flyttal. Då kan man istället läsa in talet som heltal, punkt, heltal. Tänk dock på att $5.01 \neq 5.1 = 5.10$.

Exempel 2

Lösning: Använd ej $a == b$ för att kontrollera likhet mellan a och b. Använd istället $|a-b| < \epsilon$ för något litet ϵ (absolut jämförelse) eller $|a-b| < \epsilon(|a|+|b|)$ (relativ jämförelse). Absolut jämförelse fungerar dåligt för mycket stora tal a och b eftersom differensen mellan talen då kan vara stor trots att den är relativt obetydlig. Relativ jämförelse fungerar dåligt för mycket små tal a

och b eftersom . Vill man vara helt säker kan man använda båda. Då ska man anse det som likhet om någon av relativ eller absolut jämförelse ger likhet. Välj lämpligen $10^{-13} \leq \epsilon \leq 10^{-7}$. $\leq, <, \geq, >$ kan i vissa fall behöva implementeras på motsvarande sätt.

8.1.5 Flyttal vs. heltal

I de flesta fall där man kan välja bör man använda heltal istället för flyttal, eftersom man då slipper avrundningsfel. Det finns dock vissa fall då flyttal har sina fördelar.

Exempel: Finn $\sqrt[4]{x}$ givet att detta är ett heltal.

Lösning: Läs in x som en double, låt $y = \text{pow}(x, 0.25)$ och skriv ut y avrundat till 0 decimaler.

Motivering: Låt $X \approx x \cdot (1 \pm \epsilon)$ vara vår approximation av (d.v.s. det double-värde som ligger närmast) x. Låt

$$\begin{aligned}
Y &= \text{pow}(X, 0.25) \\
&= (1 \pm \epsilon) \cdot X^{1/4} \\
&= (1 \pm \epsilon) \cdot e^{\ln(X)/4} \\
&\approx (1 \pm \epsilon) \cdot e^{\frac{\ln(x)}{4} \cdot (1 \pm \epsilon)} \\
&= (1 \pm \epsilon) \cdot y e^{\epsilon \cdot \ln(y)} \\
&\approx (1 \pm \epsilon)(1 \pm \epsilon \cdot \ln(y)) y \\
&\approx y \pm y \cdot \epsilon \cdot \ln(y).
\end{aligned}$$

I näst sista steget används att $e^w = 1 + w + \frac{w^2}{2} + ...$ och $w \approx 0 \Rightarrow e^w \approx 1 + w$. Vi får då ett fel i vår beräkning som är $|Y - y| \approx y \cdot \epsilon \cdot \ln(y) \ll 0.5$ vilket är vad som krävs för att få korrekt svar efter avrundning.

8.1.6 Mer information

Mer utförlig information om flyttalshantering i datorer finns i följande artikel:

`http://docs.sun.com/source/806-3568/ncg_goldberg.html`.

8.2 Floor och Ceil

Givet $x \in \mathbb{R}$ defineras
$\lfloor x \rfloor = \text{floor}(x)$ som x avrundat till närmsta heltal nedåt samt
$\lceil x \rceil = \text{ceil}(x)$ som x avrundat till närmsta helatal uppåt.

Tabell 8.2: Exempel på ceil och floor

x	$\lfloor x \rfloor$	$\lceil x \rceil$
5	5	5
5.3	5	6
-0.7	-1	0

8.2.1 Operationer

$$\lfloor x \rfloor = n \iff n \leq x < n + 1$$

$$\lceil x \rceil = n \iff n - 1 < x \leq n$$

8.2.2 Identiteter

$$-\lfloor x \rfloor = \lceil -x \rceil$$
$$-\lceil x \rceil = \lfloor -x \rfloor$$
$$\lfloor x \pm n \rfloor = \lfloor x \rfloor \pm n$$
$$\lceil x \pm n \rceil = \lceil x \rceil \pm n$$
$$\left\lceil \frac{n}{m} \right\rceil = \left\lfloor \frac{n + m + 1}{m} \right\rfloor \text{ för } m > 0$$

8.2.3 Beräkning av $\left\lfloor \frac{n}{m} \right\rfloor$ och $\left\lceil \frac{n}{m} \right\rceil$

Om m, n inte är för stora är det enklast att konvertera talen till doubles och använda floor()- eller ceil()-funktionerna.

Genom att använda identiteterna ovan kan man skriva mer robusta funktioner för floor och ceil:

```
floor(int n, int m)
if m<0 then
   return floor(-n, -m)
else if n<0 then
   return -ceil(-n, m)
else
   return n/m
end if
```

```
ceil(int n, int m)
if m>0 then
   return floor(n+m-1, m)
else
   return floor(-n-m-1,-m)
end if
```

8.2.4 Rare easy problem

Kattisproblemet rare easy problem (http://kattis.csc.kth.se/problem?id=easy) har en enkel explicit lösning som kan härledas med ceil och floor. Problemet går ut på att finna N givet $K = M - N$ där M är N med sista siffran borttagen.

Vi börjar med att observera $M = \lfloor N/10 \rfloor$.
Vi vill alltså lösa $N - \lfloor N/10 \rfloor = K$

$$\begin{aligned}
N - \lfloor N/10 \rfloor = K \quad &\Rightarrow \quad K = N + \lceil -N/10 \rceil \\
&\Rightarrow \quad K = \lceil 9N/10 \rceil \\
&\Rightarrow \quad K - 1 < 9N/10 \leq K \\
&\Rightarrow \quad 10K - 9 \leq 9N \leq 10K \\
&\Rightarrow \quad 10K/9 - 1 \leq N \leq 10K/9 \\
&\Rightarrow \quad \lceil (10K/9 - 1) \rceil \leq N \leq \lfloor 10K/9 \rfloor
\end{aligned}$$

Om K är delbart med 9 är lösningarna $10K/9$ och $10K/9-1$ annars $\lfloor 10K/9 \rfloor$.

8.3 Stora heltal

Stora heltal är heltal som inte får plats i standard-datatyper som long i Java och long long i C/C++. I Java finns BigInteger för att hantera dessa och i C/C++ finns GMP (GNU Multiprecision Library), men den senare finns oftast ej tillgänglig på Kattis.

8.3.1 Representation

Ett stort heltal x kan representeras i basen b som

$$x = x_{n-1}b^{n-1} + ... + x_1 b + x_0$$

där $0 \leq x_i < b$. Talen $x = (x_i)$ kan lagras som en vektor av tal.

Olika val av b kan vara fördelaktiga i olika sammanhang.

- **Potens av 2 t.ex.** $2^{32}, 2^{64}$ **eller** 2^{16}**.**

Detta är mest effektivt eftersom man utnyttjar minnesutrymmet väl och det då blir få x_i. Dessutom kan division- och modulo-operationerna som krävs i nedanstående algoritmer utföras effektivare med dessa baser eftersom det är den interna representationen i datorn. En nackdel är att det blir jobbigt med inläsning och utskrift i basen 10.

- $b = 10$

Detta är enkelt men ineffektivt då det kräver många x_i och därmed mycket minnesutrymme och fler beräkningar än med större b.

- $b = 10^d$

Bra kompromiss för enkel kod då det lätt går att skriva ut och mata in tal i basen 10 samtidigt som minnet utnyttjas ganska väl. T.ex. d = 9 och vi har 2^{64} long long. Vill ha att $10^{2d} = b^2$ får plats i datatypen för att slippa overflow vid t.ex. multiplikation.

8.3.2 Operationer

Variablerna x, y och z är stora positiva tal representerade i basen b som ovan. Vi tänker oss att $x_i = 0$ om $i > n$ där n är antalet tal i vektorn x. Samma sak gäller för y och z. Endast små logiska tillägg krävs för att kunna hantera även negativa tal. Algoritmerna är samma eller snarlika de som man använder vid beräkning på papper.

