Klassen $\mathcal{P}$

Dagens föreläsning

- Motivation och bakgrund
- Definition
- Viktiga egenskaper hos $\mathcal{P}$
- Problemet PATH och dess lösning på polynomiell tid
- RELPRIME-problemets lärdom
- En algoritm som löser A_{CFG} på polynomiell tid

Motivation och bakgrund, eller ”vad gör vi här”?

Vad går den här kursen ut på, egentligen? När används egentligen automater och grammatiker på riktigt?

Jag tror mycket väl att man kan vara en helt OK programmerare utan den här kursen!

. . . men. . .

Men programmerare är alla som har kläm på hur man uttrycker sig i ett programspråk. Det finns inga krav på förståelse för den bakomliggande teorin.

Lika lite som man är författare bara för att man kan svenska, är man en *datavetare* för att man kan programmera.

”Vaffö gör di på dette viset?”

Vi måste nå ett teoretiskt djup, en förståelse för varför en metod eller ett verktyg är mer rätt än en annan i en situation, vilka begränsningar som finns och vad som är ”bra”.

Inte nog med allt detta, vi måste också lära oss kommunicera med våra kollegor inom området.

Förutom att introducera en abstrakt och mer formell bild av datavetenskapen, lär den här kursen oss just detta!

Mer specifikt för dagens föreläsning. . .

Förra gången introducerades begreppet komplexitet. Idag ska vi titta på den snällaste klassen av problem – de vars lösningar datorer faktiskt är bra på att behandla. Mer komplicerade problem kommer i senare föreläsningar.

Det är av *enorm* vikt, även praktiskt sett, att veta om ens algoritm är effektiv eller ej – onödigt komplicerade algoritmer gör att våra program blir långsamma och gör dem mindre använd- och säljbara.

Definition

Formellt:

$$\mathcal{P} = \bigcup_k \mathrm{TIME}(n^k)$$

Informellt: Mängden av alla problem som kan avgöras inom polynomiell tid av en *deterministisk en-bands-Turingmaskin*.

Det är viktigt att TM:en är deterministisk (konverteringen från icke-deterministisk kostar oss exponentiell tid – alldeles för dyrt). Att vi kräver ett band är för att vi inte vill lämna detaljer åt slumpen – realistiskt sett är det inte helt avgörande, då en två-bands-TM konverteras till en en-bands på kvadratisk tid (relativt billigt, alltså).

Viktiga egenskaper hos $\mathcal{P}$

1. $\mathcal{P}$ är samma för alla beräkningsmodeller som är polynomiellt ekvivalenta till den deterministiska en-bands-Turingmaskinen och
2. $\mathcal{P}$ är ungefär motsvarande de problem som kan lösas av en modern dator.

Första egenskapen innebär att $\mathcal{P}$ är "robust": klassen är oberoende av detaljer i vår beräkningsmodell.

Andra egenskapen innebär att vi i praktiken kan lösa problem som ingår i $\mathcal{P}$. Problemet kan lösas av en dator på n^k tid, för något värde på k. Om k är stort, innebär det att problemet är väldigt svårlöst, men forfarande inom grepphåll för datorer.

Fråga 1

En applicerad tolkning av Moores lag (som handlade om antalet transistorer som får plats på en viss yta) säger oss att datorers hastighet i stort sett fördubblas med 18 månaders intervall.

Låt oss säga att vi har två problem, A och B. $\mathcal{O}(A) = n$ och $\mathcal{O}(B) = 2^n$. Tydligen är $A \in \mathcal{P}$ men $B \notin \mathcal{P}$. Om vi har en dator som kan klara av indata av längderna a respektive b på en timme, hur mycket indata kan då en dubbelt så snabb dator klara av?

Svar på fråga 1

Om vi med en dator kan klara av indata av längden a på en timme då komplexiteten är n, kan en dubbelt så snabb dator klara av indata med längden $2a$.

Är dock komplexiteten 2^n och vi kan hantera indata av längden b, kan vi med en dubbelt så snabb dator bara hantera indata av längden $b + 1$ på samma tid.

Uppenbarligen är problem i $\mathcal{P}$ mycket mer lätthanterliga, även för framtida datorer!

Några ord innan vi börjar...

