Hva er datamaskinens fundamentale begrensninger?

Jan Arne Telle
Institutt for Informatikk, UiB
telle@ii.uib.no

1 Introduksjon

Det kan argumenteres at all forskning innen informatikk har som mål å utvide grensene for hva datamaskinen kan utrette, det være seg bedre brukergrensesnitt, hurtigere dataoverføring, sikrere programvare eller raskere tungregning.

Et spørsmål av stor filosofisk interesse er om det finnes oppgaver en datamaskin ikke kan, og aldri vil kunne, utrette. Dette spørsmålet ble avklart av Alan Turing for over 60 år siden, og jeg vil i dette innlegget gi en forkortet framstilling av hans argument for at det finnes veldefinerte problemstillinger som beviselig ikke har en algoritme (løsningsmetode), og dermed ikke kan løses ved datamaskin. Et annet spørsmål av stor praktisk betydning er hvilke oppgaver datamaskinen kan utføre innen gitte tidsrammer, det være seg millisekunder eller århundreder. Innen fagfeltet algoritmer og kompleksitetsteori har man siden 1960-årene studert dette spørsmålet fra et teknologiuavhengig standpunkt, og i dette innlegget vil jeg vise eksempler både på oppgaver som beviselig har en effektiv algoritme og oppgaver for hvilke vi kjenner algoritmer men ingen effektiv algoritme. Dog har vi ikke bevis for at de sistnevnte oppgaver ikke også kan ha effektive algoritmer, og dette leder oss til et av de største uløste problemer innen informatikk og kontemporær matematikk, spørsmålet om P=NP.

2 Uløsbare problemer

En algoritme er en samling enkle instruksjoner for å utføre en bestemt oppgave, altså en formell oppskrift, noe vi alle bruker når vi for eksempel utfører en divisjon. I 1900 spurte matematikeren David Hilbert etter en algoritme for å bestemme om et polynom hadde heltallige røtter, og han ville nok blitt svært overrasket om han fikk vite, slik det ble bevist i 1970, at en slik algoritme ikke eksisterer! Denne oppgaven falt inn under det bredere temaet om formalisering av matematikken, og både Bertrand Russel og Kurt Gødel ga bidrag i den sammenheng. Alan Turing, kjent for blant annet sitt arbeid med å knekke det tyske militærets meldingskoder under 2.verdenskrig, og Alonzo Church, framla begge hver sin definisjon av begrepet algo-

ritme/beregnbarhet i 1936. Da begge disse og alle senere definisjoner har vist seg å være i totalt samsvar, har Church-Turing tesen blitt stående som den godtatte definisjon av begrepet algoritme. Vi vil her ikke definere tesen presist, men nøye oss med å gi dens konklusjon: "De oppgavene som har en algoritme, og dermed er beregnbare, tilsvarer de oppgavene for hvilke det finnes et program som en datamaskin kan utføre."

Turing viste også at det fantes veldefinerte oppgaver som ikke har noen algoritme, og vi vil i det følgende fokusere på problemet han kalte "The Halting Problem": Gitt et program P, bestem om det under utføring vil gå i en uendelig løkke, og dermed aldri terminere (stoppe). Om vi antar at vi kunne programmere en datamaskin slik at Terminer(P) er en boolsk funksjon som gir svaret Sant dersom programmet P terminerer, og Usant dersom P aldri terminerer, ville dette vært til stor nytte, noe de fleste programmerere (og hjelpelærere i I110) vil forstå. Men med hjelp av Georg Cantor's diagonaliseringsargument beviste Turing at en slik funksjon Terminer(P) ikke kan eksistere, for da kunne vi nemlig ha skrevet følgende program D:

```
program D
begin
L: if Terminer(D) go to L
end
```

Vil program D terminere? Den interesserte leser burde forsøke å avgjøre dette på egen hånd før hun leser videre. Hvis svaret er Ja, så er Terminer(D) Sant, og vi ser at D vil gå i en uendelig løkke og aldri terminere. Dersom svaret er Nei, så er Terminer(D) Usant og D terminerer. Altså en motsetning, og vi må konkludere med at funksjonen Terminer(P) ikke kan programmeres slik at den gir det rette svaret for enhver P, altså kan ingen datamaskin løse Halting Problemet.