Addition

$z \leftarrow x + y$

$carry \leftarrow 0$
$n \leftarrow$ maximala storleken av x och y
for $i = 0$ to $n - 1$ **do**
  $tmp \leftarrow x_i + y_i + carry$
  $z_i \leftarrow tmp \bmod b$
  $carry \leftarrow [tmp \geq b]$

end for
$z_n \leftarrow carry$

Subtraktion

$z \leftarrow x - y$ där x antas vara större än y.

$borrow \leftarrow 0$
$n \leftarrow$ maximala storleken av x och y
for $i = 0$ to $n - 1$ **do**
    $tmp \leftarrow x_i - y_i - borrow$
    $z_i \leftarrow tmp \bmod b$
    $borrow \leftarrow [tmp < 0]$
end for

Multiplikation

$z \leftarrow x \cdot y$

$z \leftarrow 0$
$m \leftarrow$ storleken av x
$n \leftarrow$ storleken av y
for $i = 0$ to $m - 1$ **do**
    for $j = 0$ to $n - 1$ **do**
        $z_{i+j} \leftarrow z_{i+j} + x_i \cdot y_j$
    end for
    propagera minnessiffror i z
end for

Algoritmen kräver att den primitiva datatypen rymmer $b^2 - 1$. Det finns betydligt snabbare algoritmer för multiplikation av stora tal.

Division

$z \leftarrow \lfloor x/y \rfloor$ och $r \leftarrow x \bmod y$ där $y < b$

$r \leftarrow 0$
$n \leftarrow$ storleken av x
for $i = n - 1$ to 0 **do**
    $tmp \leftarrow b \cdot r + x_i$
    $z_i \leftarrow \lfloor tmp/y \rfloor$
    $r \leftarrow tmp \bmod y$
end for

För att kunna hantera division med stora heltal krävs en betydligt krångligare algoritm.

Kapitel 9

Talteori

9.1 Integer arithmetics

Let x be a big integer, and e a "small" integer. We want to calculate $z \leftarrow x^e$. The naïve approach is to multiply $x \cdot x \cdot \ldots \cdot x$ e times. But this is a rather slow way to do it. A better way is to *square and multiply.*

Since we have $e = (e \bmod 2) + 2 \cdot \lfloor e/2 \rfloor$, we have:

$$x = x^{e \bmod 2} \cdot x^{2\lfloor \frac{e}{2} \rfloor}$$

where $x^{e \bmod 2}$ is either 1 or x depending on the last bit of e.

9.1.1 Algorithms

ALGORITHM SquareAndMultiply:
- $z \leftarrow 1$
- **while** $e \neq 0$
  - **if** $e \bmod 2 = 1$
    - $z \leftarrow z \cdot x$
  - $x \leftarrow x \cdot x$
  - $e \leftarrow \lfloor e/2 \rfloor$
- **return** z

Let x_i be the value of x the ith time through the while-loop. Then $x_i = x^{2^i}$, and we're looking at the ith bit of e. Alternatively we can calculate x^e recursively:

FUNCTION $\exp(x, e)$:
- **if** $e = 0$
  - **return** 1
- **return** $x^{e \bmod 2} \cdot \exp(x \cdot x, \lfloor e/2 \rfloor)$

9.1.2 Analysis

Consider the binary representation of e: $e = e_0 + 2e_1 + 4e_2 + \ldots$. It follows that $x^e = x^{e_0} \cdot (x^2)^{e_1} \cdot (x^4)^{e_2} \cdot \ldots$. So we have $\log e$ multiplications in the "square and multiply" algorithm, to compare with e multiplications in the naïve approach.

9.2 Modular arithmetics

Calculations mod n.

Let $0 \leq x, y < n$. Recall that $x \equiv y \pmod{n} \Leftrightarrow x - y = k \cdot n$, for some $k \in \mathbb{Z}$.

9.2.1 Add

$(x + y)$ % n
Another way to calculate this is:
$z \leftarrow x + y$
if $(z \geq n)$
$z \leftarrow z - n$

9.2.2 Sub

$(n + x - y)$ % n
Dangerous: $(x - y)$ % n, because e.g. -12 % $7 = -5$.
Another way to calculate this is:
$z = (y > x\ ?\ n + x - y\ :\ x - y)$

9.2.3 Mul

$(x * y)$ % n
Risk for overflow, since x, y can be almost as big as n. Works if $n \leq \sqrt{2^{63}}$ (assuming we are using a 64-bit integer type).

How avoid overflow? 1) Use biginteger, 2) Use "cautious multiplication".

"Cautious multiplication"

Note: Multiplication is related to addition, as exp is related to multiplication. mul $= \ldots + \ldots + \ldots$, exp $= \ldots \cdot \ldots \cdot \ldots$. So we have

$$x \cdot y = x \cdot (y \bmod 2) + x \cdot (2 \cdot \lfloor y/2 \rfloor)$$

which gives us the following algorithm:

Algorithm DoubleAndAdd:
$z \leftarrow 0$

while $y \neq 0$
  if $y \bmod 2 \neq 0$
    $z \leftarrow (z + x)\ \%\ n$
  $x \leftarrow (x + x)\ \%\ n$
  $y \leftarrow \lfloor y/2 \rfloor$
return z

This algorithm runs in $O(\log y)$. The biggest possible number in the calculation is $2n - 2$, so this algorithm works for $n \leq 2^{62}$ (still assuming we are using a 64-bit integer type).

9.2.4 Div

First, think about: What does it mean to divide mod n?

$$\begin{aligned}&x/y \pmod n :\\ &z = x/y \Leftrightarrow z \cdot y = x \Leftrightarrow z = x \cdot y^{-1}\end{aligned}$$

So, we want to find an inverse of y:

$$y \cdot y^{-1} = 1 \pmod n$$

To find inverses the Euclidean algorithm is used. It is based on the following:

$$\begin{aligned}y^{-1} \text{ exists } &\Leftrightarrow \exists\ k, y^{-1} \text{ so that } y \cdot y^{-1} + kn = 1\\ &\Leftrightarrow \gcd(y, n) = 1\end{aligned}$$

(Note that if $y \not\equiv 0 \pmod n$ and n is a prime number, $\gcd(y, n) = 1$.)

The Euclidean algorithm

Given a, b: find x, y so that $x \cdot a + y \cdot b = \gcd(a, b)$. (In the present case, $a = y, b = n, x = y^{-1}, y = k$.)

$(x, y) = \text{euclid}(a, b)$:
  if $a = 0$
    return $(0, 1)$
  if $b = 0$
    return $(1, 0)$
  $(y', x') = \text{euclid}(b, a\ \%\ b)$
  return $(x', y' - (a/b) \cdot x')$

Note that in the algorithm above and in the analysis that follow % is the modulus operator and a/b is integer division, both as in C/C++ and Java.

Correctness of algorithm. Let $d = \text{gcd}(a, b)$.
Note that:

$$\text{gcd}(a, b) = \text{gcd}(b, a \ \% \ b)$$

So

$$y' \cdot b + x' \cdot (a \ \% \ b) = d$$

Since we have[1]

$$a \ \% \ b = a - b \cdot (a/b)$$

we get the new equation used in the next level of recursion:

$$b \cdot y' + (a - b(a/b)) \cdot x' = d$$
$$ax' + b(y' - (a/b) \cdot x') = d$$

Time complexity. There are a logarithmic number of operations, since essentially a bit disappears in every recursive step.

Verification

To verify the solution we compute $x \bmod n_1$:

$$x \bmod n_1 = a_1 n_2^{-1} n_2 + a_2 m_1 n_1 \pmod{n_1}$$

But

$$n_2^{-1} n_2 = 1$$
$$a_2 m_1 n_1 \bmod n_1 = 0$$

So

$$x \bmod n_1 = a_1$$

and $x \bmod n_2 = a_2$ is verified in the same way.Note that when implementing this solution x can be as large as n^3before the $(\text{mod } n_1 n_2)$ evaluation is done. To reduce thepossibility of overflow this mod evaluation has to be done earlier.