För att avgöra om ett problem är i $\mathcal{P}$, måste vi undersöka komplexiteten av algoritmen som löser det. Vi gör detta med avseende på längden av indata. Hela algoritmen måste gå att lösa på polynomiell tid av en deterministisk TM.

Många problem ingår uppenbart i $\mathcal{P}$, som exempelvis att gå igenom en lista och räkna ut ett medelvärde av elementen, att sortera en lista (om man inte gör någon ytterst kreativ och dum speciallösning) och så vidare. Sådana kommer vi inte bekymra oss med, de är inte intressanta nog.

Några fler ord innan vi börjar...

Hittills har vi bara konstaterat att det går att skapa strängrepresentationer av Turingmaskiner och annat, men hur dessa ser ut är viktig då vi diskuterar komplexitet. Vi förstör exempelvis mycket för oss själva om vi använder onödigt krångliga representationer – exempelvis den *unära* representationen av tal (talet n representeras av n stycken ettor).

När vi resonerar om algoritmer, antar vi alltid att representationen är så minimal och vettig som möjligt.

PATH-problemet

Vi har en riktad graf (alla kanter i grafen har en riktning) kallad G. Vi vill avgöra om det finns en riktad väg att gå mellan noderna s och t som båda finns i grafen G. Detta kan skrivas som ett språk på följande sätt:

$$\text{PATH} = \{\langle G, s, t\rangle \mid G \text{ är en riktad graf som har en riktad väg från } s \text{ till } t\}$$

Det känns uppenbart som ett problem som en dator borde kunna lösa, men vad vi vill veta är först och främst *hur* och dessutom hur detta kan göras på ett effektivt sätt.

PATH-problemet, dålig brute-force-lösning

Den lösning som är absolut lättast att komma på är "kolla bara alla möjliga vägar mellan noder och se om vi har en mellan s och t". Vi kan öka effektiviteten genom att säga att vägarna vi testar får vara maximalt m långa, om m är antalet noder i grafen. Då vet vi att vi inte kan ha upprepningar av noder.

Problemet är att vi ändå har en algoritm som i stort sett måste testa m^m olika vägmöjligheter. Exponentiell komplexitet för denna lätta uppgift!

PATH-problemet, bra lösning

Vi har redan diskuterat den smarta lösningen på det här problemet! Visst kommer ni ihåg hur vi bevisade att E_{DFA} är avgörbart?

För att lösa PATH på ett effektivt sätt, följ denna algoritm:

M = "Vid indata $\langle G, s, t \rangle$ där G är en riktad graf med noderna s och t:

1. Markera noden s.
2. Upprepa tills inga fler noder kan markeras:
3. Om det finns en kant (a, b) från en markerad nod a till en omarkerad nod b, markera b.
4. Om t är markerad, *acceptera*. *Refusera* annars.

PATH-problemet, analys

Vi har hittat en algoritm M som löser PATH-problemet. Vi måste nu analysera den för att avgöra att den är effektiv nog för att $M \in \mathcal{P}$.

Initieringen (markera s) och avslutande testet (kolla om t har markerats) utförs endast en gång vardera och kan implementeras på polynomiell tid utan problem. Vi kan maximalt vara tvungna att loopa över noderna i grafen m gånger, där m är antalet noder i grafen.

Att undersöka om det finns en kant mellan de markerade noderna och de omarkerade går att göra på polynomiell tid om vi har valt en vettig representation av grafer.

Således är $M \in \mathcal{P}$.

RELPRIME-problemets lärdom

Bokens teorem 7.15 tar upp ett problem gällande att avgöra om två tal är *relativt prima* (de har bara 1 som gemensam delare). Vi behöver inte fördjupa oss i det här och nu (men titta gärna på det själva), eftersom det inte ingår i kursen enligt läsanvisningarna.

Den viktigaste lärdomen är dock att vi måste fundera på representationer. Eftersom Turingmaskinen bara behandlar strängar, måste vi konvertera talen till strängar. Sättet vi gör det på påverkar längden av indatan, som ju är det vi relaterar vår algoritms komplexitet till. Det är något man kanske lätt glömmer bort (”det är ju bara ett tal”), så det kan vara värt att ägna det en extra tanke.

Om jag gillar de kontextfria grammatikerna så mycket...

...varför gifter jag mig inte med dem?

