Kompleksitetsteori – reduksjoner

En slags liten oversikt, eller huskeliste, for kompleksitetsteorien i INF 4130. Ikke ment å være verken fullstendig eller detaljert, men kanskje egnet til å gi noen knagger å henge det mer teoretiske på.

1 Problemer

Det finnes mange slags databehandlingsproblemer, med ulik vanskelighetsgrad: alt fra helt enkle, til vanskelige, og videre opp til uavgjørbare (og faktisk også enda verre, såkalt sterkt uavgjørbare problemer).

1.1 Desisjonspoblemer

Problemene er vel oftest av en slik type at vi ønsker å finne den beste løsningen på et eller annet, for eksempel korteste vei fra *a* til *b*, største uavhengige mengde av noder i en graf *G*, eller liknende…, såkalte *søke-* eller *optimaliseringsproblemer*. Det er mye man kan søke etter eller forsøke å optimalisere, så slike problemer er opplagt av ganske ulik karakter.

For å gjøre det enklere for oss selv, for at alle problemene skal være av samme type, begrenser vi oss til å se på problemer hvor svaret er JA eller NEI, såkalte *desisjonsproblemer*. Når alle problemene er av samme type kan vi enklere lage teorier og matematikk om dem.

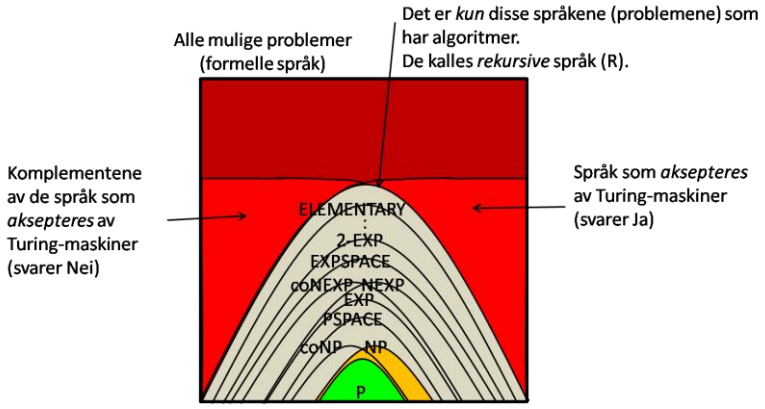
Denne overgangen fra søkeproblemer til desisjonsproblemer kan vi lett gjøre ved å stille spørsmål á la «finnes det en vei fra *a* til *b* som er kortere enn *k*?», da får vi jo et JA/NEI-svar. (Og vi kan binærsøke på denne *k*en for å finne den beste.)

1.2 Problemklasser

Problemer har som nevnt ulik vanskelighetsgrad, og de kan deles inn i klasser avhengig av denne vanskelighetsgraden, problemer med nogenlunde samme vanskelighetsgrad havner i samme klasse. Vi skal bare se på noen få slike klasser, vi konsentrerer oss om de tre-fire viktigste:

1. Klassen ***P*** – desisjonsproblemene som har polynomiske algoritmer.
2. Klassen ***NP*** – desisjonsproblemene hvor et JA-sertifikat kan sjekkes i polynomisk tid.
3. De ***NP***-komplette problemene – de vanskeligste problemene i ***NP***.
4. De uavgjørbare problemene (og verre).

Klassene 1,2 og 4 er merket av på figuren under med grønt, gult og rødt. Vi skal ikke bry oss med kompleksitetsklassene i det grå området (de inneholder for det meste relativt «eksotiske» problemer). Klassen ***P*** og de ***NP***-komplette problemene er delmengder av klassen ***NP***, men det nøyaktige forholdet mellom ***NP***, ***P*** og de ***NP***-komplette problemene er fortsatt ukjent. Dog tror vel de fleste at ***P*** er en ekte delmengde av ***NP*** (***P*≠*NP***).



Figur 1 **De aktuelle kompleksitetsklassene angitt med grønt (*P* – desisjonsproblemene med polynomiske algoritmer), gult (klassen *NP*, som inneholder både *P* og de *NP*-komplette problemene) og rødt/mørkerødt (uavgjørbare problemer). Vi skal ikke bry oss med det grå området; disse problemene er vanskeligere enn problemene i *NP*, men de har algoritmer.**

2 Polynomiske reduksjoner

Reduksjoner er vårt hovedverktøy for å bygge opp kompleksitetsklasser og rangere disse etter vanskelighetsgrad. All kompleksitetsteori dreier seg i bunn og grunn om å vise at et problem er minst like vanskelig som et annet, eventuelt at to problemer er like vanskelige; dette gir oss nemlig en inndeling av problemene i klasser og en rangering av klassene.