The resolved x'' is used to obtain the final solution for x in thefollowing way:

$$x = d \cdot x'' + a_1 \ (\text{mod } \frac{n_1 n_2}{d})$$

Note that $(n_1 n_2)/d = \text{lcm}(n_1, n_2) = n_1' \cdot n_2' \cdot d$.This is of course very expensive and the time complexity is$O(\sqrt{N}) = O(2^{n/2})$, assumed that N is so small that operations on$N$ can be considered to be constant.

[1]Compare with the definition of mod: $a \bmod b = a - b \cdot \lfloor \frac{a}{b} \rfloor$. Note that in contrast to $\frac{a}{b}$, a/b, being integer division, rounds upwards for negative a:s.

9.2.5 Sieve of Eratosthenes

The idea with a sieve is to have a bool array `isprime`$[1...N]$where entry i is true if and only if i is a prime number. When the array hasbeen constructed the look up time is constant. First we look at a simple algorithm to construct such a sieve. Then wewill look at a few methods to improve speed and the use of memory.

ALORITHM Eratosthenes:

```
for i = 2 to N
    isprime[i] ← true
for p = 2 to N
    if isprime[p]
        for r = 2p, 3p, 4p, ... to N
            isprime[r] ← false
```

The time complexity is

$$O(N + \sum_{p \leq N} \frac{N}{p}) = O(N) \cdot \sum_{p \leq N} \frac{1}{p} = O(N \log \log N)$$

where p is prime. Where we use that $\sum_{p \leq N} \frac{1}{p} \approx \log \log N + B_1$. Where$B_1 \approx 0.2615$ is called Mertens' constant.

Improvements of Eratosthenes

There are a few ways to make the implementation of the algorithm faster andmore memory efficient. The ones presented here will not improve th-easymptotic time complexity but they will speed up the program significantly.

Keep only odd numbers in the sieve. Treat the numbers 1 and 2 as special cases and let `isprime` hold only odd numbers from 3 and onwards. This leads to the following improvements:

- Memory usage is reduced to half.
- There are half as many p to check in the outer loop.
- There are half as many r for every p in the inner loop. (Check $3p, 5p, 7p, \ldots$)

All in all this gives us a speed up of roughly a factor 4.

Store 8 bools per byte.

- Memory usage is reduced to $1/8$.
- Indexing becomes more complicated and a little slower, but this iscompensated by more hits in the cache memory when N is big.

Try to avoid unsetting entries in `isprime` unnecessarily. If for example $p = 7$, the bits at $3 \cdot 7$ and $5 \cdot 7$ have been unset earlier, when p was 3 and 5. We want to avoid to unset these bits again and again. So instead of $r = 3p, 5p, 7p, \ldots$ we use $r = p^2, p^2 + 2p, p^2 + 4p, \ldots (r \leq N)$. Still, though, we will unset some bits more than once, e.g. for $3 \cdot 5 \cdot 7$.

9.2.6 The Miller-Rabin primality test

This test was only mentioned at the lecture. It is a randomizedpseudoprimality testing method. It says that a number n is definitelynot prime, or probably prime with a certain probability. The concept isthat the algorithm is run several times until the probability is highenough.

Kapitel 10

Kombinatorik

10.1 Delmängder

Notation: $[n]$ = mängd av tal $\{1, 2, \ldots n\}$. $\binom{n}{k}$ är ett binomialtal, vilket är antalet delmängder av $[n]$ av storlek k.

Exempel: $\binom{5}{3} = 10$,
$\{1,2,3\}, \{1,2,4\}, \{1,2,5\},\ \{1,3,4\}, \{1,3,5\}\{1,4,5\}, \{2,3,4\},$
$\{2,3,5\}, \{2,4,5\}, \{3,4,5\}$

Ett annat sätt att definiera binomialtal är genom formeln

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Vilket intuitivt kan motiveras med att man kastar om de n elementen (på $n!$ olika sätt) och struntar i ordningen på valda (dela med $k!$) och ickevalda (dela med de andra, dvs $(n-k)!$) element. Dessa formler kan också skrivas som

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \frac{n \cdot (n-1) \cdot \ldots \cdot (n-k+1)}{k \cdot (k-1) \ldots \cdot 1} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Några möjliga implementationer

Binomial(n,k)
(1) r ← 1
(2) **for** $i = 1$ **to** k
(3) $\quad r \leftarrow r \cdot (n-k+i)/i$
(4) **return** r

Den kan också implementeras rekursivt i enlighet med rekursionsformeln ovan och med hjälp av memoisering. Algoritmen visas utan relevanta basfall.

```
BINOMIAL(n,k)
    int b[][];
    if k < 0 OR k > n
        return 0
    else if n == k == 0
        return 1
    else if b[n][k] ej beräknat
        b[n][k] ← BINOMIAL(n−1, k−1) + BINOMIAL(n−1,k)
    return b[n][k]
```

Den rekursiva varianten är något säkrare mot overflow, eftersom den aldrig hanterar tal som är större än talet man vill beräkna.

Det totala antalet delmängder av $[n]$ är $2^n = \sum_{k=0}^{n} \binom{n}{k}$.

Multinomialtal, $\binom{n}{n_1,n_2,\ldots,n_k}$ är ett sätt att räkna ut antalet sätt att dela in $[n]$ i k stycken delmängder, där den i:te delmängden har storlek n_i. Observera att binomialtalet $\binom{n}{k}$ är samma sak som multinomialtalet $\binom{n}{k,n-k}$. Notera också följande samband.

$$\binom{n}{n_1, n_2, \ldots, n_k} = \frac{n!}{n_1! \cdot n_2! \cdot \ldots \cdot n_k!}$$

Detta kan motiveras på samma sätt som formeln för binomialtal.

10.2 Permutationer

Antalet permutationer av en sträng x av längd n, $x = x_1, x_2, \ldots, x_n$ är $\binom{n}{f_a,f_b\ldots}$, där f_a är antalet förekomster av bokstaven a. Exempel för strängen "abcaab". $f_a = 3, f_b = 2, f_c = 1$. Antal permutationer blir då $\binom{6}{3,2,1} = 60$. Vill man istället besvara frågan "Hur många permutationer börjar på 'b' " så kan man tänka att eftersom vi har valt en bokstav och en placering så blir svaret $\binom{5}{3,1,1}$.

Hur genererar vi då dessa permutationer, utan onödiga upprepningar? Jo, vi genererar "nästa" permutation utifrån en given, i lexikografisk mening. Vi kan sedan generera alla genom att börja på den första, den där tecknena i strängen är sorterade, och sedan köra algoritmen.

Exempel: strängen "abcca". $i = 2, j = 4$. Efter swap: "ac<u>cba</u>", efter reverse: "acabc". Är vi på den "sista" permutationen, som är helt sorterad i fallande ordning kommer steg ett att misslyckas, det finns inget i sådant att $x_j > x_i$. Detta måste givetvis tas i beaktande vid implementation.

NEXT(s)
(1) Hitta kortaste suffix $x_i \dots x_n$ sådant att $x_i < x_{i+1}$,
dvs $x_i < x_{i+1} \geq x_{i+2} \geq \dots \geq x_n$.
(2) Hitta största j sådant att $x_j > x_i$
(3) SWAP(x_i, x_j)
(4) REVERSE$(x_i \dots x_n)$

Om strängen har längd n och består av ettor och nollor kan vi tolka den som en delmängd på $[n]$, där vi tar med element i om tecknet på position i är en etta. Att ta nästa permutation blir då detsamma som att konstruera nästa delmängd av samma storlek, och vi har därmed ett sätt att generera alla delmängder av en viss storlek.

