Diagnosekart for oblig3, INF3/4130 h07

Dag Sverre Seljebotn

20. november 2007

1 Oppgave 1

Med "noder" mener jeg alltid en tilstand en er i i spillet, hver node er altså en spillebrettkonfigurasjon.

1.1 Feil: Ikke lagre besøkte noder

Dersom en ikke lagrer noder i grafen som allerede er besøkt får en fort trøbbel. I et standard A*-søk er det en del informasjon en skal lagre (foreldrenode, kostnad) som en er avhengig av å lete opp igjen.

Dette slo spesielt ut på input nr. 16, som straffet ganske hardt om en ikke sjekka dette (litt vanskelig å si nøyaktig fordi det er en del forskjeller på implementasjonene, men det ligger mellom 50% og 1000% flere states, avhengig av i hvilken rekkefølge vilkårlige states blir prosessert og evt. forskjellige heuristikker og så videre).

Måten å løse dette konseptuelt sett er å skille mellom *node* og *nøkkel*. En nøkkel består av brettkonfigurasjonen og bare denne (men posisjonen til 0 kan godt lagres i denne, separat, den er jo bare en del av konfigurasjonen). En node inneholder i tillegg ting som kostnader og foreldrepekere.

Når en så flytter seg i grafen (fra en kjent node u til en nabonode v) så tar en nøkkelen til u, modifiserer den ved å gjøre flyttet, og slår så opp i en HashMap over allerede kjente noder for å finne v. Dersom v ikke finnes i HashMap'en så vet vi at v ikke er besøkt enda, så vi genererer en ny node og setter den inn.

I praksis er det lettere å vise dette enn å forklare alt, så jeg har lagt ut min egen løsning i Java der jeg har fjernet selve algoritmen men der alle datastrukturene ligger. Legg merke til bruken av `hashCode` og `equals`.

1.2 Om vurdering av kjøretid

For å vurdere hvor bra en har gjort det med løsningen er det to ting å ta hensyn til: Hvor lang tid den tar, og hvor mange states en ser på. Førstnevnte har mest med effektiviten på implementasjonen, mens sistnevnte har med algoritmen (og heuristikken) å gjøre. Spesielt dersom en gjorde feil 1.1 kunne en straffes tungt på antallet states en måtte besøke.

Dersom noe tar mye tid er det derfor lurt å sammenligne antall states, for å se om problemet er at en besøker veldig mange states eller om det er prosesseringa per state som tar tid.

For en slags kjapp benchmark så regner de beste løsningene implementert i Java ut samtlige eksempelbrett på godt under ett sekund hver (på min 64-bit AMD 3800+-prosessor).

Jeg legger ut mine egne outputfiler så en kan se "typisk" antallet states på forskjellige oppgaver, dersom en er interessert (jeg gjorde ikke noe smart men bare implementerte rett fram). Jeg har desverre litt andre navn, men de med bare tall svarer til tilsvarende 3×3-brett og de med "b" først er 4×4-bretta.

2 Oppgave 2

Les bare fotnotene dersom du sitter helt fast.

2.1 Om *Formelle språk tilsvarer vanlige språk som Engelsk osv.*

Ikke les for mye inn i dette. Det er bare et har-du-lest-pensum-spørsmål og kan besvares med "nei" og en liten definisjon på hva formelle språk er.

2.2 Om *Desisjonsproblemer er like komplekse som optimaliseringsproblemer*

Her holdt det å svare litt uformelt at "ja", utifra noe Dino nevnte i en bisetning på første forelesning. Poenget her er at det er dette som er grunnen til at en holder på med språk etc.: Et språk modellerer desisjonsproblemer, men vi er jo i virkeligheten stort sett interessert i optimaliseringsproblemer, så det at de er like komplekse er grunnen til at en er så interessert i å studere desisjonsproblemene.

Detaljene på hvorfor det er sånn kommer/kom på gruppetimen 20. november og i oppgavene til denne. Står også f.eks. på side 214 i kompendiet.

Det går ut på at en, gitt et desisjonsproblem ("finnes det en TSP-tur med K i kostnad") kan gjøre et binærsøk (som blir polynomisk i forhold til input) for å finne den optimale løsningen.

Det føres imidlertid ikke generelle beviser for dette, fordi optimaliseringsproblemer er en uformell idé som hører til i den praktiske forståelsen. Modellering av optimalisering som desisjon er selve linken mellom praksis og teori, og en kan dermed ikke teoretisk vise at den er riktig.

(Det vil si, vi mangler en teoretisk presisering av hva et optimaliseringsproblem er og kommer heller ikke inn på dette – det er derfor en "tese" og ikke et "teorem" at dette stemmer.).