Det finnes mange typer reduksjoner. Siden vi anser to problemer for like vanskelige om begge har eller ikke har polynomiske algoritmer, er det *polynomiske reduksjoner* vi skal bruke (polynomisk tid). Lager vi en polynomisk reduksjon *fra* et problem *Q til* et problem *R*, forteller det oss at problem *R* vanskeligere eller like vanskelig som *Q*, på et polynom nær:

$$\text{Vanskelighetsgrad}(Q) \leq_{\text{poly}} \text{Vanskelighetsgrad}(R) \ .$$

At vi har en polynomisk reduksjon fra *Q* til *R*, skrives vanligvis slik ($\propto$ er en slags $\leq$):

$$Q \propto R \ .$$

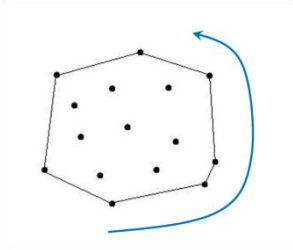
(Navnet «reduksjon» er en historisk betegnelse som kanskje forvirrer litt; det kan jo høres ut som *R* bør være lettere enn *Q*, siden det er en reduksjon fra *Q* til *R*, men slik er det altså ikke. *R* er vanskeligere eller like vanskelig som *Q*, på et polynom nær, om vi har en polynomisk reduksjon fra *Q* til *R*.)

For å vise hvordan vi lager reduksjoner, og forhåpentligvis gjøre det så lite mystisk som mulig, skal vi se på et enkelt eksempel med to kjente problemer: å sortere heltall (SORTING) og å finne konvekse innhyllinger (CONVEX HULL). Dette er begge kjente problemer, og vi skal lage en reduksjon *fra* SORTING *til* CONVEX HULL. Dette er ikke desisjonsproblemer, men det er ingen ting i veien for å redusere fra et problem til et annet selv om det ikke er desisjonsproblemer. Begge problemene er løsbare i polynomisk tid, det er heller ingen ting i veien for å redusere fra et polynomisk problem til et annet.

(Poenget med desisjonsproblemer er å ha en veldefinert basis for kompleksitetshierarkiet i Figur 1. Problemene SORTING og CONVEX HULL ligger altså utenfor det kompleksitetshierarkiet ettersom de ikke er desisjonsproblemer. De tilhører en klasse av problemer som er ekvivalent med ***P***, men som inneholder alle typer av problemer som kan løses i polynomisk tid.)

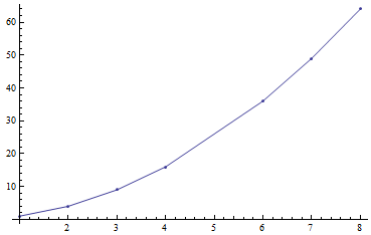
Eksempel 2.1.

En *konveks innhylling* av en punktmengde *N* i planet, $N \in \mathbf{N}^2$, er det minste konvekse polygon *H* som inneholder punktene i *N*. For å få en entydig representasjon av polygonet *H*, må hjørnene angis enten med eller mot klokka, vi velger mot klokka.



Figur 2 **Et konvekst polygon *H* som innhyller en punktmengde *N*, hjørnene i polygonet angis mot klokkerentingen.**

En algoritme som finner slike konvekse innhyllinger kan brukes til å sortere heltall. La oss si at vi ønsker å sortere tallene {6, 3, 2, 7, 1, 4, 8}, og se hvordan det gjøres. Vi *transformerer* mengden av heltall til en punktmengde i planet ved å gjøre tallet *n* om til punktet (n,n^2). Disse punktene beskriver et parabel-aktig polygon, som er konvekst, slik at alle punktene må være hjørner i en konveks innhylling av disse punktene. Ettersom rekkefølgen på hjørnene i den konvekse innhyllingen gis i retning mot klokka, får vi tallene ut i sortert rekkefølge om vi starter med hjørnet som har lavest *y*-koordinat, og leser av *x*-koordinaten til hjørnene i innhyllingen.