Vi ska än en gång titta på de kontextfria grammatikerna. På föreläsningen om avgörbara problem bevisade vi lite snabbt att det går att avgöra om en sträng har genererats av en kontextfri grammatik. Vi gjorde det genom att resonera kring att det bara fanns ett ändligt antal deriveringar vi var tvungna att testa, förutsatt att vi hade en grammatik på Chomsky-normalform.

Problemet är att detta ändliga antal är exponentiellt stort – alltså är den algoritmen inte i $\mathcal{P}$!

Repetition: Chomsky-normalform

Chomsky-normalform är en förenklad form av kontextfria grammatiker, som är mycket praktiska att använda i algoritmer. En kontextfri grammatik är på Chomsky-normalform om varje regel är på formen:

$$\begin{aligned} A &\rightarrow BC \\ A &\rightarrow a \end{aligned}$$

a är en terminal och A, B, C är icke-terminaler, men B, C får inte vara startvariabeln. Dessutom tillåts att startvariabeln har en produktion som leder till ϵ.

Bokens **teorem 2.9** säger att alla kontextfria språk kan genereras av grammatiker på Chomsky-normalform. Detta bevisas genom konstruktion, vilket den intresserade kan titta på.

Dynamisk programmering

Dynamisk programmering är en teknik man använder när man har jobbiga problem att hantera – tekniken innebär att man löser små delproblem för att efter ett tag ha löst det initiala stora problemet. Vi sparar delproblemens lösningar för att inte göra samma bearbetning flera gånger.

Att be datorn ge en sekvens av Fibonaccitalen ($F_n = F_{n-1} + F_{n-2}, F_0 = 1, F_1 = 1$) är ett enkelt exempel på när det verkligen lönar sig att spara undan lösningar på vägen, på grund av allt dubbelarbete som annars krävs.

Vår algoritms genväg

Vi håller koll på vilka variabler som genererar delsträngarna av w (strängen vi skall se om vår CFG G kan ha genererat)!

Vi skapar en tabell som är $n \times n$ stor (n är längden av w). För $i \leq j$ är elementet i tabellen på position (i, j) den mängd variabler som genererar delsträngen $w_i w_{i+1} \ldots w_j$. Då $i > j$ är tabellens celler tomma.

Först fyller vi i elementen för delsträngar som är av längden 1, sen av längden 2 och så vidare, tills vi har täckt in hela strängen. Information om de kortare delsträngarna används för att undersöka de längre delsträngarna.

