Prøveeksamen 2006 med svarforslag

UNIVERSITETET I OSLO

Det matematisk-naturvitenskapelige fakultet

Prøve-eksamen i:	**INF 3130/4130:** ***Algoritmer: Design og effektivitet***
Eksamensdag:	**Gjennomgås 30. november, kl 14.15**
Tid for eksamen:	**3 timer**
Oppgavesettet er på:	**4 sider**
Vedlegg:	**Ingen**
Tillatte hjelpemidler:	**Alle trykte og skrevne**

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

Alle syv oppgavene har lik vekt, altså omtrent 14%.
Les oppgavene nøye, og lykke til!

Oppgave 1

L = {(M_1, M_2) | M_1 og M_2 er Turingmaskiner som er ekvivalente, dvs. gir samme output for samme input}

a) Bevis at L er uavgjørbart.

b) Kommenter betydning av øverste resultatet i følgende sammenheng: Det ville vare fint hvis vi kunne teste programmer ved å sammenligne dem automatisk (ved hjelp av et generelt testeprogram) med et kjent korrekt program for samme funksjon. Er dette mulig? Hvorfor?

Svarforslag, oppgave 1:

Først en liten kommentar til oppgaven: Den forståelsen av L vi faktisk bruker i beviset under er følgende:
L = {(M_1, M_2) | M_1 og M_2 er Turingmaskiner som er ekvivalente, dvs. at for et gitt input x vil enten begge ikke stoppe eller begge stoppe}

a) Anta at L faktisk er avgjørbart (noe vi håper vil føre til en selvmotsigelse). Da må det finnes en Turingmaskin (eller en algoritme) M_E som avgjør L. Vi kan da bruke M_E 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 Ekvivalens (M1, M2) som output. Maskiner M1 and M2 er som følger:

M1: HALT
M2: SIMULER MASKIN M MED INPUT x

Det er ikke vanskelig å verifisere at R er en reduksjon:

(i) Vi viser først at M1 og M2 er ekvivalente hvis og bare hvis M stopper ved input x. For å se dette, merk at hvis M stopper ved input x, så vil også simuleringen M2 stoppe. Dermed vil M1 og M2 være ekvivalente. Hvis M derimot ikke stopper ved x, så vil M1 stoppe (for enhver input) og M2 aldri stoppe. Dermed vil de ikke være ekvivalente.

(ii) Vi viser så at R kan beregnes av en Turingmaskin (er en algoritme). For å se dette, merk at R er en enkel modifikasjon av standardreduksjonen. Hele M1 og nesten hele M2 er gitt på forhånd som konstante strenger i R. M1 er en triviell algoritme. M2 er en enkel modifikasjon av Den Universelle 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 M2s egen kode. R skal først skrive ut M1 og så M2 (som altså inneholder M og x fra inputen til R på bestemte steder).

Siden R er en reduksjon fra Stoppeproblemet til Ekvivalens, gitt M_E, så kan vi løse Stoppeproblemet ved først å kjøre R og så M_E . 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.

b) Resultatet betyr at det ikke er mulig å sammenligne programmer (og avgjøre om de er ekvivalente) automatisk. Church-Turing tesen sier at en Turingmaskin kan utføre alle beregninger som en hvilken som helst datamaskin eller et hvilket som helst programmering språk. For, hvis det var mulig å sammenligne programmer automatisk, så kunne dette også gjøres med Turingmaskiner, men vi har nettopp bevist at dette er umulig.

Oppgave 2

Svar JA eller NEI og gi en kort begrunnelse.

a) Optimaliserings-versjonen av noen NP-komplette problemer har effektive approksimasjonsalgoritmer.

b) Det finnes NP-komplette problemer der optimaliserings-versjonen *ikke* har effektive approksimasjonsalgoritmer med mindre P er lik NP.

c) En parallell-maskin med en million prosessorer kan løse *Hamiltonbarhet* ("Hamiltonicity", for urettede grafer) i polynomisk tid

Svarforslag, oppgave 2:

a) JA. Eksemplene gitt på forelesningen er NODEDEKKING (VERTEX COVER) og HANDELSREISENDES PROBLEM (TSP) med trekant-ulikhet.

b) JA. Eksempel gitt på forelesningen er (det generelle) HANDELSREISENDES PROBLEM.

c) NEI. Siden en sekvensiell maskin kan simulere ett skritt av en million-prosessor-maskin i polynomisk tid, så kan vår parallelle maskin i polynomisk tid bare løse problemer som er i P.

