UNIVERSITETET I OSLO

Det matematisk-naturvitenskapelige fakultet

Prøveeksamen 2007, med svarforslag

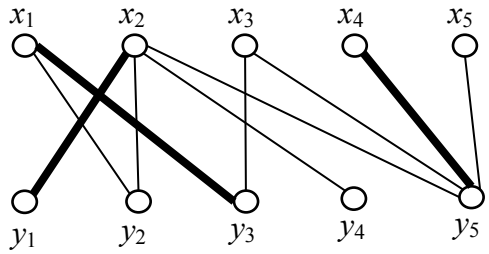
Eksamen i:	**INF 3130/4130: *Algoritmer: Design og effektivitet***
Eksamensdag:	**Fredag 15. desember 2006**
Tid for eksamen:	**Kl. 09.00 til 12.00**
Oppgavesettet er på:	**4 sider**
Vedlegg:	**Ingen**
Tillatte hjelpemidler:	**Alle trykte og skrevne**

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

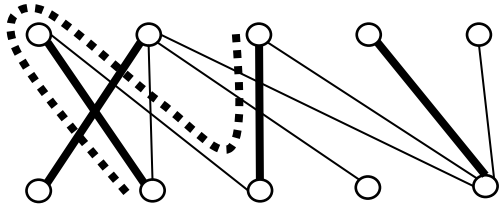
Les oppgavene nøye, og lykke til!

Oppgave 1.

Spørsmål 1a (7%)
Under er gitt en bipartit graf, med en angitt (ikke-perfekt) matching. Finn (gjerne på en intuitiv måte) en forbedringsvei i forhold til denne matchingen. Angi denne forbedringsveien og tegn grafen med den matching du får om du "bruker" denne forbedringsveien.

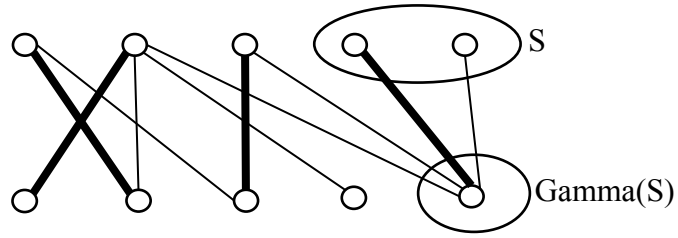


Svarforslag 1a
En (og den eneste) forbedringsveien er angitt med en stiplet linje under, og resultatet av å bruke denne er angitt i selve grafen.

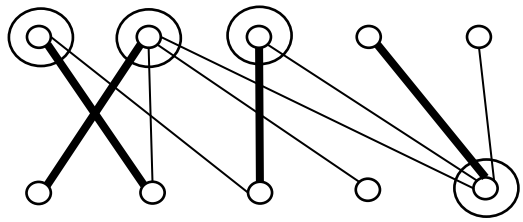


Spørsmål 1b (7%)
Påvis at den underliggende grafen fra 1a ikke har noen større matching enn den du fant i 1a.

Svarforslag 1b
En mulighet: En større matching ville være en perfekt matching. Etter Halls Teorem kan man *ikke* finne en perfekt matching hvis (og bare hvis) det finnes en mengde S blant x-nodene (eller y-nodene) slik at
|S| > |Gamma(S)|. En slik S er merket av i grafen under:

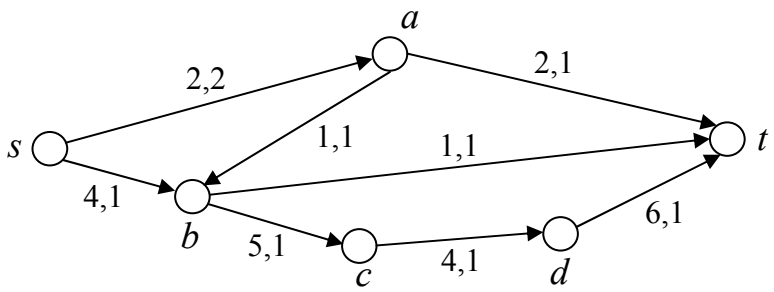


En annen mulighet er å peke ut et utplukk av noder som dekker alle kanter, og som har fire noder. Da kan ingen matching ha flere kanter enn dette. Et slikt er angitt under:

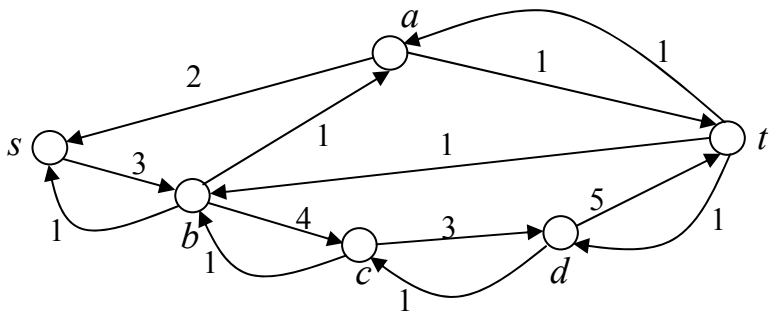


En tredje måte er å kjøre den "Ungarske algoritmen" til den stopper, og vise at den ikke finner noen forbedringsvei. Det blir jo også svært enkelt. Treet fra x5 blir f.eks. bare veien x5, y5, x4.

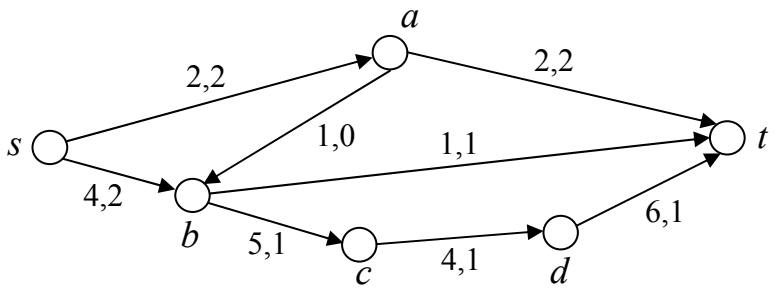
Spørsmål 1c (7%)
Under er gitt et flytnettverk, der målet er å maksimere flyten fra *s* til *t*. På hver kant er angitt en kapasitet og en flyt (i den rekkefølgen). Vi tenker oss at vi bruker FordFulkerson-algoritmen, med reglen angitt av Edmonds og Karp for å velge neste steg. Hva blir da det neste steget algoritmen gjør?



Svarforslag 1c
Grafen N_f (mulige flytforandringer på hver kant) blir slik:



Reglen til Edmonds og Karp sier at man her skal velge den korteste forbedringsveien fra s til t, og la denne få så mye flyt som mulig. Den korteste veien er s, b, a, t, og maksimal flyt på denne veien er 1. Den resulterende flyt blir som følger (men det var ikke krevd at de skulle tegne opp denne):



Oppgave 2

Hvilke av språkene angitt under er uavgjørbare? Begrunn svaret.

Spørsmål 2a (7%)
L_1 = {M | M stopper for enhver input}

Svarforslag 2a
Uavgjørbart. Bevises ved standard reduksjon. (Den enkle utfordringen er å kjenne at standard-reduksjonen virker). Ellers er en mer utvidet bevis som følger:

Anta mot en selvmotsigelse at L1 er avgjørbart. Da må det finnes en Turingmaskin (eller en algoritme) M_1 som avgjør L1. Vi kan da bruke M_1 for å løse Stoppeproblemet som følger.

Vi lager en Turingmaskin (algoritme) R (for 'reduksjon') som tar en instans av Stoppeproblemet (M, x) som input og lager en instans av L1, nemlig M', som output. Maskinen M' er som følger:

M': SIMULER MASKIN M VED INPUT x

Det er ikke vanskelig å verifisere at R en reduction:

(i) Vi viser først at M' stopper for enhver input hvis og bare hvis M stopper ved input x. For å se dette, merk at M' ser ikke på sin input. Hvis M stopper ved input x, så vil også simuleringen M' stoppe. Hvis M derimot ikke stopper ved x, så vil M' ikke stoppe uanset input. Dermed blir det å

svare om M' stopper for enhver input ekvivalent med det å løse stoppeproblemet.

