13. NP-kompletthet

Forelesning 13

NP er klassen av *ja-nei*-problemer der ethvert ja-svar har et bevis som kan sjekkes i polynomisk tid. Alt i NP kan i polynomisk tid reduseres til såkalt *komplette* problemer i NP. Dermed kan ikke disse løses i polynomisk tid, med mindre *alt* i NP kan det. Ingen har klart det så langt . . .

Pensum

- ☐ Kap. 34. NP-completeness: Innl. og 34.1–34.3
- ☐ Oppgave 34.1-4 med løsning (0-1 knapsack)
- ☐ Appendiks D i pensumheftet

Læringsmål

- [M_1] Forstå *optimering* vs. *beslutning*
- [M_2] Forstå *koding* av en instans
- [M_3] Forstå at løsningen på det binære ryggsekkproblemet *ikke er polynomisk*
- [M_4] Forstå *konkrete* vs *abstrakte* problemer
- [M_5] Forstå repr. av beslutnings-problemer som *formelle språk*
- [M_6] Forstå def. av P, NP og co-NP
- [M_7] Forstå *redusibilitets-relasjonen*
- [M_8] Forstå def. av NP-Hard og NPC
- [M_9] Forstå den konvensjonelle hypotesen om P, NP og NPC
- [M_{10}] Forstå bevis for at CIRCUIT-SAT er i NPC

📚Reduksjon

I denne forelesningen betegner reduksjoner Karp-reduksjoner, eller “many-one” reduksjoner som tar polynomisk lang tid.

Vi ønsker å beskrive problemer sin vanskelighetsgrad. Det er teoretisk mulig å finne absolutt vanskeliggrad av et problem, men det er lite nyttig, så vi vil heller sammenligne og kategorisere problemer.

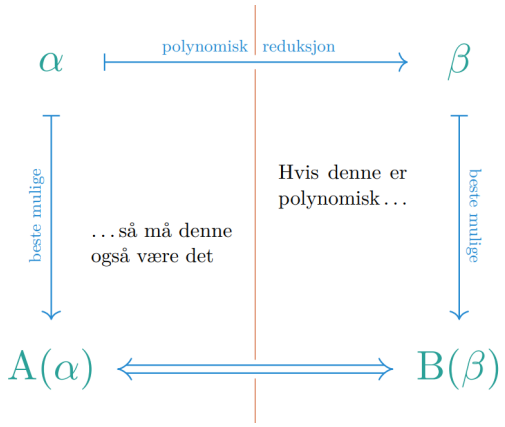
Et eksempel på en Karp-reduksjon fra forelesning:

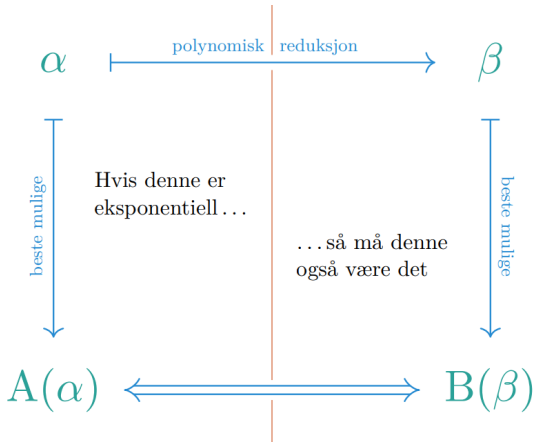
> Gitt en kiste A, og en kiste B hvor nøkkelen til A ligger inni B. Ved reduksjon er det dermed åpenbart at det ikke kan være vanskeligere å åpne A enn det er å åpne B.

Hva “minst like vanskelig” betyr kommer an på reduksjonene, men i vårt tilfelle bruker vi polynomiske reduksjoner som betyr at problemene kan løses i polynomisk tid.