Nedan beskrivs en rolig algoritm för att göra detta meddelst bitpetande, som dock kräver att datorn använder tvåkomplementsrepresentation för negativa heltal:

NEXT(a)
(1) c ← a **AND** -a
(2) r ← a + c
(3) **return** ((r **XOR** a)/(4 · c)) **OR** r

10.3 Partitioner

$B = \{B_1 \dots B_k\}$ är en partition av $[n]$ i k delar om och endast om

1. $\bigcup_{i=1}^{k} B_k = [n]$
2. $B_i \cap B_j = \emptyset$
3. $B_i \neq \emptyset$

$B_1 \dots B_n$ brukar även kallas för block.

Exempel på en partition av [8] i 4 block: $B = \{\{1, 5, 7\}, \{2, 8\}, \{6\}, \{3, 4\}\}$

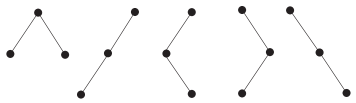
Hur många partitioner av $[n]$ finns det i k block? Jo:

$$S(n, k) = S(n-1, k-1) + k \cdot S(n-1, k)$$

Basfall:

$$S(n, k) = \begin{cases} 0 & k > n, k < 0, n < 0 \\ 1 & k = 0, n = 0 \end{cases}$$

Dessa är Stirlingtalen av det andra slaget. anledningen varför formeln ser ut som den gör kan man tänka på hur blocket som innehåller talet n ser ut.



Figur 10.1: Binära träd med 3 noder

Antingen är n ett block för sig självt, eller så bor n tillsammans med några andra element. I det första fallet utgör resten av partitionen en partition av $[n-1]$ i $k-1$ block, vilket kan göras på $S(n-1, k-1)$ sätt. I det andra fallet utgör resten av partitionen en partition av $[n-1]$ i k block, och för varje sådan partition av $[n-1]$ kan vi konstruera k partitioner på $[n]$ sätt genom att välja vilken av de $k$blocken som n ska vara med i.

Hur många partitioner av $[n]$ finns det totalt?

$$B(n) = \sum_{k=0}^{n} S(n,k) = \sum_{k=1}^{n} \binom{n-1}{k-1} B(n-k)$$

Detta är de så kallade "Bell-talen". Man kan för att övertyga sig om formelns korrekthet betrakta hur blocket som innehåler talet n ser ut, och säga att det innehåller totalt k element. Dessa k element kan väljas på $\binom{n-1}{k-1}$. -1 kommer sig av att vi redan bestämt ett av de k elementen. För varje sådant val av ett block av storlek k kan vi betrakta de återstående blocken som en partition av $[n-k]$, utav vilka det finns $B(n-k)$ stycken.

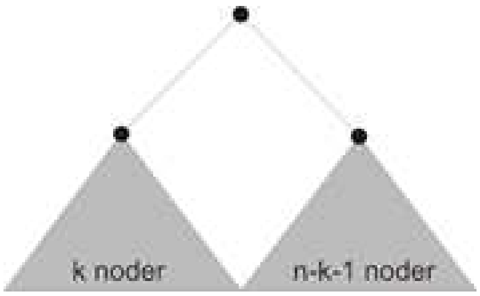
10.4 Catalantalen

Exempel på användningsområde: Räkna antalet binära träd med n noder. $T(3) = 5$, nämligen de i figur 10.1.

Formeln för beräkning av Catalantal skrivs som

$$T(n) = \sum_{k=0}^{n-1} T(k) \cdot T(n-1-k)$$

Basfallet för denna formel är $T(0) = 1$. Tid för beräkning enligt denna rekursiva formel är $O(n^2)$. För att motivera formeln: Om antalet träd som har n noder är $T_k(n)$ stycken, och om det vänstra delträdet till roten har k noder så kommer det högra delträdet ha $n-k-1$ noder, enligt figur 10.2. De två delträden beror inte på varandra, och kan alltså väljas oberoende, så vi måste ha $T_k(n) = T(k) \cdot T(n-k-1)$ noder. Vi har också att $T(n) = \sum_{k=0}^{n-1} T_k(n)$, eftersom det vänstra delträdet i varje träd på n noder kan ha någonstans emellan 0 och $n-1$ noder.



Figur 10.2: Delträd

Ett annat exempel på användningsområde är att räkna välbalanserade parenteser på n stycken parentespar: () (), (()). Ett exempel åp icke välbalanserade parenteser är till exempel ())(.

$P(3) = 5$, och de fem parentetiseringarna är: (())(), ((())), (()()), ()(()), ()()().

Rekursionen för denna är inte bara snarlik, den är exakt likadan: $P(n) = \sum_{k=0}^{n-1} P(k) \cdot P(n-1-k)$. Givetvis är tid för beräkning också $O(n^2)$. Notera att vi givet en välmatchad parentessträng på ett unikt sätt kan skriva den som "$(X)Y$", där X och Y är välmatchade parentessträngar (som kan vara tomma). Högerparentesen efter X är precis den högerparentes som matchar den första parentesen. Om $P_k(n)$ är antalet välmatchade parentessträngar på n parentespar i vilka X ovan består av k parentespar så är logiken densamma som från resonemanget för $T(n)$ ovan.

Ett mycket bättre sätt att beräkna Catalantal är enligt formeln

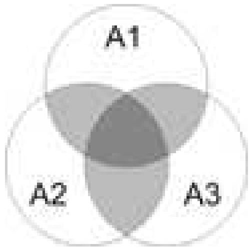
$$T(n) = \frac{\binom{2n}{n}}{n+1} = \frac{2(2n-1) \cdot T(n-1)}{n+1}$$

Denna formel har en mycket snällare rekursion, och mellansteget kommer inte innehålla lika stora tal.

10.5 Inklusion-Exklusion

Ett sätt att beräkna storleken på en union $|A_1 \cup A_2 \cup \ldots \cup A_n|$ där elementen som ingår överlappar. Ett exempel på tre mängder som överlappar finns i figur 10.3.

$$|A_1 \cup \ldots \cup A_n| = \sum_{\emptyset \neq I \subseteq [n]} (-1)^{\|I\|-1} | \cap_{i \in I} A_i|$$



Figur 10.3: Överlappning av mängder

Exempel:

$$\begin{aligned} |A_i \cup A_2 \cup A_3| &= |A_1| + |A_2| + |A_3| - |A_1 \cap A_2| - \\ & |A_2 \cap A_3| - |A_1 \cap A_3| + |A_1 \cap A_2 \cap A_3| \end{aligned}$$

Detta är praktiskt användbart om

- Det är "enkelt" att beräkna snittet, och n är väldigt litet, säg ($n \leq 20$).
- Det är "jätteenkelt" att beräkna snittet, till exempel om $|\cap_{i \in I} A_i| = F(|I|)$, alltså om snittet bara beror av $|I|$. $|A_i \cup \ldots \cup A_n| = \sum_{k=1}^{n} \binom{n}{k} (-1)^{k-1} F(k)$

Ett exempel på tillämpning av inklusion-exklusion är att räkna ut så kallade derangements, permutationer utan fixpunkter. Kan med fördel användas vid *hattproblemet*. Vid en middagsbjudning där alla gäster har med sig varsin hatt, och vid hemgång får med sig slumpvis utvald hatt från alla hattar, vad är sannolikheten att ingen får med sig sin egen hatt hem? Denna visar sig vara $\approx \frac{1}{e}$.