(ii) Vi viser så at R kan beregnes av en Turingmaskin (er en algoritme). For å se dette, merk at R er en standardreduksjonen. Nesten hele M' er gitt på forhånd som konstant streng i R. M' er en enkel modifikasjon av Universal Turingmaskinen som i steden for å lese M og x fra sin input kjører (simulerer) en bestemt M på en bestemt x som begge er gitt som konstanter i M's egen kode. R skal bare skrive ut M' som altså inneholder M og x fra inputen til R på bestemte steder.

Siden R er en reduksjon fra Stoppeproblemet til L1, gitt M_1, så kan vi løse Stoppeproblemet med først å kjøre R og så M_1 . Men dette er en selvmotsigelse, siden vi allerede har vist at Stoppeproblemet er uavgjørbart.

Siden antagelsen at L er avgjørbart leder til en selvmotsigelse, må den være gal. Vi kan derfor konkludere med at L er uavgjørbart.

Spørsmål 2b (7%)
L_2 = {M | M løser *Stoppeproblemet*}

Svarforslag 2b
Avgjørbart. Løses med den trivielle algoritmen som alltid svarer NEI.

Oppgave 3

Fire-dimensjonal matching (4DM) er en rett fram generalisering av *Tre-dimensjonal matching*.
(Sitat fra foilene:

3-DIMENSIONAL MATCHING (3DM)

Instance: A set M of triples (a, b, c) such that $a \in A$, $b \in B$, $c \in C$. All 3 sets have the same size q ($|A| = |B| = |C| = q$).

Question: Is there a matching in M, i.e. a subset $M' \subseteq M$ such that every element of A, B and C is part of exactly 1 triple in M'?)

Spørsmål 3a (7%)
Angi spesifikt hvorfor man kan si at 4DM (og 3DM) er i NP, ved å ta utgangspunkt i definisjonen av NP.

Svarforslag 3a
Kravet for å være i NP er at det er et ja/nei-spørsmål, og at det dersom en instans er en ja-instans så skal det finnes et "setifikat" for det, som er slik at man i polynomisk tid kan sjekke at sertifikatet virkelig viser at dette er en ja-instans. 4DM er i NP fordi det finnes et slikt sertifikat, nemlig et utplukk M' av kvadrupler fra M, slik at M' tilfredsstiller kravet. At dette kan kontrolleres i polynomisk til er opplagt.

Spørsmål 3b (8%) (Vente med denne til slutt?)
Bevis at 4DM er NP-komplett. (Hint: Ta utgangspunkt i beviset for at 3DM er NP-komplett.)

Svarforslag 3b
At 4DM er NP-komplett bevises med følgende reduksjon fra 3DM: For hver trippel (x, v, w) i 3DM instansen lag n fire-tupler (x,v, w, z_i), en for hver z_i i Z.

To påstander må bevises:

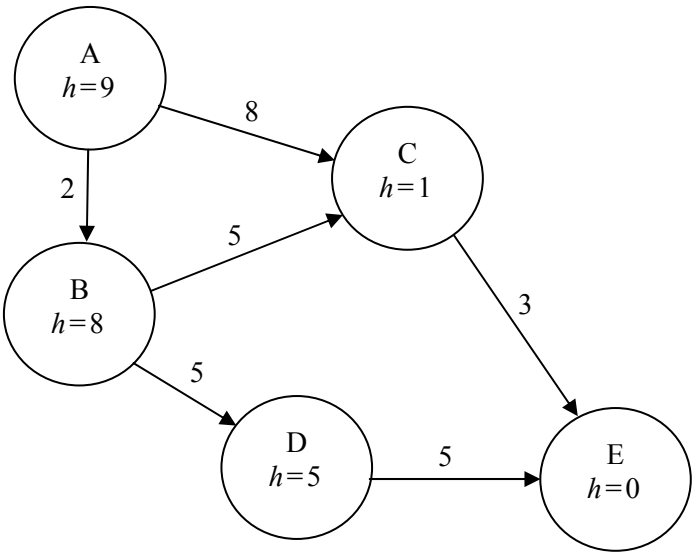
- Den nye 4DM instansen har en løsning hvis og bave hvis den opprinnelige 3DM instansen har en løsning.
- Reduksjonen kan utføres av en algoritme i polynomisk tid.