Oppgave 3

Vi skal løse følgende problem: Vi får som input to strenger: S som består bare av bokstaver, og T som også kan inneholde '*'-er. Spørsmålet er om strengene kan gjøres like, om vi får lov til å erstatte hver '*' i T med ingeneting (en tom streng) eller én selvvalgt bokstav. For S="abbcf" og T="a*bc*f", kan dette gjøres slik: T'="a(b)bc()f". For strengene "acb" og "a*c" er det derimot umulig. Beskriv en algoritme som avgjør om man kan oppnå likhet, og som bygger på dynamisk programmering. Skisser et program som implementerer algoritmen.

Hint: Forsøk med et skjema med de to strengene langs hver sin akse, og med boolske verdier. Skill mellom den generelle reglen, og initialiseringen for de ruter der minst en av strengene er tomme.

Svarforslag, oppgave 3:

I eksempelet under har vi brukt S = "abbcdc" og T = "a*bcd*c". Etter hintet prøver vi oss, på tradisjonell måte, med å ha en to-dimensjonal boolsk array B[0:S.length, 0:T.length], der B[i,j] rett og slett skal angi om det kan oppnås likhet mellom S[1:i] og T[1:j]. Null'te kolonne og null'te rad skal initialiseres på passelig måte, men vi tar først det generelle tilfellet. I tabellen under er true markert med 't', mens false er angitt ved et punktum.

S[i] er alltid en bokstav, og anta først at også T[j] er en bokstav. Om disse er ulike kan det opplagt ikke oppnås likhet mellom S[1:i] og T[1:j]. Dersom S[i] og T[j] er like kan det kanskje oppnås likhet, men bare om S[1:i -1] og T[1:j -1] kan gjøres like, altså at B[i-1, j-1] er true.

Anta så at T[j] er '*'. Det er da hvertfall klart at om S[1:i] og T[1:j -1] kan gjøres like (altså om B[i, j-1] er true), så kan også S[1:i] og T[1:j] gjøres like, ved at vi lar '*' i T[j] gå til den tomme streng. Men det er også slik at om S[1:i -1] og T[1:j-1] kan gjøres like (altså om B[i-1,j-1]==true) så kan også S[1:i] og T[1:j] gjøres like ved at vi lar '*' i T[j] gå til tegnet S[i].

```
                  T j
    B   | 0  1  2  3  4  5  6  7
        |    a  *  b  c  d  *  c
    ----|-----------------------
    0   | t  .  .  .  .  .  .  .
    1 a | .  t  t  .  .  .  .  .
S   2 b | .  .  t  t  .  .  .  .
i   3 b | .  .  .  t  .  .  .  .
    4 c | .  .  .  .  t  .  .  .
    5 d | .  .  .  .  .  t  t  .
    6 c | .  .  .  .  .  .  t  t
```

Når det gjelder initialiseringen så skal vi hvertfall angi at den tomme strengen S[1:0] kan gjøres (eller snarere *er*) lik den tomme strengen T[1:0], og dermed skal B[0,0] settes til true. I tillegg må vi sjekke om T starter med en '*', og i så fall må også B[1,0] settes til true (og om T starter med flere *-er må det gjøres tilsvarende). Forøvrig skal alle verdiene i null'te kolonne og null'te rad settes til false.

Svaret på hele spørsmålet blir opplagt stående i B[S.length, T.length]. En programskisse av det hele kan da se omtrent slik ut:

```
B[0,0] = true;
for i = 1 to S.length do { B[i,0] = false; }
for j = 1 to T.length do { B[0,j] = false; }
if T[1] == '*' then { B[0,1] =  true; }  // Forutsetter T ikke starter med flere '*'-er på rad

for i = 1 to S.length do {           // Rekkefølgen av løkkene er vilkårlig
    for j = 1 to T.length do {
        if T[j] == '*' then {
            if B[i ,j-1] == true or B[i-1,j -1] == true then { B[i,j] = true; }
        } else {
            if S[i] == T[j] and B[i -1,j -1] == true then { B[i,j] = true; }
        }
    }
}
return B[S.length, T.length];
```

Oppgave 4

Vi skal søke i strengen "banananobana" etter patternet "nano", på litt ulike måter.

a) Beregn NEXT[0:3]-arrayet, som Knuth-Morris-Pratt-algoritmen bruker.

b) Beregn SHIFT[a:z]-arrayet som den forenklede Boyer-Moore-alagoritmen (kalt Horspool-algoritmen på forelesningen) bruker; altså algortimen som beregner skift bare ut ifra den siste bokstaven i den aktuelle delen av strengen.

c) I denne deloppgaven skal vi søke i et to-dimensjonalt array, A, etter et to-dimensjonalt pattern, P.

