Diagnosekart for oblig 2, INF3/4130 h07

Dag Sverre Seljebotn

1. november 2007

Dette er et dokument jeg har skrivd for å gjøre det enklere å gi tilbakemelding på obligene, siden så mange ting går igjen – slå opp på numrene du får tilsendt og se kommentaren din. Som fasit er det noe mangelfullt siden jeg ikke ønsker å avsløre alt for de som skal levere for andre gang.

Når jeg skriver kan noe av det kanskje oppfattes som om det har en litt sint tone – dette er ikke hensikten og skyldes bare at jeg blir ivrig. Jeg beklager at jeg ikke har tid til å gå over teksten og luke ut sånt noe.

1 Oppgave 1

1.1 Innsettingsalgoritme

Dette er ikke et trivielt punkt, det er viktig at en får med seg det som står under.

Algoritmen som skulle brukes var altså "splay det som ville blitt forelder og sett inn i rot". Det var ikke for å være vrang, men fordi vi tenkte det var greit at det stod helt konkret hva som skulle gjøres (og at dette var det samme som blei lært bort på forelesning). Jeg har nå også godtatt løsninger som bruker alternativ innsettingsalgoritme (å sette inn ny node og så splaye ny node etterpå).

Imidlertid, *å ta ting for gitt kan fort lede galt av sted*. Problemet er at i det en gjør den minste endring i en algoritme som baserer seg på amortisert analyse så kan det godt hende at ting fungerer greit, men *en har ikke lenger garantier for kjøretid*. Med andre ord må en ha det fra en kilde (eller bevise det selv) at den alternative innsettings-metoden gir like god kjøretid før en gir seg i kast med det (ellers kunne en jo bare lagd et vanlig, ubalansert binærtre, det er jo veldig greit).

(I praksis er det kanskje sannsynlig at innsettingsmetodene er like gode asymptotisk og at det bare skiller et konstantledd – problemet er om en

setter i gang å kode ting sånn en *tror* det er før en har sjekket om det fins et sånt bevis og er klar over problemstillingen).

1.2 Ikke oppdatere oldeforelder

En typisk dobbeltrotasjon innebærer endringer i X, P og G (der X er noden som splayes, P er foreldrenoden og G er besteforeldrenoden). Men i tillegg vil det skje en endring i oldeforelder, på den måten at G blir byttet ut med X i treet.

Om en bruker en eksplisitt stakk er det greit nok å bare endre oldeforelder også. Om en gjør det uten eksplisitt stakk er en måte å gjøre det på å returnere "ny barnepeker" fra den rekursive metoden, slik: Om en er på noden p og kaller $splay(p.left)$ så vil denne returnere det nye venstrebarnet til p (som ofte vil være det samme, men forskjellig om det har skjedd dobbeltrotasjon).

En alternativ metode som unngår hele problemet er å bytte om nodeverdiene mellom X og G som en del av rotasjonen, dermed vil pekere som før pekte på G nå peke på samme-objekt-som-nå-er-X.

1.3 Strategi for når en utfører dobbeltrotasjon

Det finnes mange forskjellige strategier for når en skal utføre rotasjonene, og de fleste blir litt hårete på en eller annen måte uten at det er lett å kåre en klar vinner.

Likevel vil jeg her skissere én mulig løsning i tilfelle noen trenger den som har den fordelen at den er konseptuelt rimelig enkel å forholde seg til (om ikke nødvendigvis raskest runtime?). Den går slik:

- Vi lar løvnoden som skal splayes være i fred når vi finner den først, bare noterer oss at det er den node som skal splayes (setter en global variabel el.l. til denne) og trekker oss tilbake. [1]

- Når vi kommer tilbake til noe som kan være en besteforelder så sjekker vi alle fire tilfellene, og gjør tilsvarende rotasjon (det er også mulig at vi ikke finner splay-noden på noen av de fire mulige plassene og da er vi forelder og bare fortsetter å trekke oss tilbake. Så det blir

[1] En kan diskutere globale variabler lenge, men jeg vil si at det ikke er et problem. Grunnen er at en kan alltid kapsle inn den rekursive metoden inn i en eget Splayer-klasse og se på den som en prosess istedetfor en metode – og siden det er en oblig trenger en ikke engang gjøre det. Globale variabler er derimot et problem når de fører til programdesign som ikke lar seg kapsle inn.

fem tilfeller.). Ifra besteforelderen har vi full tilgang til alle involverte noder (men ikke til oldeforelder, så dette må kombineres med 1.2).

- Roten blir et spesialtilfelle og må da kodes for seg (siden det er eneste plass enkeltrotasjoner kan skje).

2 Oppgave 2

2.1 Bruk av grådig algoritme

Noen har levert algoritmer som bare legger til nye flytforbedringsveier i grafen til det er stopp og så terminerer. Spesifikt skjer det ikke at en endrer på allerede eksisterende veier, men bare legger til nye – dette er en grådig algoritme, som ikke vil gi optimalt svar (en trenger da egentlig ikke N_f, en kan bare modifisere N destruktivt...).