Kapitel 11

Beräkningsgeometri

11.1 Triangle area

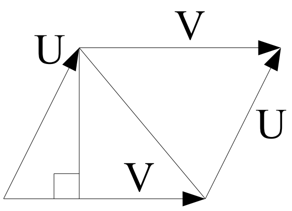


Figure 11.1: Area of triangle

Calculating the area of a triangle can be done with a number of methods. One of the most useful starts with the triangle being described as two vectors U and V, see figure 11.1. Using some simple vector mathematics we get the following formula for the area:

$$\left\| \begin{array}{c} -U- \\ -V- \end{array} \right\| = 2A$$

We insert the vectors and get the following formula for the area:

$$\left\| \begin{array}{cc} U_x & U_y \\ V_x & V_y \end{array} \right\| = 2A$$

Calculating the actual value of the determinant using linear algebra gives us the following solution:

$$|U_x V_y - V_x U_y| = 2A$$

Here we should observe two things. First, the result will be signed, which can be very useful (as we will show later). Also, this solution can be generalized to more dimensions. In three dimensions we have

$$\left\| \begin{array}{ccc} A_x & A_y & A_z \\ B_x & B_y & B_z \\ C_x & C_y & C_z \end{array} \right\| = 6V$$

Giving the volume of the space between three vectors. Once more the result will be signed. It rare to need more than three dimensions, but the calculations are the same and the multiplicative factor will be !n, n the number of dimensions involved.

11.2 Inside outside circle

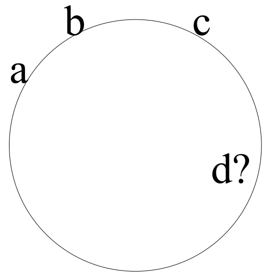


Figure 11.2: Points on a circle

Given three points in a plane we can construct a circle out of these, except if these three points form a line. Given a fourth point we can also check if this point is a part of the circle formed by the previous three points. In the figure 11.2 we have four points given a, b, c and d. We want to find out if d is located inside or outside the circle formed by a, b and c. To do this we create the following matrix:

$$\begin{vmatrix} a_x a_y & a_x^2 a_y^2 & 1 \\ b_x b_y & b_x^2 b_y^2 & 1 \\ c_x c_y & c_x^2 c_y^2 & 1 \\ d_x d_y & d_x^2 d_y^2 & 1 \end{vmatrix}$$

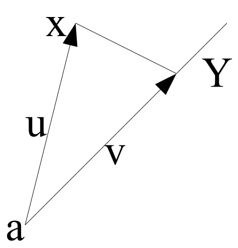


Figure 11.3: Side of a line

If the evaluation of this matrix is positive d is located inside the circle formed by abc, if it's negative d is located outside the circle. And if d is located exactly on the circle the value of the matrix will be 0.

11.3 Side of a line

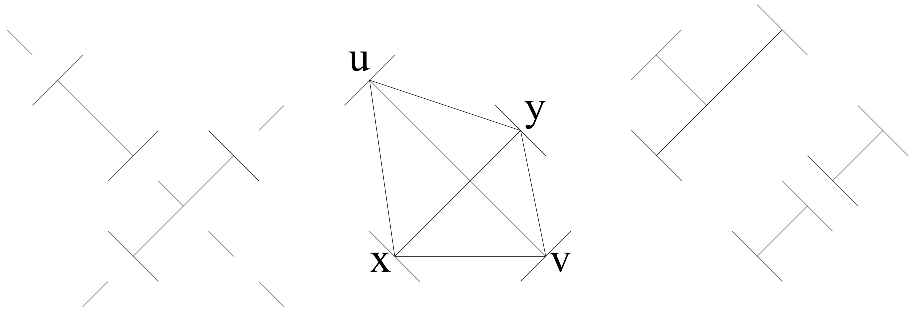
Using the previous result with the triangle area we can easily find out which side of a line a point is. In this case we have the point x and the line Y. We form a vector v from a along Y and a vector u from a to x. Having these two vectors we can use the formula for calculating the area. A positive area means that x lies on the left side of Y seen from a, a negative area means that x lies on the right side of Y. If the area is 0 then x lies exactly on the line Y.

11.4 Three points on a line

To check if three points lies on a line we once more use the triangle area. Assume we have points x, y and z. If the area of the triangle xyz is 0 or -0 then they lie on a line. Creating the vectors xy and xz should not be any problem

11.5 Intersection point of two segments

Finding the intersection point of two segments in the general case involves solving a linear equation system, which can be problematic. This is especially true is the lines that extends the segments are almost parallel. In that case



(a) Nonintersecting (b) Intersecting (c) Special cases

Figure 11.4: Various intersections

the solution might very well be close to infinity. To avoid this problem we will once more take advantage the previous method to calculate which side of a vector a point can be found. Examine figure 11.4(b). In this case we have two intersecting segments xy and uv. We begin with examining the situation from x. We check that u and v lies on different sides of the line between x and y. This is done by checking so the sign of the area of the triangle xyv is different from the triangle xyu. The second step is to examine the situation from v, and make sure the sign of the area of the triangle uvx is different from the sign of the area of the triangle uvy. If these two statments holds, the segments intersects. We have to deal with the two special cases, illustrated in figure 11.4(c). If the area of one of the triangles is 0 it means that the examined point lies on the line. This is acceptable, and a valid solution, as long as the area of the other triangle is non-zero. However, if both triangles have the area 0 we are dealing with two segments that lies on a common line and will not be able to find if they intersect with this method. Instead we will have to check the intervals of the two segments for overlap.

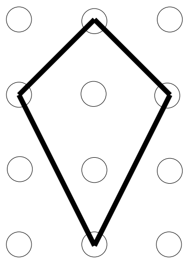


Figure 11.5: Area of polygon in Z

11.6 Area of polygon with corners in Z

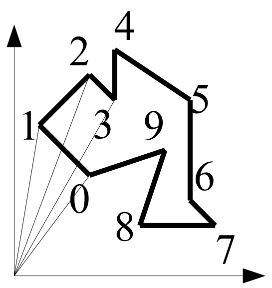


Figure 11.6: Area of polygon, general method

Assuming that the polygon does not intersect itself at any point we can use the following method to calculate the area of the polygon. We place the polygon on a grid where all corners of the polygon ends up in integer coordinates. This means that the polygon must either lie in Z or be possible to transform to Z. Then we count the number of grid intersections that are inside the polygon. In figure 11.5 this is two. Then we calculate the number of grid intersections that lies exactly on the edge of the polygon. Once more, in figure 11.5 this is four. Knowing these two facts lets us calculate the area of the polygon, in the unit of the grid. The formula is $Area = I + \frac{R}{2} - 1$ where I is the number of internal grid points and R is the number of grid points on the edge. The actual proof of why this method works is left as an exercise to the reader.

11.7 Area of polygon

Another method of calculating the area of a polygon assumes that we have an ordered set of all the corners in the polygon. The corners should be ordered so n+1 is always the next corner when walking along the sides of the polygon. To get the area we chose a special location (usually origo). We calculate the sum of the signed area of the triangle from origo to n to n+1 for all n. Notice that the signed area will "cancel out" the parts outside the polygon. This method will give us the area, signed, depending on if we went around the polygon clockwise or counterclockwise.

11.8 Green's formula

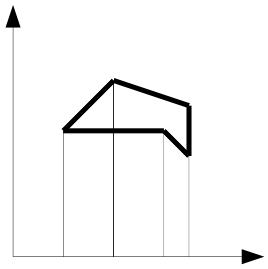


Figure 11.7: Area of with Green's formula

We can also use Green's formula, which is similar to the previous method, but in Green's formula we calculate the area in the form of four cornered polygons instead of triangles. We begin with choosing a base axis, usually the x axis. The idea is simply to sort the sides in two sets, "top" and "bottom", depending on the distance from the chosen axis. Then we calculate the sum of the area between the top sides and the axis and remove the sum of the area between the bottom axis and the bottom sides. Figure 11.7 should give a general idea of how the method works.

