UNIVERSITETET I OSLO

Det matematisk-naturvitenskapelige fakultet

Eksamen i:	**INF 4130: *Algoritmer: Design og effektivitet***
Eksamensdag:	**12. desember 2008**
Tid for eksamen:	**Kl. 09:00 – 12:00 (3 timer)**
Oppgavesettet er på:	**XX sider**
Vedlegg:	**Ingen**
Tillatte hjelpemidler:	**Alle trykte og skrevne**

Kontrollér at oppgavesettet er komplett før du begynner å besvare spørsmålene.

Les oppgavene nøye, og lykke til!

Med svar-forslag

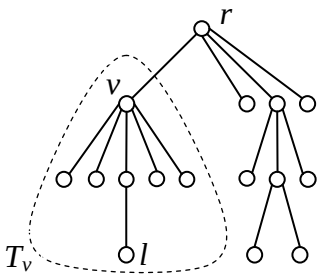
Oppgave 1 Dynamisk programmering (*21 %*)

En *uavhengig mengde* av noder i en graf er en delmengde av noder det ikke går noen kanter mellom. To nabonoder i grafen kan altså ikke begge være med i en slik uavhengig mengde.

Figur 1. De svarte nodene i treet til venstre utgjør en uavhengig mengde – det går ingen kanter mellom noe par av svarte noder. Dette er faktisk også en største uavhengig mengde – det finnes ingen uavhengig mengde større enn elleve noder i det venstre treet. De svarte nodene i treet til høyre utgjør ingen uavhengig mengde – det går kanter mellom to og to av dem.

Å finne den største uavhengige mengde av noder i en generell graf er et NP-hardt problem, i denne oppgaven skal vi derfor kun se på trær. I trær kan vi nemlig bruke dynamisk programmering til å finne størrelsen av største uavhengige mengde (og også den faktiske mengden, om vi ønsker). [Sidebemerkning: Å *kun* se på trær er ikke en *så* stor forenkling som det kanskje kan høres ut. Det finnes nemlig en mye større klasse av tre-aktige grafer hvor den samme metoden kan brukes. Vanlige trær er spesialtilfeller av disse grafene.]

La T være treet vårt og r være roten. For hver node v i treet, la T_v være sub-treet med v som rot (se Figur 2). Husk at det for enhver node v vil være to muligheter når vi forsøker å danne en uavhengig mengde: enten å ta v med i mengden eller å la det være.



Figur 2. Et tre T med rot r, en løvnode l, en intern node v, og subtreet T_v med rot i v.

Spørsmål 1.a (7 %)
Om vi skal beregne størrelsen av største uavhengige mengde av noder i T, *bottom-up*, må vi opplagt starte med løvnodene. Vi tenker oss at de beregnede verdiene lagres i nodene i treet, slik at vi slipper å bruke en tabell. For en løvnode l, og sub-treet T_l, skal vi beregne to verdier:

$l.m$	størrelsen av den største uavhengige mengde i T_l som inneholder noden l,
$l.u$	størrelsen av den største uavhengige mengde i T_l som *ikke* inneholder noden l.

Hva blir disse verdiene for en løvnode l? (Merk at vi *kun* ser på sub-treet T_l.)

Spørsmål 1.b (7 %)
Når løvnodene er ferdige, kan disses foreldre gjøre sine beregninger, og så videre oppover i treet. Generelt må en node vente til alle dens barn er ferdig før den kan gjøre sine egne beregninger. De resterende nodene v, med korresponderende sub-trær T_v, skal også beregne:

$v.m$	størrelsen av den største uavhengige mengde i T_v som inneholder noden v,
$v.u$	størrelsen av den største uavhengige mengde i T_v som *ikke* inneholder noden v.