A =

b	a	n	a	n	a	n	o	b	a	n	a
n	a	n	a	n	o	b	a	n	a	b	a
n	a	n	o	b	a	n	a	b	a	n	a
…											

P =

n	a
n	o

Forklar, kort, gjerne med pesudo-kode, men på en slik måte at ideen din kommer klart fram, hvordan man kan kan tilpasse eller sette sammen de søkealgoritmene vi har studert i kurset, på en slik måte at vi kan klare to-dimensjonale søk.

Svarforslag, oppgave 4

a) NEXT[i], $2 \leq i \leq$ m-1, er lengden av lengste prefix av P[0 : i-2] som er likt et suffix av P[1 : i-1]

NEXT[0] : 0 (per def)
NEXT[1] : 0 (per def)

NEXT[2] : Lengste prefix av P[0 : 2-2] = ”n” som er likt et suffix av P[1 : 2-1] = ”a” – 0.
NEXT[3] : Lengste prefix av P[0 : 3-2] = ”na” som er likt et suffix av P[1 : 3-1] = ”an” – 1 .

b) SHIFT er den korteste avstanden en bokstav har fra siste bokstav i patternet. Skal beregnes for alle bokstaver i alfabetet, for bokstaver som ikke finnes i patternet setter vi avstanden til patternets lengde.

SHIFT [n] = 1 (nærmeste n i patternet er nabo med o)
SHIFT [a] = 2 (a ligger to tegn unna o)
SHIFT [o] = 4 (finnes ingen andre steder i patternet enn til slutt)
SHIFT [*] = 4 (alle andre bokstaver)

c) Oppgaven er kun ment som en liten “tenk igjennom”-oppgave.

Det enkleste man kan gjøre er legge tabellen ut som en 2-etasjers streng. Rad 1 og 2 settes sammen med rad 2 og 3, 3 og 4 osv. Så kan man benytte en vanlig streng-algoritme ved å se på kolonnene i ”strengen” (disse består av to tegn) som *en* bokstav.

(Dette er selvsagt ikke helt optimalt, ettersom vi ikke nyttiggjør oss av lengre skift enn 1 i vertikal retning.)

Oppgave 5

Under kommer vi med noen påstander omkring trær og prioritetskøer. Angi for hver av dem om påstanden er riktig, og gi en kort begrunnelse.

1. Binomialtreet B_k kan lages fra treet B_{k-1} ved å legge ett nytt barn til hver node.
   (Hint: Betegnelsen *binomialtre* refererer til strukturen i treet, ikke nødvendigvis måten de bygges opp på.)

2. Dersom vi implementerer prioritetskøer ved AVL-trær (med prioriteten som innsettingsnøkkel), så vil elementer med samme prioritet alltid komme ut (ved kall på deleteMin) i den rekkefølge de er satt inn. (For at det skal være håp om at dette går bra, antar vi at når et element skal settes inn i treet og treffer på en node med samme prioritet, så går man videre i *høyre* subtre for å finne plass til det).

3. Desom vi implementerer prioritetskøer som ”leftiest heaps”, så vil elementer med samme prioritet alltid komme ut (ved kall på deleteMin) i den rekkefølge de er satt inn.

4. Høyden av det høyeste treet i en Fibonacci-heap med n elementer ligger i intervallet fra og med $\lfloor \log_2 n/2 \rfloor + 1$ til og med $\lfloor \log_2 n \rfloor$.

5. Antall barn en node kan ha i en Fibonacci-heap på n elementer ligger i intervallet fra og med 0 til og med $\lfloor \log_2 n \rfloor$.