Første påstand følger fra reduksjonen: Hvis opprinnelige 3DM instansen har en løsning, så har den 4DM instans også en løsning – bare velg 4-tupler ved å sette til z_i –er i en naturlig rekkefølge, fra z_1 til z_n. Hvis 4DM instansen har en løsning, da lages tilsvarende 3DM løsningen ved å slette z_i – er.

Andre påstanden følger lett fra konstruksjonen – algoritmen går gjennom alle tripler og for hver av dem lager n fire-tupler.

Oppgave 4

Gitt følgende rettede graf, hvor vi ønsker å bruke A*-søk (altså "*A*-SearchMH(...)*" i boka) til å finne korteste vei fra A til E.



Her er alle veilengder (eller "costs" eller "weights") for enkelt-steg angitt som tall på kantene (i boka er dette *c*-funksjonen, slik at vi her altså har *c*(A, B) = 2). Dessuten er den heurstikk-funksjonen *h* som skal brukes slik at den for de forskjellige nodene blir som angitt (når E er målnode).

Spørsmål 4a (7%)
Gjennomfør et søk fra A til E med A*-algoritmen. Angi stegene som algoritmen går gjennom så nøye at du kan påvise at algoritmen faktisk ikke finner korteste vei fra A til E. Angi f- og g-verdier.
(**Merk:** Det er en trykkfeil på foilen som beskriver A*-algoritmen (med foil-overskrift: "Selve algoritmen"). Linja som starter med: "- Ta ny minste ut av PQ, …", skal erstattes med: "- Ellers, la x bli en tre-node".)

Svarforslag 4a
Algoritmen arbeider med tre nodemengder, og vi kaller disse: PQ (prioritetskøen), tre-nodene, og de usette nodene (og disse mengdene er disjunkte, og dekker alle nodene). Etter initialiseringen er startnoden A(g=0, f=9) i PQ, og resten er usette.
Steg 1: A taes ut av PQ, og blir tre-node. Dette fører til at de usette nodene B og C blir tatt in i PQ som: B(g=2, f=10) og C(g=8, f=9).

Steg 2: C taes ut av PQ og blir tre-node. Dette fører til at E går inn i PQ som E(g=11, f=11).
Steg 3: B taes ut av PQ, og blir tre-node. Dette fører til at D går inn i PQ som D(d=7, f=12).
Merk her at C ikke påvirkes siden den allerede er tre-node.
Steg 4: E taes ut av PQ, og algoritmen stopper. E sin g-verdi (som er 11) skal da angi lengden av korteste vei. Men lengden av korteste vei er opplagt 10 (langs veien A, B, C, E), så vi får galt svar.

Man kunne her også holdt greie på "tilbakeveien fra alle trenoder til startnoden" med foreldrepekere, men vi regner oppgaven som besvart uten at man gjør dette.

Spørsmål 4b (7%)
Finn grunnen til at A*-algoritmen her ikke finner korteste vei. Angi denne generelt, og eksplisitt i forhold til grafen over.

Svarforslag 4b
For at A*-algoritmen alltid skal virke må *h*-funksjonen være *monoton*, altså for enhver kant fra *M* til *N* må:
$h(M) \leq h(N) + lengde(M, N)$
$h(P) = 0$ for enhver målnode *P*
Dette er ikke oppfylt for alle kanter i grafen over, siden det for kanten fra B til C gjelder:
$h(B) = 8$ og $h(C) + lengde(B, C) = 1 + 5 = 6$
Altså er den *h*-funksjonen som er brukt ikke monoton. Merk at for alle *h*-verdiene i grafen gjelder at de er mindre enn eller lik korteste vei fra denne noden til målnoden. Men dette holder altså ikke til at algoritmen skal virke riktig.