Angi med enkel pseudokode hvordan $v.m$ og $v.u$ beregnes for en node v som ikke er et løv. Du skal ikke skrive kode for traversering av treet eller liknende, kun vise hvordan v gjør sin beregning basert på verdier som nå allerede er beregnet. (Merk at vi fortsatt *kun* ser på sub-treet T_v.) Du kan anta at vs barn ligger i en passende datastruktur, og gå igjennom disse med en for-løkke `FOR i IN v.children DO{}` som aksesserer allerede beregnede verdier for hvert barn i av v.

Spørsmål 1.c (7 %)
Når løvnodene har beregnet initialverdiene, og de resterende nodene har beregnet sine verdier nedenfra og opp, er vi i prinsippet ferdig. Hvor/hvordan finner vi nå størrelsen av største uavhengige mengde i T?

Svarforslag 1.a
$l.m = 1$,
$l.u = 0$.

De uavhengige mengdene vil være $\{l\}$ og $\emptyset$, hhv.

Svarforslag 1.b
```
v.m = 1     // node v er selv med
v.u = 0     // node v er ikke med
FOR i in v.children DO
{
   v.m = v.m + i.u                  // Er v med, kan vi ikke ta med barna.
   v.u = v.u + max(i.u, i.m)        // Er v ikke med, kan vi ta med de barna
}                                   // vi ønsker, men behøver ikke ta alle.
```

Det lille vanskelige her blir vel her nå (?) å se at man ikke *må* ta med alle barn av en node *v* som ikke selv er tatt med, men at man *kan* gjøre det, og at man altså tar med/utelater på den måten som gir høyest verdi; tilsvarende er det ikke om man har tatt med node *v*, da kan man *ikke* ta med noen av barna.

Svarforslag 1.c
Svaret finner vi ved å ta max(*r.u*, *r.m*).

Oppgave 2 Strengsøk (18 %)

Vi skal bruke Karp-Rabin-algortimen for å søke etter patternet `666` i strengen `12345666`.

Vi antar at algoritmen arbeider med de vanlige tallene i titallssystemet, som beskrevet i kursboken (Berman & Paul, *Algorithms: Sequential, Parallel, and Distributed*), slik at vi slipper omregning via et annet og mer uvant tallsystem, videre antar vi at algoritmen arbeider modulo 3.

Spørsmål 2.a (12 %)
Vis hvordan algoritmen arbeider stegvis under søket, hva den aktuelle delen av strengen (vinduet) omegnes til modulo 3, og angi hvor det ikke blir match, hvor det blir uekte (spuriøs) match og hvor det blir ekte match.

Du kan gjerne gjøre dette i kort tabell-form, f.eks slik:

1	2	3	4	5	6	6	6	Omregning av vinduet modulo 3	Match?
1	2	3						123 mod 3 = 0	?
	2	3	4					?	?
⋱								⋮	⋮

Spørsmål 2.b (6 %)
Hvordan forholder det antall uekte (spuriøse) matcher du fant i 2.a seg til det forventede antall uekte (spuriøse) matcher i uniformt fordelte strenger når man arbeider modulo 3?

Svarforslag 2.a

1	2	3	4	5	6	6	6	Omregning av vinduet modulo 3	Match?
1	2	3						123/3 = 41 / 0(mod 3)	spuriøs match
	2	3	4					234/3 = 78 / 0(mod 3)	spuriøs match
		3	4	5				345/3 = 115 / 0(mod 3)	spuriøs match
			4	5	6			456/3 = 152 / 0(mod 3)	spuriøs match
				5	6	6		566/3 = 188,66 / 2(mod 3)	ikke match
					6	6	6	666/3 = 222 / 0(mod 3)	ekte match stopp

Svarforslag 2.b
Det forventede antall spuriøse matcher er *n*/3 om vi gjør *n* forsøk. Vi gjør seks forsøk, og forventer derfor 6/3=2. De fire spuriøse matchene vi fikk er altså noe i overkant av det forventede antallet.