3 Tidskompleksitet

Oppgavene som datamaskiner utfører ser ut til å øke for hver dag som går. Datajournalistikken fokuserer ofte på de nye maskinene og presenterer deres MegaHertz, GigaBytes og TerraFlops. Dette selvfølgelig fordi en ny maskin med intern klokkesykel ti ganger raskere enn den gamle maskin, hvertfall i teorien vil kunne utføre programmene ti ganger raskere. Og den teknologiske utviklingen av maskinvare har vært formidabel, maskinhastigheten kan røfft sies å ha blitt fordoblet hvert tredje år. Men det antaes at i de fleste felt har forbedringer på programvare, ved hjelp av raskere algoritmer, bidratt minst like mye til utviklingen.

La oss ta et eksempel på en enkel algoritmeteknikk som kalles binært søk: anta at vi har n kort liggende foran oss i en lang rekke fra venstre mot høyre, hver med et heltall påskrevet, men med forsiden ned, og med kortene sortert fra lavest lengst til venstre i rekken til høyest lengst til høyre. Vår oppgave er å raskt bestemme om det finnes et kort med tallet x. En naiv algoritme som snur kort i rekkefølge fra venstre

mot høyre, må i verste fall snu n kort før et eventuellt negativt svar kan gies. Istedet, siden kortene er sortert, kan vi ved å snu kun det midterste kortet redusere problemet fra et spørsmål om n kort til et spørsmål om $n/2$ kort. Etter å ha snudd i kort vil algoritmen som alltid snur det midterste kortet og fortsetter i enten den lavere eller høyere delen, enten ha funnet svaret, eller stå igjen med et spørsmål om $n/2^i$ kort. Siden $2^{log_2(n)} = n$ ser vi at vi selv i verste fall ikke snur mer enn $log_2(n)$ kort. Altså trenger vi for eksempel ikke snu mer enn 10 kort ut av 1000.

Tidskompleksiteten til en algoritme gir en øvre grense for hvor mange operasjoner (=tid) algoritmen vil måtte utføre og uttrykker dette som en funksjon av oppgavestørrelsen n. For eksempel er det enkelt å finne en algoritme som sorterer n tall uten å foreta mer enn n^2 sammenlikninger. Det er vanskeligere å finne en som garantert aldri bruker mer enn $2nlog_2(n)$ sammenlikninger. Dette vil gi en optimal algoritme da man kan vise at dette antallet, opp til en multiplikativ konstant faktor, er en nedre grense for antall sammenlikninger som er nødvendig for å sortere n tall.

Tidskompleksitet	Dagens maskin	Maskin X100	Maskin X1000
n	N	100N	1000N
$nlog_2(n)$	N	14N	100N
n^2	N	10N	32N
2^n	N	N+7	N+10

Om vi antar at vi med dagens datamaskin ved hjelp av en algoritme med tidskompleksitet 2^n iløpet av en time kan løse en oppgave på størrelse N, ser vi av tabellen over at med en ny maskin som er 1000 ganger raskere så kan vi med samme algoritme iløpet av den samme timen ikke løse en oppgave på størrelse mer enn $N + 10$. Altså nesten ingen forbedring selv om maskinen er 1000 ganger raskere! Dette i motsetning til om algoritmen har tidskompleksitet n hvor vi får full uttelling og kan løse, hvertfall i teorien, en oppgave av størrelse $1000N$. Legg merke til at N er forskjellig i hver rekke. Argumentet er teknologi-uavhengig og viser hvor viktig det er å finne algoritmer med lav tidskompleksitet.