Figur 3 **De transformerte punktene beskriver et parabel-aktig polygon, som er konvekst, slik at alle punkter må være hjørner i den konvekse innhyllingen.**

Vi kan altså sortere heltall ved å gjøre dem om til punkter i planet på en enkel måte, og bruke en algoritme for å finne konvekse innhyllinger. La `ConvexHull` være en algoritme for CONVEX HULL, røff pseudokode for en algoritme `Sorting` for SORTING vil jo da bli noe á la:

```
proc Sorting(N)
{
   P = Transform(N)           // Transformerer tallene: n til punkt (n,n^2)
   H = ConvexHull(P)
   l = <den h i H med lavest y-koordinat>                  // O(n)
   S = <les av x-koordinat til hjørnene i H, med l først>  // O(n)
   return (S)
}
```

Vi har lagd algoritmen `Sorting` ved å bruke `ConvexHull` som subrutine. Dette er en polynomisk reduksjon fra SORTING til CONVEX HULL. Siden vår algoritme `Sorting` bruker `ConvexHull` som subrutine, må `ConvexHull` være en minst like kraftig algoritme som `Sorting`, på et polynom nær. Sagt på en annen måte: problemet CONVEX HULL må være minst like vanskelig som problemet SORTING, på et polynom nær. Vi kan ikke løse et vanskelig problem med en svak subrutine (når vi bare tillater en enkel transformasjon og ett kall på subrutinen). ■

Reduksjoner er i prinsippet ikke annet enn enkel bruk av subrutiner. Vi har to problemer *Q* og *R* og lager en polynomisk reduksjon *fra Q til R,*

$$Q \propto R$$

ved å lage en algoritme for problem *Q* som bruker en algoritme for *R* som subrutine, etter følgende mal:

```
proc Q(Q_Input)
{
   R_input = Transform(Q_Input) // Q_input -> R_input, polynomisk tid
   output = R(R_input)          // Kall subrutine R med transformert input
   return(output)               // løsning på Q fra kallet på R
}
```

dette forteller oss at

$$\text{Vanskelighetsgrad}(Q) \leq_{\text{poly}} \text{Vanskelighetsgrad}(R)\ .$$

Det er selvfølgelig et krav at transformasjonen, `Transform()`, transformerer instanser av *Q* til instanser av *R* med samme svar, ellers vil vi jo ikke få riktig svar fra subrutinen.

En polynomisk reduksjon fra *Q* til *R* består altså av to steg:

1. en transformasjon av *Q*-instansen til en *R*-instans, i polynomisk tid, og
2. et kall på en subrutine med den transformerte instansen som input.

Polynomiske reduksjoner kan anvendes både praktisk og teoretisk:

- Praktisk brukes de til å lage en algoritme for *Q* med en eksisterende algoritme for *R* som subrutine.
- Teoretisk brukes de til å vise at *R* er vanskeligere eller like vanskelig som *Q*, på et polynom nær. Da trenger vi faktisk ikke ha noen algoritme for *R*, det holder at vi tenker oss en, «Hvis vi hadde hatt en polynomisk algoritme for *R*, så kunne vi lagd en polynomisk algoritme for *Q* med *R* som subrutine.»

3 *NP*-kompletthet

Polynomiske reduksjoner lar oss altså vise at et problem er vanskeligere eller like vanskelig som er annet, på et polynom nær. Nå skal vi bruke slike reduksjoner til å vise at enkelte problemer er ***NP***-komplette.

For å vise at et problem *C* er ***NP***-komplett, må to ting vises:

1. $C \in \mathbf{NP}$, (*C* må ligge i ***NP***.)
2. $\forall B \in \mathbf{NP} : B \propto C$. (Polynomisk reduksjon fra alle *B* i ***NP*** til *C*.)

Det er selvsagt vanskelig å vise punkt 2, ettersom vi må gå igjennom alle problemer *B* i ***NP*** (det er uendelig mange). Heldigvis har vi Cooks teoerm, som sier at SAT (*Satisfiability*) er ***NP***-komplett. Cook viste punkt 1 og 2 for SAT, slik at vi fikk et første ***NP***-komplett problem som vi kan bruke videre. Når *vi* skal vise at et problem *C* er ***NP***-komplett, trenger vi bare et kjent ***NP***-komplett problem *K* (for eksempel SAT), og så viser vi:

1. $C \in \mathbf{NP}$ (*C* må ligge i ***NP***.)
2'. $K \propto C$. (Polynomisk reduksjon fra *K* til *C*.)

(Punkt 2 vil da følge av punkt 2' når *K er* ***NP***-komplett ettersom $\propto$ er en transitiv relasjon; vet vi for eksempel at $\forall B \in \mathbf{NP} : B \propto \text{SAT}$, og $\text{SAT} \propto C$, så følger $\forall B \in \mathbf{NP} : B \propto C$.)