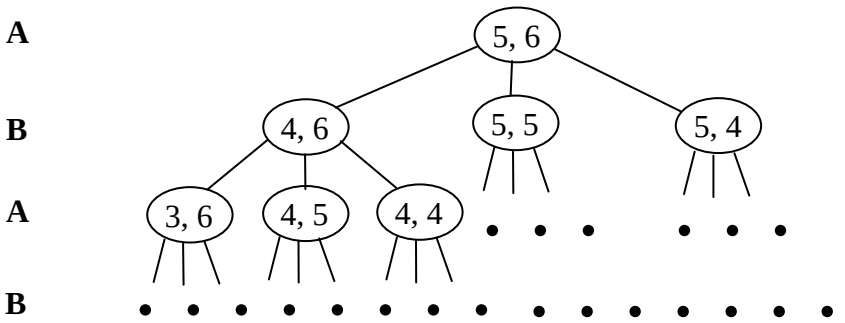
Oppgave 3 Spilltrær (20 %)

Et spill for to personer er som følger: Det er to bunker med fyrstikker *F* og *G*, med h.h.v. *f* og *g* fyrstikker. Vi skriver en gitt situasjon slik: (*f, g*), eller med hel ring rundt. Det er fra hver situasjon maksimalt tre mulige trekk:

- Fjerne en fyrstikk fra *F*,
- Fjerne en fyrstikk fra *G*,
- Fjerne to fyrstikker fra *G*.

Et trekk er bare lovlig om det nok fyrstikker i den aktuelle bunken. Spillerne er **A** og **B**, de trekker annenhver gang, **A** trekker først. Den spilleren som er i trekket når situasjonen er (0, 0) har vunnet (altså, den som tar siste fyrstikken taper).

Vi skal tegne opp spilltrær for dette spillet. Subnodene til en node skal alltid tegnes fra venstre mot høyre i den rekkefølge de tre eller færre mulige trekk er angitt over. Et tre fra startsituasjonen (5, 6) kan for eksempel begynne fra toppen slik:



Figur 3. Toppen av spilltreet med (5, 6) som startsituasjon.

Spørsmål 3.a (4 %)
Tegn opp hele spilltreet fra startsituasjonen (1, 3). Ikke gjør noe forsøk på å slå sammen like noder.

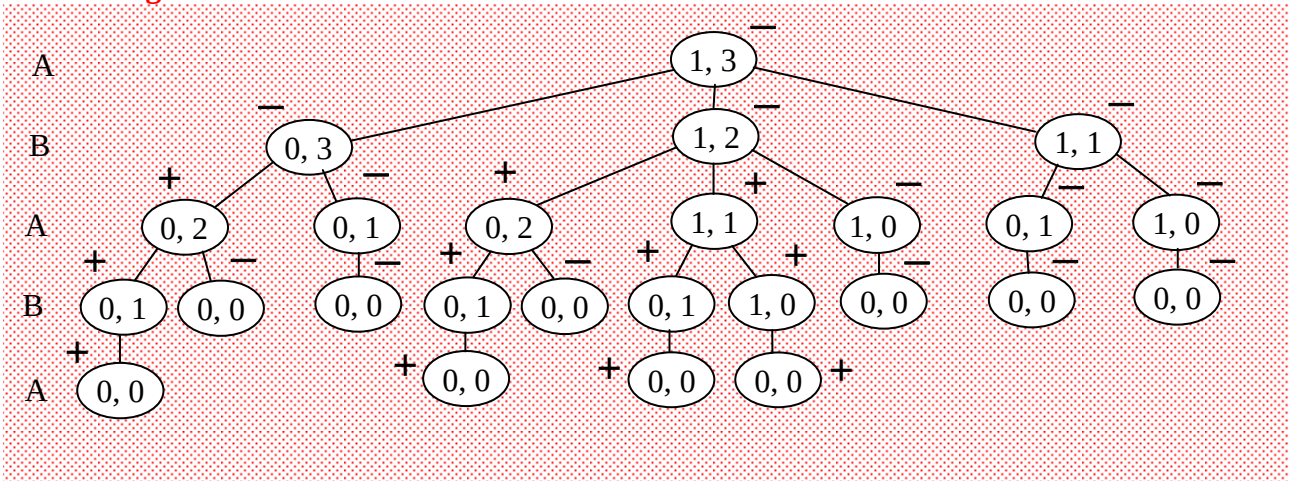
Spørsmål 3.b (4 %)
Vi vil merke vinstnoder for **A** (altså, der **A** har en vinnende strategi) med **+** og vinstnoder for **B** med **–**. Merk ut fra dette alle nodene i spilltreet fra oppgave 1.a med enten **+** eller **–**.