11.9 Closest pairs

Finding the closest pairs of points in the plane can easily be done in $O(n^2)$ by testing all pairs of points. However, we obviously want a faster solution. This can be done by first creating two lists, p and v, with p containing the x coordinate and v containing the y coordinate of the point. Then the pairs are sorted after the x coordinate.

mindist (p, v)
    if$|p| \leq 3$
        return $min(p, v)$
    else
        $v_1 \leftarrow v[1...n/2]$
        $v_2 \leftarrow v[n/2...n]$
        $p_1 \leftarrow p[1...n/2]$
        $p_2 \leftarrow p[n/2...n]$
        $d_i = mindist(p_i, v_i)$
        $d = min(d_i)$

Having gotten this far we can observe that we have not checked for all pairs. If one part of the shortest paid lies in 1 and one part lies in 2 then we will obviously not find the pair. So we will also have to check those that lies close to the edge. For any pair to be interesting the distance between them has to be less than d. So we grab those points in 1 that lies less than d from the first point in 2. And we grab the points in 2 that lies less than d from the last point in 1. Then we can make a number of additional observations. First, a pair has to have one point in 1 and one point in 2 to be interesting. Secondly, the points has to lie within d from the first point. This gives us at most 4 points to examine the distance to. So we simply iterate through the points we found in 1, for each point we find the (at most 4) points in 2 that lies within d from the point in 1 and calculate the distances to each of these.

11.10 Inside a polygon with crossing numbers

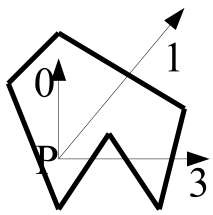


Figure 11.8: Number of crossing from two points to two points.

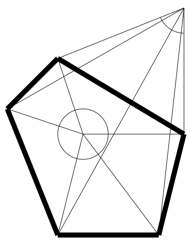


Figure 11.9: Winding numbers.

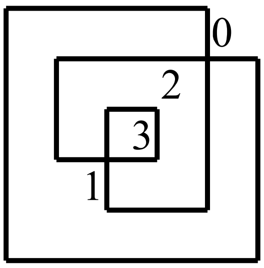


Figure 11.10: Complicated polygon.

A simple way to figure out if a point lies inside or outside a polygon is to draw a straight line from the point to a point we know to be inside or outside the polygon. Then we can simply count the number of times the line intersects the polygon. An odd number of intersections tells us that the unknown point lies on the other side as the known point, an even number of intersections means that the two lies on the same side of the polygon. This method is quick and rather easy to implement. However, it is prone to problems if we happen to go through a corner of the polygon. In that case there are a number of special rules that can be used. A common set in graphic applications are as follows:

1. An upward edge includes its starting endpoint, and excludes its final endpoint.
2. A downward edge excludes its starting endpoint and includes its final endpoint.
3. Horizontal edges are excluded
4. The edge-ray intersection point must be strictly to the right of the point P.

Here P is the point we are trying to find and the edge-ray is the line along which we will examine the polygon.

11.11 Inside a polygon with winding numbers

Using winding numbers we calculate how many times a closed polygon circles around a given point. To do this we start in the given point and calculate the angles between each following pair of corners n, n+1. Adding all these angles together we will in the end receive a comparison number that is on the form $(n * 2\pi)$, n being how many times the polygon circles around the point p. Obviously 0 is outside the polygon and 2π is inside it. The method can be made more efficient, removing the arctan calculation. This method also has the advantage of finding points that are encircled several times by the polygon, unlike the crossing numbers that will only find an odd number of crossings. Figure 11.9 illustrates this very well. Crossing numbers will detect 0 and 2 as outside and 1 and 3 as inside while winding numbers will detect 1,2 and 3 as inside with varying winding numbers.

11.12 Convex hull (Greyham scan)

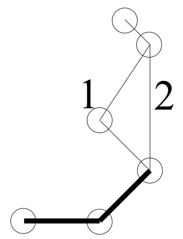


Figure 11.11: Grayham scan.

Finding the convex hull of a set of points can be done with a method called Greyham scan. Using this method we begin with an extreme point and then we sort all remaining points after the angle between them and a basis line. Having done this we begin adding points, in the order they are sorted, testing for each point we add so we always turn in a positive direction. Due to the nature of a convex hull we always have to turn in the same direction. If we reach a point where adding the next point would lead to a turn in the wrong direction we will have to backtrack until we can add the next point without this problem. Looking at figure 11.11 we have added several points, but at this point we would first go down path 1, detect that it turns in the wrong direction. Then we would have to backtrack and go down path 2 instead, ignoring the point at 1 .

Kapitel 12

Amorterad analys

Kapitlet handlar om Kruskals algoritm, en datorstruktur för operationer på disjunkta mängder som kan användas för att optimera denna algoritm samt amorterad analys, för analysera vår datastruktur.

12.1 Kruskals algoritm för MST

Under graf-föreläsningen nämndes Kruskals algoritm endast i förbigående. Kruskals algoritm bygger, givet en graf G, upp ett minimalt spännande träd för G genom att börja med en graf utan kanter och sedan succesivt lägga till kanter från G. I varje steg i algoritmen läggs den kant som, dels har minst vikt, dels inte skapar en cykel till i den nya grafen, till. (Se Algoritm 12.1).

Algoritm 14: Kruskals algoritm
Beskrivning: Vi börjar med $T = (V, 0)$ och lägger succesivt till kanter från G
Input: En graf $G = (V, E)$ med viktade kanter
Output: Minsta Spännande Träd för G
(1) $T \leftarrow 0$
(2) **for** $i = 1$ **to** $|V| - 1$
(3) Hitta billigast $e \in E$ sådan att $T \cup \{e\}$ ej har någon cykel
(4) $T = T \cup \{e\}$

Vi har två delar som är oklara hur vi bör hantera dem

- "Hitta billigaste $e \in E$": Om vi först sorterar kanterna efter vikt kan detta göras enkelt. Sortering tar tid $O(|E| \log |E|)$.
- "..sådan att $T \cup \{e\}$ ej har någon cykel": För att inte skapa en cykel måste noderna x och y för kanten $e = (x, y)$ ligga i olika komponenter i

T. För att avgöra vilken komponent en nod ligger i kan vi, för varje nod x, lagra detta i säg $komponent[x]$, då kan vi göra uppslagningen här i konstant tid. Däremot kommer vi behöva slå ihop två komponenter i steget efter. Om vi lägger till en kant $e = (x, y)$ så måste vi då markera om alla hörn i antingen den komponent som x ligger i eller i den som y ligger i. Om vi alltid märker om den komponent som är minst får vi en total körtid $O(|V| \log |V|)$ för algoritmen. Efter ommärkning av en nod tillhör den en minst dubbelt så stor komponent som innan. Vi kan alltså som mest märka om den $\log |V|$ gånger då komponenten noden tillhör efter $\log |V|$ ommärkningar måste innehåller hela V. Detta kan förbättras genom att använda den datatyp för disjunkta mängder som följer.

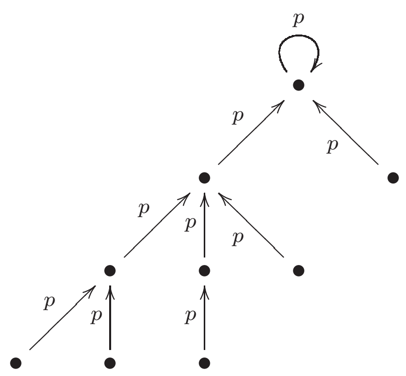
12.2 Datatyp för operation på disjunkta mängder

Låt $X = \{x_1, \ldots, x_n\}$ vara alla element som skall behandlas och låt $S_1, \ldots, S_m$ vara ett antal mängder av dessa element. Vi definierar följande operationer:

- $MakeSet(x_i) = \{x_i\}$
- $Find(x_i) = j \in \mathbb{N}$ sådant att $x_i \in S_j$
- $Union(x_i, x_j) =$ Slår ihop $S_{Find(x_i)}$ och $S_{Find(x_j)}$ till en mängd. De två ursprungliga mängderna förstörs.