For å vise at et problem *C* er ***NP***-komplett, må vi altså først vise:

1a. at *C* er et desisjonsproblem, og
1b. at et JA-sertifikat for *C* kan verifiseres i polynomisk tid.

Så må vi vise vise

2'. $K \propto C$ (en polynomisk reduksjon fra et kjent ***NP***-komplett problem *K* til *C*) etter følgende mal:

```
proc K(K_Input)
{
   C_input = Transform(K_Input) // Q_input -> R_input, polynomisk tid
   output = C(C_input)          // Kall tenkt C med transformert input
   return(output)               // Løsning på K fra kallet på tenkt C
}
```

Det er selvfølgelig et krav at transformasjonen, `Transform()`, transformerer instanser av *K* til instanser av *C* med samme svar, ellers vil vi jo ikke få riktig svar fra subrutinen. Her er det desisjonsproblemer vi snakker om, så JA-instanser av *K* må transformeres til JA-instanser av *C*, og tilsvarende for NEI-instanser.

Her er subrutinen for *C* en tenkt algoritme. Ideen er at *om C* vært løsbart i polynomisk tid (med en polynomisk algoritme), *så* hadde vi hatt en polynomisk algoritme for *K*, men *K* er ***NP***-komplett og har høyst sannsynlig ingen polynomisk algoritme, altså er det lite sannsynlig at vår tenkte polynomiske algoritme for *C* kan eksistere. Reduksjonen (punkt 2') viser at *C* er vanskeligere eller like vanskelig som det kjente ***NP***-komplette problemet *K*, og siden vi viser at $C \in$ ***NP*** (punkt 1a og 1b), så følger det at *C* er ***NP***-komplett.

4 Uavgjørbarhet

Vi har sett at reduksjoner brukes til å vise at et problem er vanskeligere eller like vanskelig som et annet. Nå skal vi bruke reduksjoner til å vise at enkelte problemer er uavgjørbare.

For å vise at et problem *U* er uavgjørbart, lager vi en reduksjon fra et kjent uavgjørbart problem til vårt problem *U*. I prinsippet er ikke det vanskelig, men vi trenger altså et kjent uavgjørbart problem å redusere fra. Ofte brukes HALTING (Stoppeproblemet), som Turing viste uavgjørbart i 1937.

For å vise at et problem *U* er uavgjørbart, lager vi en reduksjon fra HALTING (eller et annet kjent uavgjørbart problem når vi har fått flere av dem) til *U*, med følgende mal:

```
proc Halting(H_Input)
{
   U_input = Transform(H_Input)  // H_input -> U_input
   output = U(U_input)           // Kall subrutine U med transformert input
   return(output)                // Løsning på HALTING fra kallet på U
}
```

Her behøver ikke reduksjonen være polynomisk. Siden vi bare er interessert i *om* en algoritme eksisterer eller ei og ikke bryr oss om kjøretiden, er det ingen restriksjoner på reduksjonens tids- eller plassbruk. Transformasjonen, `Transform()`, trenger altså ikke være polynomisk, det holder at den er terminerende, men selvfølgelig må JA-instanser av HALTING transformeres til JA-instanser av *U*, og tilsvarende for NEI-instansene.

Her er det i grunn opplagt at subrutinen for *U må* være en tenkt algoritme, vi viser jo at *U* er uavgjørbart. Hadde vi hatt en algoritme for *U*, kunne vi løst Stoppeproblemet, men Stoppeproblemet er uavgjørbart. Reduksjonen viser at *U* er vanskeligere eller like vanskelig som HALTING, og siden vi vet at HALTING er uavgjørbart (ikke har noen algoritme), må også *U* være uavgjørbart (ikke ha noen algoritme).

Oppgave 1.

Vi skal lage en polynomisk reduksjon fra BIPARTIT MATCHING til MAX FLOW. Input til BIPARTIT MATCHING er en bipartit graf $G=(V_1, V_2, E)$ (det er altså to mengder av noder V_1 og V_2, og alle kanter i E går mellom en node i V_1 og en V_2), svaret skal være størrelsen av største matching i grafen. Input til MAX FLOW er et nettverk med kapasiteter, svaret skal være verdien av største flyt i nettverket.

a) Beskriv hvordan transformasjonen gjøres, og tidskompleksiteten av den.

b) Skriv pseudokode for reduksjonen.

c) Hvilket problem må da være vanskeligere eller minst like vanskelig som det andre, på et polynom nær?

[slutt]