På en måte er dette en alvorlig feil, siden en da mister det geniale hovedpoenget med FordFulkerson-algoritmen og som gjør at den ikke "er noe en kunnet komme på selv" (vel, raskt). På den annen side er det en liten feil på den måten at i disse tilfellene skal gjerne bare to-tre kodelinjer endres for å løse det, siden nesten all kode deles mellom en grådig algoritme og FordFulkerson.

Problemet er altså at en ikke legger inn "virtuelle" bakover-kanter i N_f som representerer flyt-reduksjon. Les i boka, om det ikke blir klart av det så kanskje notatet til Bedeho Mender (på gruppelærerbloggen) hjelper, og forsøk på nytt. Ha i mente at ikke noe særlig skal endres i programmet – så forstå algoritmen først, og så er det lett å gjøre endringene. *Skriv gjerne ut kildekoden på papir for å motstå fristelsen til eksperimentering!*, for dette er (antagelig?) noe det tar lenger tid å eksperimentere seg til enn å forstå.

Bruk f.eks. dette testcaset som ikke fungerer med grådig:

```
8
0 1 0 0 1 0 0 0
0 0 1 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 1
0 0 0 1 0 1 0 0
0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0
```

2.2 Representasjon av forbedringsveier

Her følger et forslag til hvordan en greit kan representere optimal vei. Dette er et veldig uviktig punkt som du kan hoppe over med god samvittighet.

- La `bfs` være bredde-først-metoden. Et bredde-først-søk legger et tre oppå grafen. La `bfs` returnere et foreldre-array som for hver node spesifierer hva som er foreldren til noden i bredde-først-treet, altså returnerer `bfs` treet og *ikke* forbedringsveien direkte.
- Metoden som kaller `bfs` kan da, avhengig av om en forbedringsvei ble funnet (forbedring > 0 eller forelder til målnode satt i foreldretre-arrayet) eller ikke enten løpe gjennom dette arrayet (fra sluttnoden og bakover langs foreldrepekere) for å oppdatere grafen, eller bruke dette arrayet til å finne cut.

Om en virkelig vil gå inn for pen kode (noe som *ikke* er poenget i dette kurset, det er minst like bra uten) kan en så lage iteratorer på dette, for å få noe sånt som

```
BfsResult r = bfs(N, f, 0, m-1);
if (r.increase() > 0) {
  for (Edge e : r.iterateEdgesToRoot(m-1)) {
    // oppdater f utifra e.u og e.v
  }
} else {
  // kuttet kan nå itereres over ved r.iterateVisited()
}
```

Dermed slipper en å kopiere strukturen over til lister eller lignende bare for å få finere kode. Sjekk `java.util.Iterator` for mer informasjon om å skrive iteratorer. Et alternativ som kan være vel så greit er å bruke "reverse iterator pattern", det vil si noe sånt som dette (det blir litt færre klasser å skrive):

```
int[] parent = new int[m];
int cost = bfs(N, f, 0, m-1, parent);
if (cost > 0) {
  iterateEdgesToRoot(parent, m-1, new EdgeCallback() {
     public void visit(int u, int v) {
        // oppdater f utifra u og v
     }
```

```
  });
  // men en kan jo bruke en vanlig løkke også da...
} else {
  // her vil en vanlig for-løkke være mest naturlig
}
```

2.3 Lagring og representasjon av grafene

Det var to alternativer. I begge tilfeller lagres N for seg.

- Lagre f eksplisitt underveis og gjøre oppdateringer i denne. N_f kan da enten beregnes eksplisitt eller enkelt og greit kalles som en funksjon av N, f, i, j. På slutten kan en bare skrive ut f som den er.
- Lagre N_f og oppdatere denne underveis. f må da på slutten beregnes med utgangspunkt i N_f

Det er ikke store forskjeller på de – den første følger mer rett fram fra algoritmen, mens den andre gir bittelitt mindre kode og antagelig et par aritmetiske operasjoner mindre per iterasjon (altså nesten ingenting); hovedsaken er vel at den andre sparer en array på n^2 i lagringsplass.

En ting som er viktig å være klar over er at det for hver N_f kan være flere f, så koblingen er ikke entydig, og en har derfor flere valg for hvilken f en vil velge. Det som som regel er naturlig er å sørge for at en av retningene mellom to noder har flyt 0, dette er alltid mulig og er en måte å determinere f på sånn at f er entydig.

I tillegg er det mulig å lagre grafene som kantlister istedet for matriser, men det er nok ikke å anbefale i dette tilfellet når grafen så tydelig er spesifisert i matrise-form i input og output (en må konvertere fram og tilbake og det blir bare mye ekstra kode). Om en skal bruke kantliste eller matriser avhenger vel ellers mest av fyllingsgraden i grafen – dersom antallet kanter er nær $|V|$ bruker en kantliste, om det er nær $|V|^2$ er matrise greit.