12.2.1 Implementation

Vi representerar en mängd som ett träd med noder i X och riktade kanter mot trädets rot. Vi låter trädets rot representera hela trädet (Se Figur 12.1).



Figur 12.1: Trädrepresentation av en mängd

Operationer

Vi kan nu implementera våra operationer enligt Algoritm 12.2.1 och 12.2.1. Våra träd kan dock i värsta fall degenerera till listor (om argumentet b i link alltid är ensamt i sin mängd) och då tar Find-operationen i värsta fall tid $O(k)$ där k är antal element i mängden.

Algoritm 15: Operationer på disjunkta mängder
Beskrivning: Ett första försök till implementation av tidigare definierade operationer på disjunkta mängder (MakeSet och Find)

```
MakeSet(x)
    p[x] ← x #gör x till rot

Find(x)
    if x = p[x]
        return x
    else
        return Find(p[x])
```

Algoritm 16: Operationer på disjunkta mängder
Beskrivning: Ett första försök till implementation av tidigare definierade operationer på disjunkta mängder (Union)

```
Union(x, y)
    a ← Find(x)
    b ← Find(y)
    if a ≠ b
        Link(a, b)

Link(a, b)
    p[a] ← b
```

För att undvika detta kan vi hålla reda på storleken av varje delträd: Vi låter $rank[x]$ vara en (över-) skattning av djupet i det träd där x är rot och skriver om våra operationer som i Algoritm 12.2.1 och 12.2.1 (där Union är som tidigare) så att vi hela tiden försöker hålla träden så grunda som möjligt. Find-operationen kommer nu max att ta tid $O(s)$ där s är djupet i trädet som representerar den sökta mängden, operationen kommer dessutom hela tiden försöka hålla s så litet som möjligt genom att uppdatera förälderpekarna (Se Figur 12.2).

Algoritm 17: Operationer på disjunkta mängder
Beskrivning: Ett andra försök till implementation av tidigare definierade operationer på disjunkta mängder där vi försöker att minska trädens djup (MakeSet och Find)

```
MakeSet(x)
    p[x] ← x # gör x till rot
    rank[x] ← 0 # initiera rank[x]


Find(x)
    if x ≠ p[x]
        p[x] =Find(p[x]) # uppdatera förälderpekare
        enligt Figur 12.2.
    return p[x]
```

Algoritm 18: Operationer på disjunkta mängder
Beskrivning: Ett andra försök till implementation av tidigare definierade operationer på disjunkta mängder där vi försöker att minska trädens djup (Link)

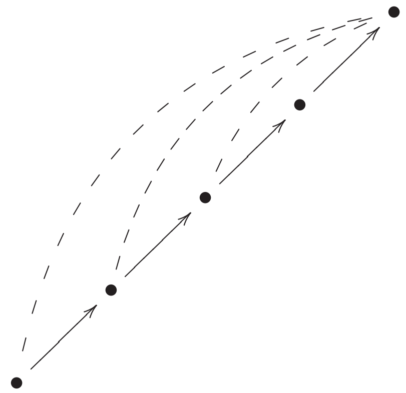
```
Link(a, b)
    if rank[a] < rank[b]
        p[a] ← b
    else
        p[b] ← a
        if rank[a] = rank[b]
            rank[a] ← rank[a] + 1 # om träden är lika
            djupa kommer det totala djupet att öka med
            ett
```

12.2.2 Analys

Våra operationer har nu komplexitet som följer för (n element total):

- MakeSet: $O(1)$
- Find: $O(\log n)$ men väldigt få Find-anrop kommer ta så lång tid.
- Union: $O(1)$
- Link: $O(1)$

Vi vill visa att Find sällan tar så lång tid som $O(\log n)$ stämmer i påståendet ovan och att $O(m \cdot \alpha(n))$ faktiskt är en övre gräns för körtiden om vi gör m st operationer (av någon av ovanstående typer) på n st element (där $\alpha(n)$ är



Figur 12.2: Uppdatering av förälderpekare i Find-operationen

en funktion som växer mycket långsamt). Vi vill då också visa att funktionen $\alpha()$ ovan växer väldigt långsamt. Låt A_k vara som följer:

$$A_k(j) = \begin{cases} j+1 & \text{om } k=0 \\ A_{k-1}^{(j+1)}(j) & \text{annars} \end{cases}$$

där $A_{k-1}^{(j+1)}(j) = A_{k-1}(A_{k-1}(\ldots(A_{k-1}(j))\ldots))$ så att A_{k-1} appliceras $j+1$ gånger (Se Figur 12.3).

$$\begin{aligned} A_1(j) &= A_0^{(j+1)}(j) = A_0^{(j)}(j+1) = 2j+1 \\ A_2(j) &= 2^{j+1}(j+1)-1 \end{aligned}$$

$$\begin{aligned} A_0(1) &= 2 \\ A_1(1) &= 3 \\ A_2(1) &= 7 \\ A_3(1) &= A_2^{(2)}(1) = A_2(7) = 2047 \\ A_4(1) &= A_3^{(2)}(1) = A_3(2047) = A_2^{(2048)}(2047) \gg A_2(2047) \gg 2^{2048} \end{aligned}$$

Figur 12.3: Värden för $A_k(j)$ för olika k, j

Vi kan nu definiera α som följer (Se även Figur 12.4):

$$\alpha(n) = min\{k | A_k(1) \geq n\}$$

För att visa att vårt α faktiskt stämmer gör vi en amorterad analys av vår datastruktur.

$\alpha(n) = 0$ för $n \leq 2$
$\alpha(n) = 1$ för $n = 3$
$\alpha(n) = 2$ för $4 \leq n \leq 7$
$\alpha(n) = 3$ för $8 \leq n \leq 2047$
$\alpha(n) = 4$ för $2048 \leq n \leq A_4(1) \gg 2^{2048}$

Figur 12.4: Värden på $\alpha(n)$ för olika n

12.3 Amorterad analys, datastrukturer

Det finns tre vanliga typer av amorterad analys:

- Totalanalys
- Bokföringsmetoden
  Man tilldelar här varje operation en amorterad kostnad $\hat{c}_i$ sådan att $\hat{c}_i$ inte nödvändigtvis är större än den verkliga kostnaden c_i men så att $\sum_{i=1}^{m} \hat{c}_i \geq \sum_{i=1}^{m} c_i$ där vi summerar över alla operationer i en körning.
- Potentialmetoden
  Man skapar en potentialfunktion $\Phi(D_i) = \Phi_i$, där D_i är datastrukturen efter i operationer, sådan att $\Phi_0 = 0$ och $\Phi_i \geq 0$. Man tilldelar sen operationerna amorterade konstnader enligt $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$. Vi får då att
  $$\sum_{i=1}^{m} \hat{c}_i = \sum_{i=1}^{m} c_i + \Phi_i - \Phi_{i-1} = \Big(\sum_{i=1}^{m} c_i\Big) + \Phi_m - \Phi_0 \geq \sum_{i=1}^{m} c_i \ .$$
  (Potentialfunktionen utformas sådan att "billiga" operationer ökar potentialen något medan dyra operationer sänker den.)

Vi kommer att använda Potentialmetoden.