4 P versus NP

Grovt sett sier vi, som Jack Edmonds gjorde i 1965, at en algoritme er effektiv dersom tidskompleksiteten er polynomisk i n, slik som n^k for en konstant k. Sikkert er det at en algoritme som bruker 2^n operasjoner ikke er av nytte for de verdiene av n som de fleste applikasjoner idag baserer seg på. En oppgave som har en effektiv algoritme sies å tilhøre klassen P av oppgaver, og tilsvarer de oppgaver som har et bevis (en løsning) som kan finnes i polynomisk tid. De oppgaver som har et gitt bevis hvor korrektheten til beviset kan verifiseres (sjekkes) i polynomisk tid, sies å tilhøre klassen NP av oppgaver. Intuitivt skulle man tro det var mye lettere å **verifisere** et gitt bevis enn det er å **finne** et bevis, og av den grunn tror de fleste at det er mange oppgaver i NP som ikke er i P. Men dette har ingen klart å bevise, og den som gir svaret på dette spørsmålet er

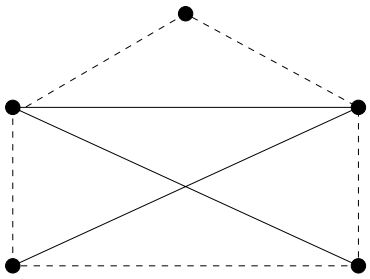

Figure 1: Grafen over med 5 noder og 8 kanter har en Hamiltonsk sykel, vist med stiplede kanter, men den har ikke en Eulersk sykel.

garantert evig berømmelse!

La oss se noen eksempler fra grafalgoritmer. En graf er en samling noder med en binær kant-relasjon, og brukes til å modellere mange praktiske problemstillinger, f.eks et datanettverk hvor vi har en node for hver maskin og en kant mellom to noder dersom disse maskinene er direkte tilkoblet hverandre. Gitt en graf G med n noder ser vi på de to oppgavene å bestemme om grafen har en Eulersk sykel, slik Leonhard Euler gjorde i 1736, og om den har en Hamiltonsk sykel, slik William Hamilton gjorde i 1859. En Eulersk sykel er en vandring fra node til node kun via kanter, som besøker hver **kant** nøyaktig en gang, og returnerer til utgansgpunktet. En Hamiltonsk sykel er en vandring fra node til node kun via kanter, som besøker hver **node** nøyaktig en gang, og returnerer til utgansgpunktet.

Siden en slik Eulersk sykel i grafen entrer en node langs en kant, forlater noden langs en annen kant, og besøker hver kant nøyaktig en gang, må antall kanter enhver node deltar i være et partall for at en Eulersk sykel skal eksistere. For en sammenhengende graf kan dette vises å være også en tilstrekkelig betingelse. Observasjonen kan brukes til å utvikle en n^2 algoritme som finner en Eulersk sykel i en gitt graf med n noder, eller bestemmer at en slik sykel ikke eksisterer. Hamiltonske sykler har vi derimot ingen god forståelse for og vi kjenner ingen polynomisk algoritme for dette problemet. Gitt en vandring i grafen er det selvsagt enkelt å verifisere om vandringen er Hamiltonsk, så problemet er i klassen NP. I tillegg er Hamiltonsk sykel et såkalt NP-komplett problem, hvilket betyr at om noen skulle finne en polynomisk algoritme for dette problemet, så følger det automatisk at P=NP.

Teorien om NP-komplette problemer ble utviklet av Stephen Cook og Leonid Levin på begynnelsen av 1970-tallet, og selv om enormt mye arbeid har blitt lagt ned i det å avgjøre om P=NP, er dette spørsmålet fortsatt ubesvart. NP-komplette problemer florerer i praktisk arbeide. Om man støter på en oppgave som man ikke klarer å skrive et effektivt program for, så kan man forsøke å vise at det er NP-komplett (ved en **polynomisk reduksjon** fra et annet NP-komplett problem.) Dermed kan man unngå sjefens nedlatende bemerkninger ved å vise til det faktum at verdens fremste informatikere og matematikere heller ikke klarer å skrive et effektivt program for denne oppgaven.