Oppgave 5

Spørsmål 5a (7%)
Vi skal se på følgende problem: Gitt to bokstavsekvenser S og T. Vi skal avgjøre om man kan lage S ut fra T ved kun å fjerne tegn fra T. Om S = "abc" og T = "aacbbac" er dette mulig, men ikke om T = "aacbba".

Angi hvordan man kan løse dette problemet med dynamisk programmering. Angi her hva slags tabell du vil bruke, hvordan den generelle reglen for utfylling er, og hvordan du vil initialisere tabellen. Skisser også et program som gjør dette.

Svarforslag 5a
Dersom bokstavene S[i] og T[j] er like kan vi enten "bruke" denne likheten, da må S[1:i-1] og T[1,j-1] kunne gjøres like, eller vi kan fjerne T[j], og da må S[1:i] og T[1,j-1] kunne gjøres like.
Dersom S[i] og T[j] ikke er like må vi fjerne T[j], og da er vi avhengig av at S[1:i] og T[1,j-1] kunne gjøres like.

Den generelle reglen for verdien i B[i,j] blir dermed:
Dersom S[i] = T[j]: *true* dersom B[i-1, j-1] = *true* eller om B[i-1, j-1] = *true*
elles: *false*
Dersom S[i] ≠ T[j]: *true* dersom B[i, j-1] = *true*
elles: *false*

Initialsiseringen blir som følger:

B[0, j] for alle j settes til *true,* siden den tomme strengen alltid kan oppnås ved fjerninger.

B[i,0] for i>0 settes til *false*, siden S-strengen her er lenger enn T-strengen, og da er det ikke håp.

I eksempelet under (ikke spurt etter!) ser vi på S = ”abc” og T = ”aacbbcc”. Vi ser at det er mulig å gjøre strengene like.

		T						
		a	a	c	b	b	a	c
		t	t	t	t	t	t	t
	a	.	t	t	t	t	t	t
S	b	.	.	.	.	t	t	t
	c	.	.	.	.	.	.	t

Et program som gjør dette kan være:

```
for j = 0 to T.length do { B[0, j] = true; }
for i = 1 to S.length do { B[i, 0] = false; }

for i = 1 to S.length do {          // Rekkefølgen av løkkene er vilkårlig
    for j = 1 to T.length do {
        if S[i]==T[j] then {
            if B[i , j -1] == true or B[i-1, j -1] == true then { B[i, j] = true; }
                                                          else { B[i, j] = false; }
        } else {
            if B[i , j -1] == true then { B[i, j] = true; }
                                  else  { B[i, j] = false; }
        }
    }
}
return B[S.length, T.length];
```

Spørsmål 5b (8%) (Vente med denne til slutt?)
Forklar hvordan du kan forandre din algoritme fra 5a slik at den løser følgende problem: Man kan fremdeles stryke tegn i T, men får ikke lov til å stryke to tegn som står etter hverandre i (den opprinelige) T. Er det da mulig å få fram S fra T? (Hint: Forsøk å bruke to varianter av “true” i tabellen.)

Svarforslag 5b
Vi innfører to varianter av true, nemlig: B[i, j] = *tu*, som betyr at man kan få likhet mellom S[1:i] og T[1:j], *uten* at man behøver å fjerne T[j]. Ellers B[i, j] = *tm*, som betyr at man kan få likhet mellom S[1:i] og T[1:j], men da er man nødt til å fjerne T[j]. (”*tm”* er ment å stå for*”true* ***med*** *fjerning)*

Den generelle reglen for verdien i B[i,j] blir da:

Dersom S[i] = T[j]: *tu* dersom B[i-1, j-1] = *tu* eller B[i-1, j-1] = *tm*
elles: *tm* dersom B[i, j-1] = *tu*
elles: *false*

Dersom S[i] ≠ T[j]: *tm* dersom B[i, j-1] = *tu*

elles: *false*

Initialiseringen blir som følger:

B[0, 0] settes til *tu*, siden de tomme strenger er like (uten noe fjerning)

B[0, 1] settes til *tm*, siden vi kan oppnå den tomme streng ved å fjerne det ene symbolet i T[1].