2.3 Om *Problemer som ikke kan løses i polynomisk tid kalles "NP-komplette problemer"*

Nei, dette er helt galt. NP-komplette kan antagelig ikke løses på (deterministisk) polynomisk tid (men ingen har enda funnet et bevis for det og det er et av de store ukjente problemene). Men det er også mange andre klasser som ikke kan det, og der det også er bevis at de ikke kan løses på polynomisk tid.

Grunnen til at NP-komplette problemer får så stort fokus er ikke fordi de er unike i kjøretid på noen måte, men fordi det er en klasse av problemer som en i praksis er veldig interesserte i (de har mange anvendelser).

2.4 Om $L_2 = \{M | M$ **gjenkjenner** $L_1\}$

Her burde det antagelig stått "avgjør" istedetfor "gjenkjenner". Spørsmålet var altså om M vil svare Y hvis og bare hvis input er i L_1 og N ellers. Denne er ikke avgjørbar, etter standard reduksjon fra halting problem. Se beviset for $L_\$$ for mer info. [1]

2.5 Om $L_3 = \{M | M$ **gjenkjenner** $L_2\}$

Denne er litt morsom. Jeg har ikke lyst til å gi for mange hint her fordi den er så lur. Men det er i alle fall fullstendig galt å forsøke å redusere fra halting. [2]

---

[1] Den er trivielt avgjørbar. Om du fortsatt ikke er med så les eksemplene med π på side 70.

[2] Programmet som skal sendes til M_2 internt i reduksjonen vil se slik ut: 1) Simuler M på x, 2) Avgjør om det står 1 på input-tapen.

2.6 Feil: Ikke vise at 2-HAM er i NP

Å bare redusere HAM til 2-HAM er ikke nok. En slik reduksjon viser at 2-HAM er minst like kompleks som HAM, men en har ikke vist at 2-HAM er NP-komplett – kanskje er 2-HAM EXP-komplett eller PSPACE-komplett? En må også vise at 2-HAM ligger i NP, ved å gjøre en betraktning om at gitt en løsning kan en verifisere på polynomisk tid.

2.7 "Feil": Bruke en usammenhengende graf i reduksjonen

Det står i oppgaven at grafen skal være sammenhengende, men det er jo trivielt å fikse.

2.8 Feil: Redusere 2-HAM til HAM og så slutte noe fra dette

Det finnes en del variasjoner på følgende argument:

- På denne og denne måten kan en løse 2-HAM ved å bruke HAM.
- Siden 2-HAM må løses ved å bruke HAM, er den minst like komplisert, og dermed NP-hard.

Dette blir feil. Problemet er at trinn 1 som regel går ut på å vise én algoritme for 2-HAM. Men 2-HAM kan løses ved mange forskjellige algoritmer (ufarlig fotnote)[3]. Å vise at en av disse impliserer HAM sier ingenting om hvorvidt andre algorimer, mange hvorav aldri vil bli funnet, evt. kan klare seg uten HAM.

For å få dette argumentet til å gå gjennom må en dermed si noe om *alle* tenkelige algoritmer (og de vi ikke tenker på) for 2-HAM. Det kan vi bare gi opp. En må derfor snu på det, og redusere HAM til 2-HAM. Helt konkret:

Anta at du får i hånda et dataprogram som løser 2-HAM (merk: Som svarer Y eller N på om en graf er 2-HAM, ikke som gir selve løkkene. Konkret har vi Decide-2-HAM, ikke Search-2-HAM). Det du skal gjøre er så å gi en metode som, gitt et slikt 2-HAM-program, kan hjelpe deg med å løse et vilkårlig HAM-problem. Altså:

[3]Et problem har generelt uendelig mange korresponderende Turing-maskiner. Mer konkret har bare sorteringsproblemet fem til ti algoritmer i aktiv bruk. Det er ingen grunn til å anta at den samme underskogen av algoritmer ikke finnes for 2-HAM.

1. Du får et HAM-problem (en graf)
2. Gjør noen endringer med grafen (legg til noen noder og kanter)
3. Mat den endrete grafen til 2-HAM.
4. Svaret du får fra 2-HAM (enten Y eller N) skal så si deg noe om det opprinnelige HAM-problemet.

Siden 2-HAM da kan brukes til å løse HAM, får en da ut følgende logiske slutning:

$$\text{HAM er ikke mer kompleks enn det 2-HAM er}$$
$$\Leftrightarrow \text{2-HAM er minst like kompleks som HAM}$$

Siden vi vet at HAM er NP-Hard er dette bevis for at 2-HAM er NP-Hard.

Det som da gjenstår er å vise at 2-HAM ligger i NP, det vil si at du gitt et forslag til løsning (konkrete løkker denne gangen, ikke Y eller N) kan verifisere om løsningen er korrekt på polynomisk tid (på en vanlig deterministisk maskin).