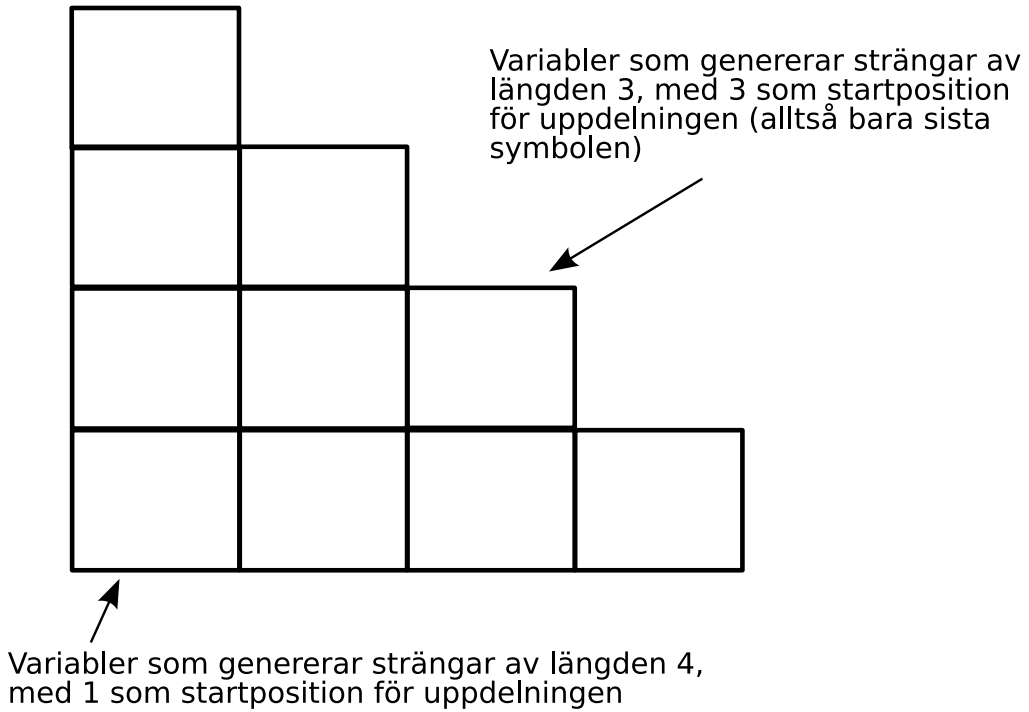
Mer om vår genväg

Antag att vi redan vet vilka variabler som genererar alla delsträngar av längden k. Vi vill nu kolla vilka som genererar delsträngar av längden $k+1$. Om vi inte tillåter att någon del är tomma strängen, finns det k sätt att dela upp en sträng av längden $k+1$.

För varje av dessa k uppdelningar kollar vi varje regel på formen $A \to BC$ och ser om B genererar första biten och C den andra. I så fall vet vi att vi kan generera den aktuella delsträngen med hjälp av variabeln A och för in detta i tabellen.

Vi börjar givetvis med reglerna på formen $A \to b$, alltså de som genererar strängar av längden 1. Resten faller helt enkelt på plats!

Ett exempel som säger minst 1030 ord



Algoritmen under körning

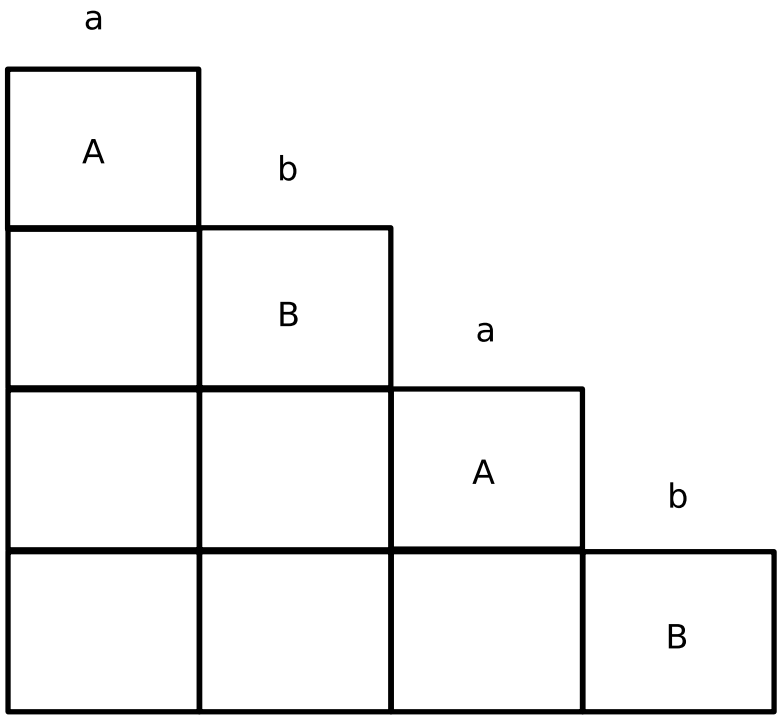
Vi har en CFG, som är på Chomsky-normalform, med följande regler:

$$
\begin{aligned}
S &\rightarrow \epsilon \mid AX \mid YB \mid AB \mid BA \\
Y &\rightarrow AS \\
X &\rightarrow BT \\
T &\rightarrow XA \mid BA \\
A &\rightarrow a \\
B &\rightarrow b
\end{aligned}
$$

Våra terminaler är $\{a, b\}$. Vi vill nu använda algoritmen för att testa om strängen $abab$ kan genereras av grammatiken.