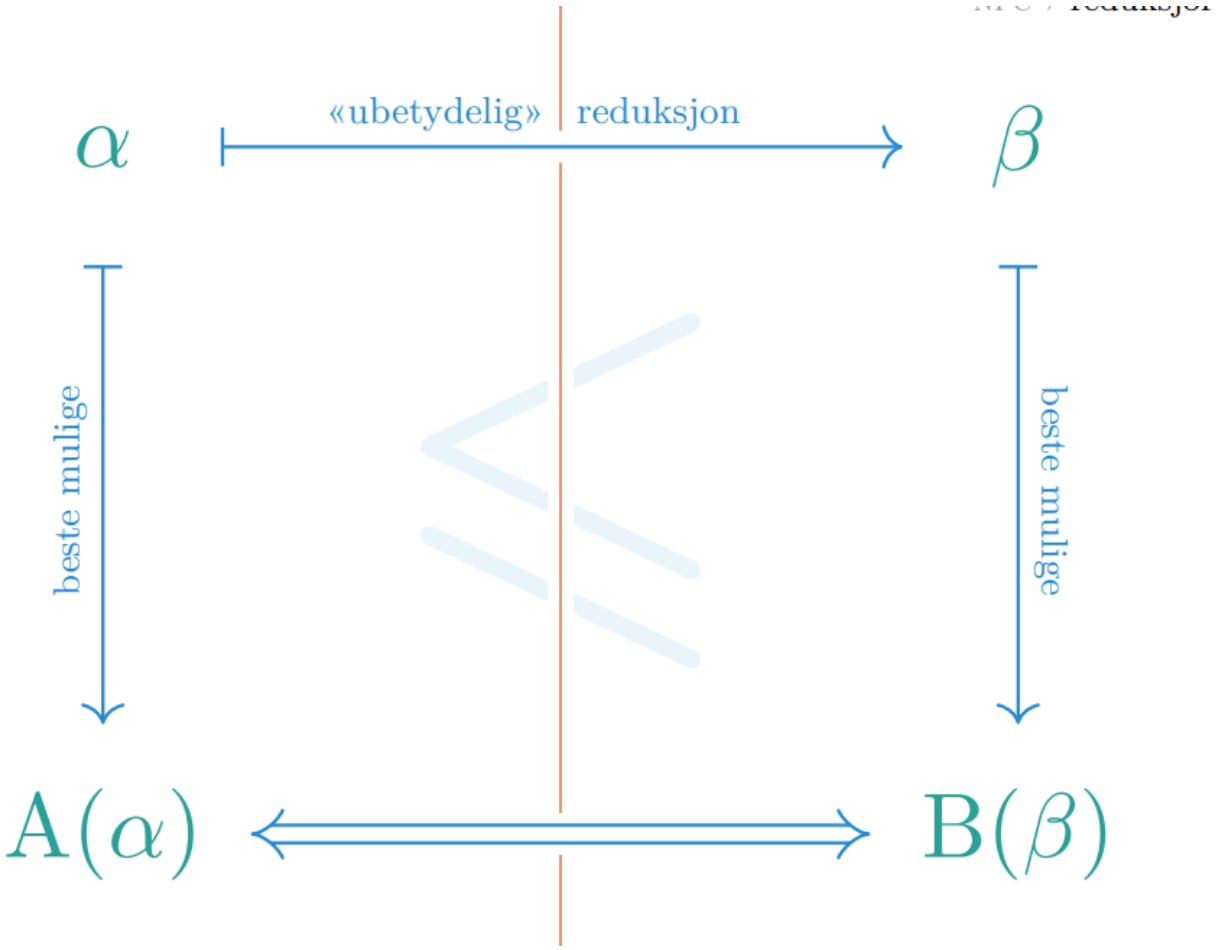
Hvis vi ønsker å løse problemet “har en graf en kort sti mellom node u og v” kan vi redusere dette ned til å finne korteste sti mellom nodene. Det er dermed åpenbart at å finne ut om det finnes en kort sti, er minst like vanskelig som å finne korteste sti.

```
HAS-SHORT-PATH(G, u, v, k)
1  BFS(G, u)
2  return v.d ⩽ k
```





Hvis vi har et problem α som vi kan polynomisk redusere til et annet problem β som også er polynomisk, er det åpenbart at vi kan bruke løsningen $B(\beta)$ for å lage en løsning $A(\alpha)$.



Vi kan redusere Hamilton-sykel problemet ned til Long-Path problemet i polynomisk tid, så det betyr at $HamCycle$ kan umulig være vanskeligere enn $LongPath$, og $LongPath$ er minst like vanskelig som $HamCycle$.

På denne måten kan vi også redusere perfekt matching til $HamCycle$, og det betyr at $Match$ kan umulig være vanskeligere enn $HamCycle$, og dermed ikke vanskeligere enn $LongPath$. Det eneste problemet er at vi ikke vet hvordan vi kan finne en løsning på $LongPath$.

📚Verifikasjon

Beslutningsproblemer er problemer vi kan verifiserer et sertifikat for et ja-svar. Hvis vi lurer på om X finnes, kan vi bruke verifikasjonsfunksjonen til å sjekke enhver mulig X for en instans. Verifikasjonsmetoden i dette tilfellet kjører i polynomisk tid slik at det er mulig å verifisere det. Hvis svaret er “nei”, så har vi ingen sertifkat for problemet, ingenting å verifisere.