2.4 Har tatt med sluknoden i kildesiden av kuttet

Sluknoden skal ikke være med i kildesiden av kuttet. Om du får denne feilkoden fra meg er det imidlertid bare et symptom på en feil et annet sted – programmet ditt er sånn at dette *kan* skje, men det rette programmet skal ikke kreve ekstra steg for å unngå dette (det vil bare ikke skje).

Denne feilen var litt interessant fordi den fikk testprogrammet mitt til å feile – å spesifisere et kutt som inneholdt sluknoden gikk altså rett gjen-

nom `oppgave2-validate.py` uten at flyten var optimal. Beklager dette. Interesserte kan patche programmet slik:

```
c = Checker() # denne var her fra før
c.check("cut cannot contain sink node", not m - 1 in cut) # ny
```

3 Oppgave 3

3.1 Viser ikke nok detaljer i a)

Dette er bittelitt komplisert å forklare men jeg skal gjøre et slags forsøk, så får folk heller spørre.

La p være en tilstand og r en tilstand en kan komme til fra p. Det er nemlig bare første steg å komme seg til at en skal konstruere veien

$$v = \text{kanten fra siste node i } p \text{ til siste node i } r + \text{korteste vei fra } \ell(r) \text{ til } s$$

For, en kan ikke bruke lengden av denne som en øvre grense for $h(p)$ uten å vise at v faktisk ikke er innom det indre i p. Etter min (subjektive) mening er dette tilstrekkelig lite selvfølgelig til at en bør vise det.

3.2 Glemme å vise at $h(p) = 0$ for alle måltilstander p

Jeg bærer mye av skylda for denne gjengangerfeilen siden jeg selv glemte å ta med denne betingelsen i notatet mitt. Det er veldig viktig å slå opp 23.3.1 – det er denne og kun denne som danner grunnlaget for å vise monotonitet.

Uten denne betingelsen er det ikke sikkert at h er et underestimat, og da kan den heller ikke brukes i A*-søk, så dette er en betingelse vi er veldig interesserte i at skal holde.

Dette må altså vises for alle heuristikker, men er forholdsvis trivielt for a) og b). For c) er det imidlertid dette som feller hele heuristikken, for c) er ikke monoton!

I praksis gjør en det sånn i bevisene:

> La p være en vilkårlig tilstand.
>
> Dersom p er en måltilstand vet vi at ... så $h(p) = 0$ og betingelsen for monotonitet for måltilstander er oppfylt.
>
> Dersom p ikke er en måltilstand vet vi at .... La r være en tilstand vi kan komme til fra p. Da vil ... så $h(p) = \cdots \leq \cdots = h(r) + c(p, r)$.

3.3 Dele opp i unødig mange tilfeller i b)

Dette er ikke en feil, jeg vil bare komme med et tips. Det er en del som har delt opp i flere tilfeller også i b), hvor en har likhet for den ene og strengt mindre-enn i den andre. Disse tilfellene kan slåes sammen ved å gjøre en mindre-eller-lik-betraktning, konkret kan beviset skrives ned veldig ryddig på denne måten:

$$h(p) = n(p) \cdot q(p) \leq \cdots \leq \cdots = h(r) + c(p, r)$$

altså konkret med to $\leq$-betraktninger etter hverandre. (Får vel kanskje komme tilbake med detaljer her når alle har levert endelig.)

3.4 Bruk av induksjonsrammeverket

For å vise monotonitet må en vise at noe holder for alle noder p. Dette kan kanskje lede tankene til induksjonsbevis, men det er ikke riktig. (En kan godt forsøke å føre beviset innenfor et induksjons-rammeverk, men resultatet blir ikke et induksjonsbevis.)

Grunnen er at når en skal sjekke om 23.3.1 er oppfylt for en konkret tilstand så trenger en ikke bruke at 23.3.1 er oppfylt for andre tilstander. En viser altså at 23.3.1 holder lokalt for alle tilstander hver for seg $\Rightarrow$ 23.3.1 holder generelt; en trenger ingen trappetrinnseffekt der en først viser for noen og så viser for andre forutsatt de første.

Det som skjer i stedet er en mye brukt teknikk som nok er mer brukt enn induksjon; vi sier heller

> La p være en tilstand. Da ..... (gjerne der en deler opp i tilfeller og deler opp og gjør forskjellige forutsetninger på p som til sammen dekker alle tilfeller), så 23.3.1 er oppfylt for p. Siden p var en vilkårlig tilstand har en da vist det for alle tilstander, så 23.3.1 holder generelt.

3.5 Vise at h er et underestimat men ikke at den er monoton

Å vise at $h(p)$ er mindre enn den faktiske kostnaden det har å komme seg til en måltilstand, altså at h underestimerer, er ikke det samme som å vise at heuristikken er monton. En har

$$h \text{ er monoton} \quad \Rightarrow \quad h \text{ underestimerer}$$

men *ikke* den motsatte implikasjonen. Det som skal vises er et lokalt underestimat, og en betingelse for målnoden. Se 23.3.1 i boka.