Exempel 12.1. *Låt oss analysera en stack, vi har operationerna push(x), pop$()$ och multipop(k) där push och pop fungerar som vanligt och multipop poppar k element från stacken. Låt potentialfunktionen $\Phi(D_i)$ vara stack-djupet efter i operationer. Varje operation ges amorterad kostnad $\hat{c}_i = c_i +$ $<$ökning av $\Phi>$, d.v.s:*

- *push*(x) : $\hat{c}_i = 1 + 1 = 2$
- *pop*$()$: $\hat{c}_i = 1 - 1 = 0$
- *multipop*(k) : $\hat{c}_i = k - k = 0$

12.4 Amorterad analys av vår tidigare datastruktur

Vi är nu redo att analysera vår datastruktur för disjunkta mängder. Notera följande:

- Om $x = p[x]$ och $rank[x] = r$ har trädet med rot i x minst 2^r noder.
- $rank[x] \leq n - 1$ (faktiskt $\leq \log n$)
- $rank[x] = rank[p[x]]$ omm x är en rot
  $rank[x] < rank[p[x]]$ annars
- $rank[x]$ kan aldrig minska
- om $x \neq p[x]$ så ändras inte $rank[x]$ mer

12.4.1 Potentialfunktion

Vi vill definiera en funktion $\varphi_i(x)$ för varje x sådan att $\Phi_i = \sum_x \varphi_i(x)$. För att göra detta behöver vi två hjälpfunktioner:

$level(x)$

Låt $level(x) = \max\{k | rank[p[x]] \geq A_k(rank[x])\}$. Notera att $level(x)$ endast är definierad om $x \neq p[x]$ och $rank[x] \geq 1$. Notera också att att $0 \leq level(x) < \alpha(n)$.

$iter(x)$

Notera att

$$A_{level(x)}(rank[x]) \leq rank[p[x]] < A_{level(x)}^{(rank[x]+1)}(rank[x]) = A_{level(x)+1}(rank[x])$$

Låt $iter(x) = \max\{i | A_{level(x)}^{(i)}(rank[x]) \leq rank[p[x]]\}$. Notera att $1 \leq iter(x) \leq rank[x]$.

$\varphi(x)$

Vi har nu två mått på hur mycket större $rank$ är för föräldern till en nod x jämfört med $rank$ för x och kan med dessa definiera vårt φ enligt följande:

$$\varphi(x) = \begin{cases} \alpha(n) \cdot rank[x] & \text{om } x = p[x] \text{ eller } rank[x] = 0 \\ (\alpha(n) - level(x))rank[x] - iter(x) & \text{om } x \text{ inte är en rot och } rank[x] \geq 1 \end{cases}$$

Notera att $0 \leq \varphi(x) \leq \alpha(n) \cdot rank[x]$ då $iter(x)$ både kan ökas eller minskas, om den minskar gör den då det med som mest $rank[x] - 1$ varvid *level*(x) ökar. *level*(x) kan dessutom aldrig minska. Detta medför att om någon av dem ändras i steg i gäller

$$\varphi_i(x) \leq \varphi_{i-1}(x) - 1$$

12.4.2 Analys av datastrukturen

I analysen räknar vi Union-operationen som två Find-operationer och en Link-operation och antar att vi kör alla MakeSet-operationer i början innan någon annan typ av operation. Då får vi amorterade konstnader för de olika operationerna enligt följande.

MakeSet

Både före och efter anropet till MakeSet kommer $rank[k]$ att vara noll, alltså kommer $\varphi_i(x) = \varphi_{i-1}(x) = 0$ om MakeSet körs i steg i. För $y \neq x$ kommer även $\varphi_i(y) = \varphi_{i-1}(y)$ att gälla, vi har alltså ingen förändring i potential och

$$\hat{c}_i = c_i = 1$$

Link

Vi antar att $rank[x] \geq rank[y]$. I så fall kommer x bli ny förälder till y och $\varphi(y)$ kan minska, liksom $\varphi(z)$ där z ligger i den mängd som har x som rot. Skulle båda träden vara lika djupa ökar $rank[x]$ med 1 och då ökar $\varphi(x)$ med $\alpha(n)$. Vi får då

$$\hat{c}_i \leq c_i + \alpha(n) = 1 + \alpha(n)$$

Find

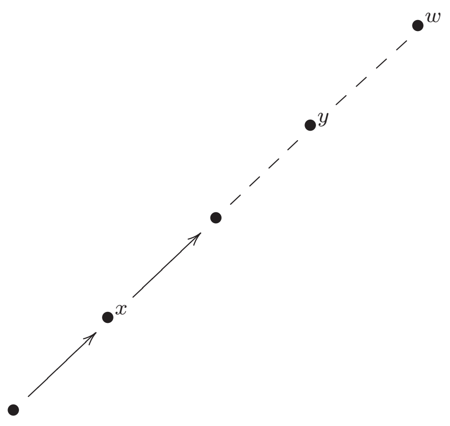
Vi betraktar $Find(x)$ där x ligger s steg från roten i sin mängd, c_i är då precis s. Vi vill visa att för minst $s-(\alpha(n)+2)$ noder längs stigen så minskar potentialen. Vi har då

$$\Phi_i \leq \Phi_{i-1} - s + \alpha(n) + 2$$

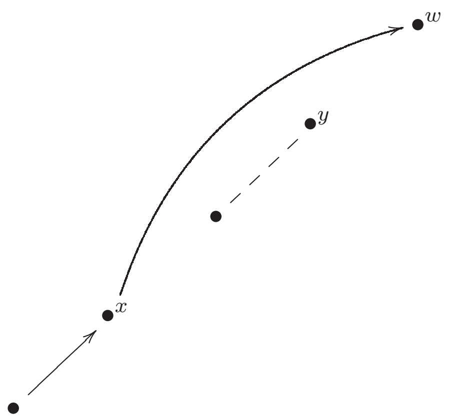
och som en direkt följd

$$\hat{c}_i \leq s - s + \alpha(n) + 2 \in O(\alpha(n))$$

Betrakta situationen i Figur 12.5, låt situationen vara sådan att vi före operationen har $level(x) = k$, $iter(x) = i$, $level(y) = k$. Låt x vara en nod så att $rank[x] > 0$ och x har en annan nod y efter sig som inte är en rot. $level(y) = level(x)$ precis innan operationen. Alla förutom högst $\alpha(n) + 2$ noder uppfyller dessa krav på x. De som inte uppfyller dem är första noden(om den har rank 0), sista noden(roten) och sista noden ω for



Figur 12.5: Situation innan en körning av Find-operationen



Figur 12.6: Situation under körning av Find-operationen, w blir ny förälder till x

ilken $level(\omega) = k$, för varje $k = 0, 1, 2, ..., \alpha(n) - 1$. Låt oss finna en sådan nod x och visa att x's potitial minskar med minst 1.

Låt $r = rank[p[y]]$. Vi har då även att:

$$\begin{aligned} rank[y] &\geq rank[p[x]] \geq A_k^i(rank[x]) \\ r = rank[p[y]] &\geq A_k(rank[y]) \\ r &\geq A_k(rank[y]) \geq A_k(A_k^{(i)}(rank[x])) = A_k^{(i+1)}(rank[x]) \end{aligned}$$

Efter operationen får vi (Se Figur 12.6):

$$rank[p[x]] \geq r \geq A_k^{(i+1)}(rank[x])$$

Eftersom operationen gör att både x och y får samma förälder, vet vi att efter operationen gäller $rank[p[x]] = rank[p[y]]$ och att operationen inte minskar $rank[p[y]]$. Då $rank[x]$ inte ändras får vi $rank[p[x]] \geq A_k^{(i+1)}$. Operationen kommer antingen att få $iter(x)$ att öka (till minst $i + 1$) eller $level(x)$ att öka (vilket inträffar om $iter(x)$ ökar till minst $rank[x] + 1$). Oavsett vilket så kommer $\varphi(x)$ att minska med minst 1 med hjälp av $\varphi_i(x) \leq \varphi_{i-1}(x) - 1$. Den amorterade kostnaden av operationen är den aktuella kostnaden plus skillnaden i potential. Den aktuella kostnaden är s och vi har visat att den totala potentialen minskar med $s - (\alpha(n)) + 2)$. Den amorterade kostnaden är därför som mest $s - (s - (\alpha(n) + 2)) = s - s + \alpha(n) + 2 = \alpha(n)$.