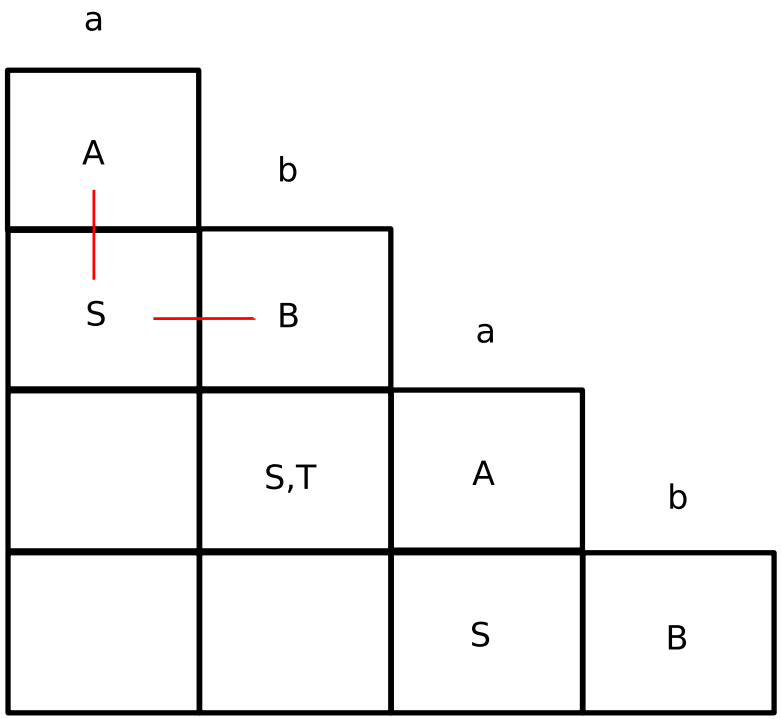
Algoritmen under körning, forts.

Först betraktar vi delsträngarna av längden 1 – vi fyller alltså i de yttersta ”lådorna” i figuren:



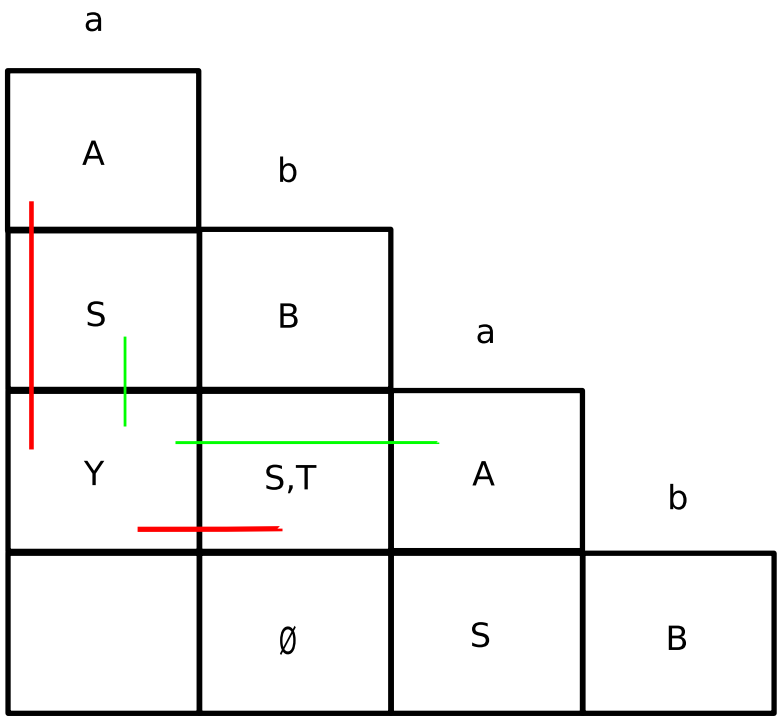
Algoritmen under körning, forts.

Nu betraktar vi delsträngar av längden 2 (vi arbetar mot nedre vänstra hörnet), alltså "lådorna" med index $(1,2),(2,3),(3,4)$. På position $(1,2)$ lägger vi variabeln S, eftersom vi har en regel $S \rightarrow AB$ och vi vet att dessa (A och B) krävs för att kunna skapa delsträngen ab. På liknande vis resonerar vi kring de övriga "lådorna".