B[0, j] for j>1 settes til *false,* siden vi måtte fjerne alle (og dermed flere ved siden av hverandre) om vi skulle oppnå den tomme streng

B[i,0] for i>0 settes til *false*, siden S-strengen her er lenger enn T-strengen, og da er det ikke håp.

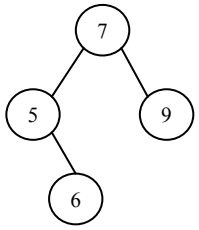
I eksempelet under (ikke spurt etter!) ser vi på S = ”abc” og T = ”aacbcc”. Vi ser at vi her kan få likhet.

		T				
	a	a	c	b	c	c
	tu	tm	.	.	.	.
a	.	tu	tu	tm	.	.
S b	.	.	.	tu	tm	.
c	.	.	.	.	tu	tu

Oppgave 6

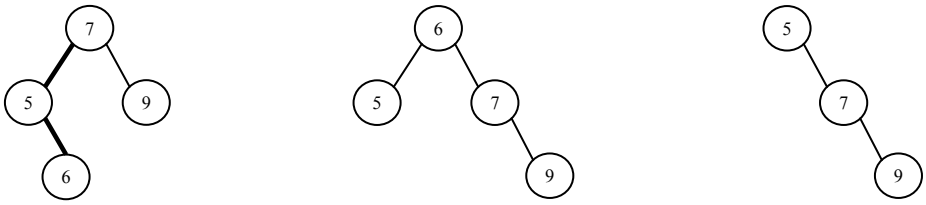
Spørsmål 6a (7%)

Vi starter med det følgende splay-treet:



Tegn treet vi ender opp med når noden med verdi 6 slettes. Angi hvilken aksess-sti som følges og hva slags rotasjon som utføres.

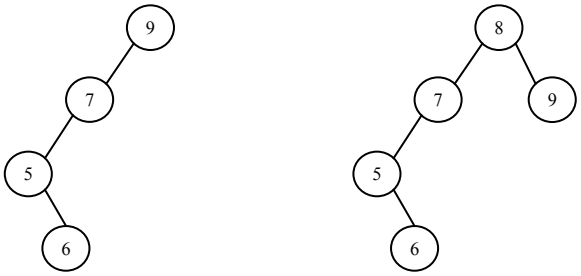
Svarforslag 6a



Node 6 aksesseres via en sikk-sakk sti. Vi gjør en dobbel rotasjon. 6 blir ny rot, 5 og 7 dens barn (9 er fortsatt barn av 7). 6 er nå rot og kan slettes. 5 er største node i venstre tre og blir ny hovedrot når trærne 5 og 7-9 spleises.

Spørsmål 6b (7%)
Vis så resultatet av å sette inn verdien 8 i det **opprinnelige** splay-treet i spørsmål 6a.

Svarforslag 6b



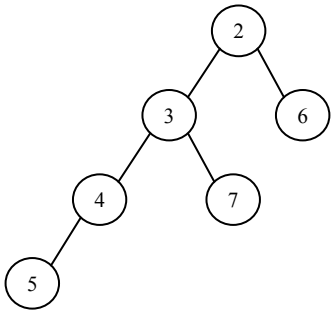
Node 9 skal få nytt barn, den splayes. 8 settes så inn mellom 7 og 9, som rot.

Spørsmål 6c (7%)

Vi har følgende to venstrevridde heaper (leftist heaps) L1 og L2:

Vis resultatet av å spleise L1 og L2 (`merge(L1,L2)`).

Svarforslag 6c



Høyrestiene 3 og 2-7 spleises, vi får 2-3-7. Vi må snu i 2 fordi nullstikravet er brutt.