Spørsmål 3.c (4 %)
Finnes det en vinnende strategi for den som skal trekke fra situasjon (1, 3)?

Spørsmål 3.d (4 %)
Finnes det en vinnende strategi for den som skal trekket fra situasjon (1, 2)? Forsøk å bruke informasjon du allerede har samlet.

Spørsmål 3.e (Uavhengig av 1.a – 1.d) (4 %)
Vi vil bruke alfa-beta-avskjæring ved gjennomgang av et spilltre, men vi er så uheldige at vi alltid fra en situasjon først ser på et dårligst mulig trekk for den som skal trekke. Kari påstår da at det umulig kan bli noen alfa-beta-avskjæring i det hele tatt? Ola mener at dette er feil. Hvem har rett? Forklar!

Svarforslag 3.a:



Svarforslag 3.b
Merkene er angitt på treet over.

Svarforslag 3.c
Nei, for rotnoden er merket med –.

Svarforslag 3.d
Ja. Her er det bare å se på det midterste subtreet i treet fra 1.a. Roten av det er riktignok merket med –, men man må kompensere for at det her er **B** som er i trekket, og for den finnes en vinnende strategi.

Svarforslag 3.e
Ola har rett. For selv om man ser på det dårligste trekket først, så kan det neste man ser på være det beste trekket fra denne situasjonen, og det kan gjøre at man får avskjæring på neste nivå under senere noder.

Oppgave 4 Uavgjørbarhet (21 %)

En instans av DET BEGRENSEDE UNIVERSELLE STOPPEPROBLEM [BUS] er et program (= en algoritme) *A* og en konstant *k*. Det tilhørende spørsmålet er om *A* stopper for alle input-strenger etter *k* eller færre skritt.

Spørsmål 4.a (7 %)
Bestem om BUS-problemet er avgjørbart eller ikke. Forklar.

Spørsmål 4.b (7 %)
Bestem om følgende problem *Q* er avgjørbart eller ikke. Forklar.

Instans: Et program B.
Spørsmål: Vil B avgjøre BUS-problemet?

Spørsmål 4.c (7 %)
Bestem om følgende problem *R* er avgjørbart eller ikke. Forklar

Instans: Et program C
Spørsmål: Vil C avgjøre Q.

Svarforslag 4.a
BUS-problemet er avgjørbart. Det kan avgjøres ved en algoritme som lager alle mulige inputstrenger av lengde *k* eller mindre og for hver av dem starter *A* og utfører *k* skritt. (...)

Svarforslag 4.b
Q er uavgjørbart. Vi velger å redusere fra en instans S av det forenklede stoppeproblemet (det der S hverkan leser input eller gir output, og som er uavgjørbart). Transformasjonen av S til en instans T av Q går slik:

T lages ved i S å sette inn hele algoritmen som løser BUS-problemet (det fra 4.a) umiddelbart før alle steder det er en stopp-instruksjon S. Vi ser da at T vil løse BUS-problemet hvis og bare hvis S stopper. Om S ikke stopper kommer T aldri til algoritmen som avgjør BUS-problemet, og om S stopper vil algoritmen som avgjør BUS-problemet settes i gang. Merk at programmet S, eller en del av S, ikke kan løse BUS-problemet siden det ikke gir output.