En verifikasjonsalgoritme for et beslutningsproblem: Tar inn en instans og et sertifikat – som egentlig kan være hva som helst.

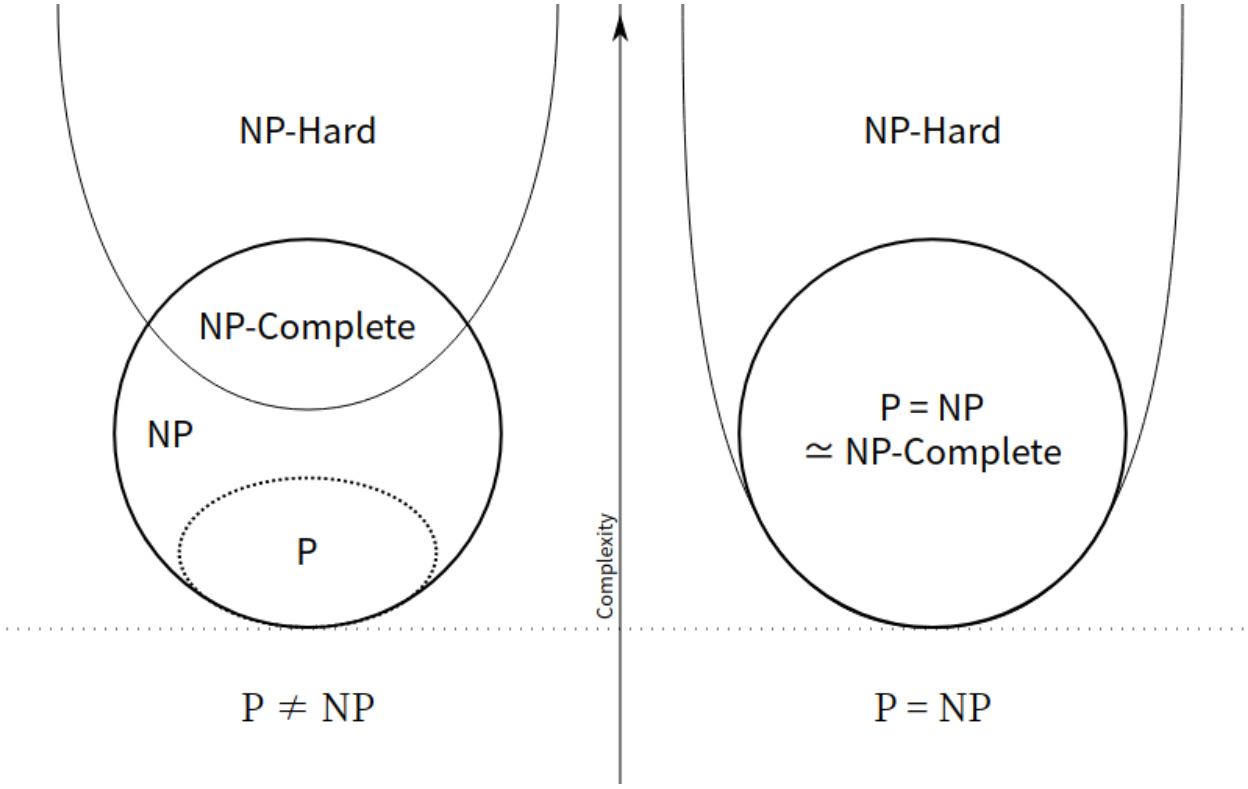
Det skal finnes et sertifikat som gir svaret «ja» hvis og bare hvis problemet har svaret «ja».

Et beslutningsproblem stiller rett og slett spørsmålet “finnes det et sertifikat for at svaret er ja?” for et slikt sertifikat skal finnes hvis, og bare hvis svaret er “ja”.

- Problemklassen **P** er problemer som kan løses i polynomisk tid.
- Problemklassen **NP** har ja-svar sertifikater som kan sjekkes i polynomisk tid. Problemer i **NP** som også er i **P** er selvfølgelig løsbare i polynomisk tid.
- Problemklassen **NPC** kan vi ikke løse, men vi kan verifisere de i polynomisk tid. NP-komplette problemer er subsettet av **NP** som alle andre **NP** problemer kan reduseres til i polynomisk tid.
- Problemklassen **NPH** kan ikke løses eller verifiseres, men de som også er i **NPC** kan selvfølgelig sjekkes i polynomisk tid. Dette er problemer som er minst like vanskelige som **NPC**, men de trenger ikke å være i **NP**, og trenger ikke å være beslutningsproblemer.
- Problemklassen **co-NP** har nei-svar som kan sjekkes i polynomisk tid.