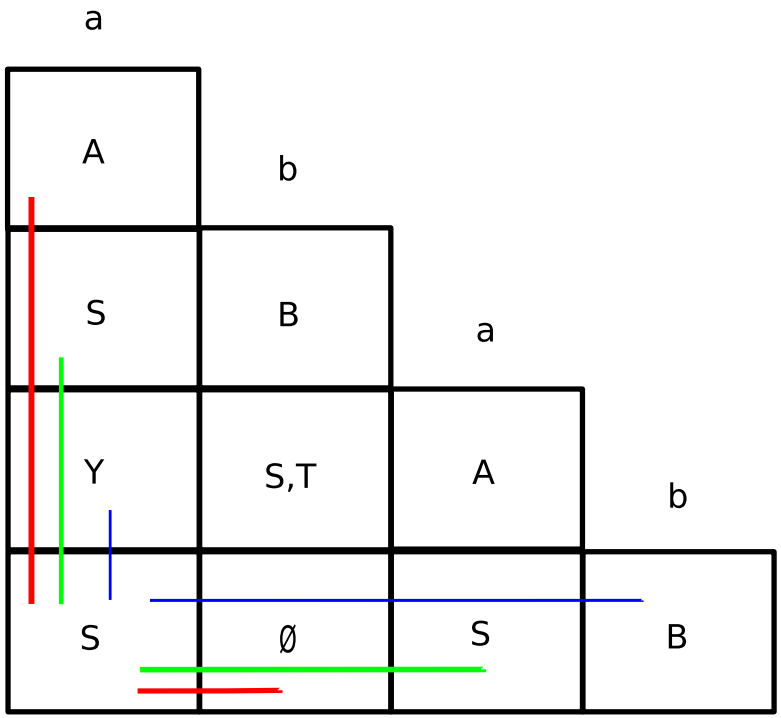
Algoritmen under körning, forts.

I steg 3, då vi betraktar "lådorna" $(1,3)$ och $(2,4)$ börjar det bli fler kombinationer att testa – vi måste nu för $(1,3)$ se om det finns en regel $R \to KL$ där antingen $K \in (1,1)$ och $L \in (2,3)$ *eller* $K \in (1,2)$ och $L \in (3,3)$.



Algoritmen under körning, forts.

Fjärde och sista steget går ut på att testa alla uppdelningar där vi får en sträng av längden 4. Om det visar sig att startsymbolen S återfinns bland de variabler som kan skapa en sådan uppdelning, innebär det att grammatiken kan generera strängen!



Algoritmen, del 1

Algoritmen är rätt lång, så vi delar upp den i tre bitar. Första biten tar hand om specialfallet som tomma strängen innebär:

D = "Vid indata $w = w_1 w_2 \ldots w_n$:

1. Om $w = \epsilon$ och $S \to \epsilon$ är en regel, *acceptera*.

Denna del är uppenbart implementerbar på polynomiell tid.

Algoritmen, del 2

Initiering av tabellen, så att den har någon vettig information.

2. För $i = 1$ till n (längden av strängen):
3. För varje variabel A:
4. Testa om $A \rightarrow b$ är en regel, där $b = w_i$.
5. Om så är fallet, placera A i $table(i, i)$.

Om vi har v antal variabler, kommer denna loop att behöva ungefär vn steg. v är oberoende av n och således är komplexiteten $\mathcal{O}(n)$, alltså polynomiell.

Algoritmen, del 3

Algoritmens huvuduppgift:

6. För $l = 2$ till n (l är delsträngens längd):
7. För $i = 1$ till $n - l + 1$ (i är delsträngens startpos.):
8. Låt $j = i + l - 1$ (j är delsträngens slutpos.)
9. För $k = i$ till $j - 1$ (k är uppdelningens position):
10. För varje regel $A \rightarrow BC$:
11. Om $table(i, j)$ innehåller B och $table(k + 1, j)$ innehåller C, lägg A i $table(i, j)$
12. Om S finns i $table(1, n)$, *acceptera* annars *refusera*.

Komplexiteten är här $\mathcal{O}(n^3)$ eftersom vi har nästlade loopar i tre nivåer, som allihop är beroende av n (stegen 6, 7 och 9). Således är algoritmen $D \in \mathcal{P}$!

Sammanfattning

Alla algoritmer som kan lösas på polynomiell tid utgör klassen $\mathcal{P}$.

Algoritmer i $\mathcal{P}$ kan lösas effektivt av datorer, och är således att föredra framför mer invecklade algoritmer.

Vi har sett några algoritmer som är i $\mathcal{P}$, speciellt en som är en förbättring av en tidigare algoritm som krävde exponentiellt lång tid.