Svarforslag 4.c
R er avgjørbart, siden det er klart at ingen programmer vil avgjøre *Q*. *Q* er jo uavgjørbar, og samme hvilket program *C* man kommer opp med er det derfor helt sikkert at det ikke kan avgjøre *Q*. Ingen programmer kan avgjøre Q, og et program som avgjør R er derved rett og slett ett som svarer ”nei” uten en gang å se på C.

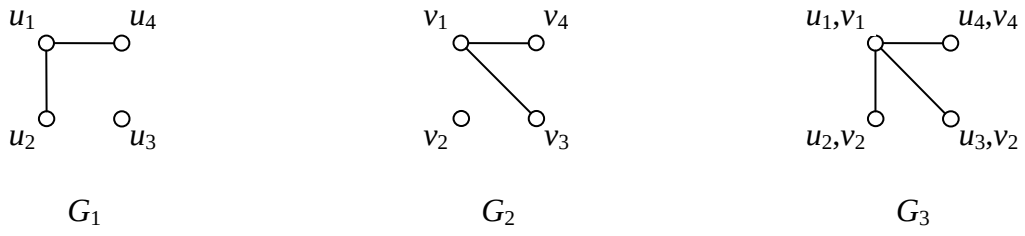
Oppgave 5 Kompleksitet (*20 %*)

Vi definerer en samposisjonering av grafer (*graph superposition*) intuitivt som grafen vi får ved å plassere to grafer med *n* noder hver oppå hverandre (slik at nodene slås sammen parvis). Formelt definerer vi *graf-samposisjonering* slik: La G_1 og G_2 være grafer på *n* noder hver. Vi sier at G_3 er en samposisjonering av G_1 og G_2 dersom:

(i) nodene til G_3 er par (v_1, v_2) slik at $v_1 \in G_1$ og $v_2 \in G_2$, og

(ii) $\{(u_1, u_2), (v_1, v_2)\}$ er en kant i G_3 dersom $\{u_1, v_1\}$ er en kant i G_1, eller $\{u_2, v_2\}$ er en kant i G_2, eller begge deler.

Vi definerer GRAF-SAMPOSISJONERINGSPROBLEMET [GSP] slik: Gitt tre grafer med *n* noder hver: G_1, G_2 og G_3 som input, avgjør hvorvidt en graf isomorf med G_3 kan lages ved å samposisjonere G_1 og G_2. (Intuitivt vil dette si at man skal avgjøre hvorvidt det er mulig å plassere G_1 og G_2 oppå hverandre slik at den resulterende grafen vil se ut som G_3.)



Figur 4. Samposisjonering G_3 av grafer G_1 og G_2.

Hva er kompleksiteten til GSP? Bevis svaret ditt.

Svarforslag 5

GSP er NP-komplett. Det kan vises slik:

For å vise at GSP er i *NP* kan vi bruke et sertifikat som består av en måte å legge $u_1,\ldots, u_n$ "oppå" $v_1, \ldots, v_n$ (det er n! slike måter, hver av dem kan angis som en permutasjon). Dessuten må vi angi en korrespondanse mellom nodene i resultatet av denne samposisjoneringen og nodene i grafen G_3. Det å sjekke dette sertifikatet består i å sjekke at grafen som fremkommer av den angitte samposisjoneringen av G_1 og G_2, er lik G_3 etter den angitte korrespondanse. Det kan opplagt gjøres i polynomisk tid.

For å bevise at GSP er *NP*-komplett viser vi en reduksjon fra HAMILTONBARHET til GSP. Gitt en instans *G* av HAMILTONBARHET (en graf med n noder) lager vi en instans av GSP ved å la G_1 være en "sirkel-graf" med n noder (og n kanter), og lar G_2 og G_3 begge være lik *G*. Vi ser da er svaret er ja på GSP-instansen hvis og bare hvis *G* er hamiltonsk. Altså er GSP *NP*-komplett.

(Slutt på oppgavesettet)