Dette betyr at hvis man finner et bevis for å løse et NP-komplett problem, kan man løse alle andre NP problemer.

Det er også praktisk å se på beslutningsproblemene som mengder av instanser (bitstrenger) der svaret er ja. En slik mengde er et formelt språk.



What are the differences between NP, NP-Complete and NP-Hard?

I assume that you are looking for intuitive definitions, since the technical definitions require quite some time to understand. First of all, let's remember a preliminary needed concept to understand those definitions.

https://stackoverflow.com/a/19510170/14996499

📚Kompletthet

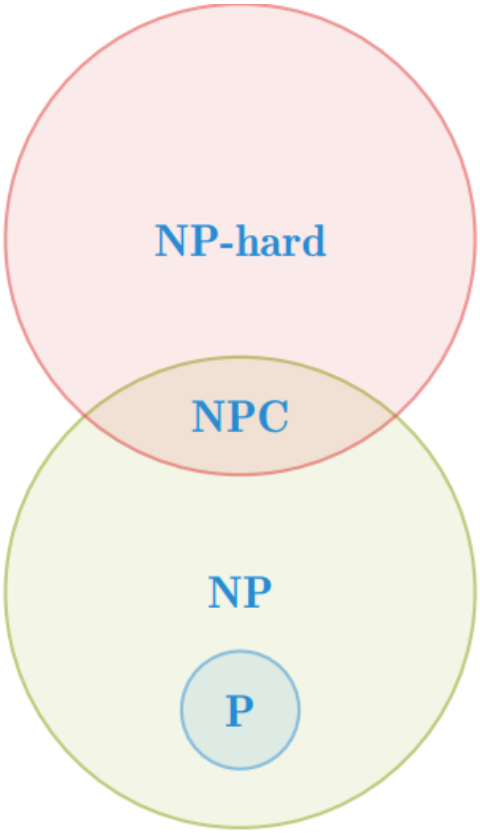
Vi har et univers av problemer og vi kan sammenligne dem. Det gir faktisk opphav til et sett med maksimalt vanskelige problemer. Disse kalles komplette for klassen NP under polynomiske reduksjoner.

Hvis vi har en mengde med kister, og en ekstra kiste K som har nøkkelen til alle de andre, så kan vi redusere åpne alle kistene til åpne K. Kompletthet er en universalnøkkel for alle problemene. Finner vi nøkkelen her, kan vi løse alle de andre problemene.

Slik ser vi for oss at hierarkiet til problemene er, selv om det ikke er helt mulig å vite. Det finnes også mange andre klasser med problemer, og det er mange

mulige scenarier. Problemet er at ingen vet hva som stemmer.

Se slides for bedre beskrivelse



Hvis det har seg slik at $P = NP$, kan vi ikke bare svare på om det finnes et sertifikat, men vi kan rekonstruere sertifikatet. Med andre ord, kan vi løse beslutningsproblemer, så kan vi løse søkeproblemer også, og finne gyldig output.

💻Oppfyllbarhet

Oppfyllbarhetsproblemet var det første problemet som ble bevist til å være **NPC**. Det går ut på å finne ut om det er mulig at en logisk krets sine inputs kan bestemme om det er mulig at output er 1.

Instans: En kombinatorisk logisk krets med én utverdi.

Spørsmål: Er det mulig å *oppfylle* eller *tilfredsstille* kretsen?

Det vil si, finnes det et sett med inn-verdier som gir utverdi 1?

- **Vil vise:**
  - CIRCUIT-SAT $\in$ **NP**
  - $(\forall \mathrm{L} \in \mathbf{NP})\ \mathrm{L} \leqslant_{\mathbf{P}} \text{CIRCUIT-SAT}$
- L har minst én verifikasjonsalgoritme A
- For en instans x, simuler A med en krets, slik at:
  - Kretsen kan tilfredsstilles hvis og bare hvis ...
  - ... det finnes en y slik at $\mathrm{A}(x, y) = 1$
- Det er akkurat tilfellene der $x \in \mathrm{L}$

L er i **NP**, så . . .

Det finnes en pol. alg. A, som er slik at

- $x \in \mathrm{L}$

nøyaktig når minst én $y \in \{0,1\}^*$ gir

- $\mathrm{A}(x, y) = 1$,

der $|y| = \mathrm{O}(|x|^c)$, for en eller annen c.