Svarforslag, oppgave 5

Pkt. 1: *Riktig.* Enkle forsøk viser jo at det stemmer for f.eks. B_0 og B_1. For å vise at det stemmer generelt kan man bruke induksjon. Basisen i induksjonen kan være at det stemmer for B_0, og vi må vise at om det stemmer for B_n så stemmer det også for B_{n+1}:

Vi vet at B_{n+1} består av to B_n-trær, satt sammen slik at det ene er subtre til roten av det andre. Alle nodene i B_{n+1} er altså i det ene eller det andre av disse B_n-trærne. Det å sette inn et nytt ekstra barn for hver node i B_{n+1} vil dermed være det samme som å sette inn et nytt barn for hver node i hvert av de to B_n-trærne. Ut fra induksjonshypotesen blir hver av disse da lik B_{n+1}, og når de sees som et sammensatt tre får vi dermed B_{n+2}. Altså vil innsetting av nye barn i B_{n+1} gi B_{n+2} og dermed er også induksjonssteget vist.

(Et mindre formelt argument kan være å observere at hver node i et binomialtre B_k har vært (rot i) et B_0-tre, en eller annen gang i prosessen, om vi bygger opp treet på den vanlige måten. På samme måte har alle noder, bortsett fra løvnoder, i et binimialtre B_{k+1} også vært rot i et B_1-tre. Utvidelsen fra B_0-trær til B_1-trær kan vi godt gjøre til slutt når vi går fra $B_{k \text{ til }} B_{k+1}$.)

Pkt. 2: *Riktig.* Uten at det blir viktig, kan vi først merke oss at det å ta ut noden med minst verdi fra et AVL-tre ikke er å ta ut roten, men å gå fra roten og til venstre subnode gjentatte ganger, inntil man finner en node der venstre subtre er tomt. Denne noden har minste verdien i hele treet (og det å ta den ut er også lett, siden den har minst ett tomt subtre).

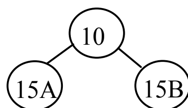
Så til oppgaven: Når vi passer på å gjøre innsettingen slik oppgaven sier, setter vi en node inn på det stedet i treet den ville komme om prioriteten var hårfint større enn de med samme prioritet som var i treet fra før. Mens denne noden så ligger i treet kan den bli involvert i mange rotasjoner, og man kunne tenke seg at disse kunne ødelegge dette forholdet. Men det skjer ikke, siden ideen ved disse rotasjonene nettopp er at det som var av større/mindre relasjoner mellom nodene blir bevart. Derved vil den stadig bli liggende som om den hadde en slik hårfint større verdi enn de med samme prioritet, og i forbindelse med deleteMin vi den derved også komme ut etter alle de med samme prioritet som lå i treet fra før da den ble satt inn.

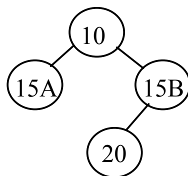
Pkt. 3: *Ikke riktig.* Intuisjonen for at "leftiest heap"-operasjone ikke bevarer innsettings-rekkefølgen er at to noder N1 og N2 med lik prioritet kan ligge langt nede i treet, og at deres dypeste felles supernode kan komme til å bytte om sine to subtrær ut fra helt andre grunner enn det som har med hvilken rekkefølge N1 og N2 ble satt inn i. Avgjørelsen om ombytting taes jo ut fra hvilke andre noder som er i treet, og hvordan de er plassert i treet.

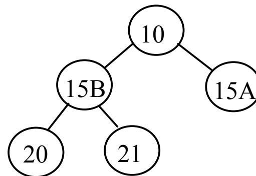
En god forklaring som likner på den over burde være svar godt nok, men det absolutt ultimate er jo å vise et eksempel. Vi må da bestemme oss for nøyaktig hvordan algoritmen behandler de tilfellene der den kan gjøre vilkårlige valg. Vi antar følgende:

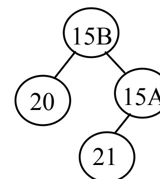
(1) Om to trær skal flettes sammen (merge) og det blir et valg mellom noder med samme prioritet, så vil de fra det "høyre" treet bli lagt lenger fra roten enn de fra "venstre".
(2) Når vi skal sette inn en ny node så regnes treet som bare består av denne som et høyre tre under flettingen (denne blir egentlig ikke brukt i eksempelet).

Et eksempel kunne da være følgende: Sett inn i rekkefølge 15A, 10, 15B, 20, 21. De tre første trærne under er stadier under denne innsetningen. Når vi så kaller deleteMin én gang (og får ut 10), får vi det siste treet angitt under. Ved et kall på deleteMin én gang til får vi så ut 15B, slik at 15B altså kommer ut før 15A, som var satt inn først. Ved andre valg for (1) og (2) over kan tilsvarende eksempler lages.









Pkt. 4: *Ikke riktig*. I sære tilfeller kan heapen bestå av ett eneste tre, som er en lang streng av n noder, slik at høyden vil være n-1 (hvis en teller antall kanter), evt. n (om en teller antall noder). En slik heap kan også bli en liste av trær med bare én node i hvert tre (sett inn mange, uten å ta noen ut), og da blir maks høyde lavere enn nedre-grensen av intervallet.

Pkt. 5: *Riktig.* Roten i binomialtreet B_k har k barn. Det største treet vi kan ha i en heap på n elementer som er bygget med insert og delete, er $\lfloor \log_2 n \rfloor$. To heaper på m og m´ elementer, satt sammen med merge vil begge kun bidra med trær som er mindre enn $\lfloor \log_2 (m + m') \rfloor$.

Oppgave 6

Vi har et rutenett, der rutene er nummerert "nedover" med i = 1 til m, og "bortover" med j = 1 til n. Til hver rute er det knyttet en heltallig kost K(i,j) > 0. Vi ønsker å lage en A*-algoritme som får oppgitt to ruter: r1 = (i1,j1) og r2 = (i2, j2) (samt m, n og alle K(i,j)), og som finner "beste" vei (innenfor det gitte rutenettet) fra r1 til r2. Vi vil i den forbindelse se på mulige heuristikk-funksjoner.

En vei består av enkeltskritt, som går fra en rute til en av de 8 naborutene, diagonalt, horisontalt eller vertikalt. Kosten av en vei er summen av kosten av de rutene den er innom, der r2, men ikke r1 regnes med, og slik at om r1 og r2 er samme rute (og veien bare består av denne), så er kosten 0. (Vi kan mer presist si at K(i, j) er det det koster å gå *inn* i ruten (i,j)). Beste vei er den som har minst kost. Under er angitt et rutenett med m=5 og n=8, og med en vei fra r1 = (4, 3) til r2 = (2, 7). K-verdiene er gitt for rutene langs veien, og veien har altså kost = 32.

	j = 1	2	3	4	5	6	7	8
i = 1				2	2			
i = 2			4			7	8	
i = 3		9						
i = 4			3					
i = 5								

a) Under finner du to forslag til to heuristikker for et A*-søk etter beste vei. Vi antar at vi står ved rute r = (i, j), og at målet er r2 = (i2, j2). Angi for hvert forslag om det gir en monoton heuristikk-funksjon, og forklar kort hvorfor eller hvorfor ikke.

Forslag 1: $| i2-i | + | j2-j |$.

Forslag 2: $minK * (min(|i2-i|, |j2-j|) + | |i2-i| - |j2-j| |)$.
Her er minK lik kosten av den ruten med minst kost på hele brettet, og "min(a,b)" er den minimale

verdien av tallene a og b, på vanlig måte.

b) (Vent gjerne med denne til slutt) Angi en heuristikk-funksjon som er monoton, og som, om noen av de angitt i del a) skulle være monotone, er bedre enn denne/disse. Forklar kort.

Svarforslag oppgave 6:

a) Forslag 1 er ikke en monoton heuristikk, i og med at enhver monoton heuristikk gir en h-verdi som ikke er større enn den virkelig korteste vei (se linjene under definisjonen, side 723 i læreboka, samt en foil fra forelesningen). Et eksempel på at denne heuristikken ikke tilfredstiller dette kan være et 2x2-rutenett, der r = (1,1) og r2 = (2,2), og der alle koster er 1. Kosten av korteste vei er da 1 (gå diagonalt), mens den angitte heuristikken gir 2.

Forslag 2 er derimot en monoton heuristikk. Kall det som står inne i den ytterste parentesen for L_r. Vi ser da at L_r angir det minimale antall skritt fra r til r2, ved at ”min(|i2-i|, |j2-j|)” angir hvor mange skritt vi kan gå diagonalt (= siden i det største kvadratet som kan innskrives i rektangelet spent ut av r og r2), mens ”| |i2-i| – |j2-j| |” angir det som står igjen, og som må gåes horisontalt eller vertikalt. Siden vi ganger L_r med den minste rute-kosten, er det opplagt at denne heuristikken blir mindre enn kosten til enhver vei fra r til r2. Dermed er det hvertfall *håp* om at den er monoton.

Vi kaller den foreslåtte heuristikken h, og ut fra monotonitetskravet (def., side 723) må vi da vise at ”h(ra) <= h(rb) + kost(ra, rb)”, hver gang vi kan gå i ett skritt fra ra til rb (de er naboruter), og kost(ra,rb) er kosten av dette skrittet. Anta at rb=(ib,jb). Da er kost(ra, rb) opplagt lik K(ib,jb), og vi vet at minK<=K(ib,jb). Vi observerer da først at $L_{ra} <= L_{rb} + 1$, siden L_{ra} og L_{rb} er antall skritt i korteste veier til r2, og at vi hvertfall kan gå fra ra til r2 gjennom rb. Vi får da:

$$h(ra) = minK * L_{ra} <= minK * L_{rb} + minK <= minK * L_{rb} + K(ib, jb) = h(rb) + kost(ra,rb)$$

At h(ra) = 0 dersom ra = r2 er opplagt. Dermed ser vi at monotonitetskravet er oppfylt.

b) ”Bedre heuristikk” kan egentlig tolkes på to måter: (1) ved at funksjonen h(x) gir et svar som er nærmere (men stadig ikke over) kosten av den beste veien til målet og (2) ved at heuristikken gir et bedre A*-søk. Et eksempel på en som tilfredsstiller (1) men ikke (2) er å observere at om r ikke er r2, så må kosten til r2 alltid være med, og den kan være større enn minK. Vi kan derfor, når r er ulik r2, bruke:

$$h(r) = minK* (L_r - 1) + K(i2, j2)$$

Denne vil imidlertid ikke gi bedre A*-søk, siden leddet K(i2, j2) kommer inn på samme måte for alle ruter under søket.

Et alternativ, som *vil* gjøre en forskjell i søket, er at vi først går gjennom hver av rutene r i rutenettet, og finner den minimale kosten av de 8 naboene, og lagrer den som M_r. Da kan vi tilsvarende bruke:

$$h(r) = minK * (L_r - 1) + M_r \quad \text{dersom r er ulik r2. Når r er lik r2: } h(r) = 0$$

Denne vil gi forskjellige forbedring av h-funksjonen for forskjellige ruter r, og kan derved gi bedre søk. Vi kan selvfølgelig i stedet se på naborutene hver gang vi er ved en rute, og ikke forhånds-beregne. Monotonitetsbeviset for denne er nokså likt det over, og går som følger (for ra ulik r2). Anta at man kan gå fra ra til rb i ett skritt. Da vet vi:

$$h(ra) = minK * (L_{ra} - 1) + M_{ra} <= minK * L_{rb} + M_{ra} = minK * (L_{rb} - 1) + minK + M_{ra}$$

Vi vet her at $minK <= M_{rb}$ og $M_{ra} <= kost(ra, rb)$, og får derved:

$$h(ra) <= minK * (L_{rb} - 1) + minK + M_{ra} <= h(rb) + kost(ra, rb)$$

At h(ra) = 0 dersom ra = r2 er riktig, pr. definisjon.

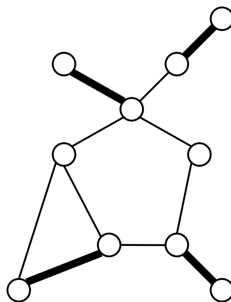
Det kunne være fristende også å se på følgende forslag (som dog ville kreve mye forberedelse): Vi sorter først alle rutekostene, og kan da lett forhåndsberegne hvor mye en vei av lengde 1, av lengde 2 osv. minimum vil koste. Dette kan vi så bruke i stedet for $\min K^*L_r$ i Forslag 1. Det viser seg imidlertid at denne ikke er monoton (og den er dermed et eksempel på at selv om h(v) ikke overgår minste kost fra v til målet, så behøver den *ikke* å være monoton). Et eksempel som viser at den ikke er monoton er som følger: Anta at kosten på rutene er:

```
2 2 2
2 1 2
2 2 2
```

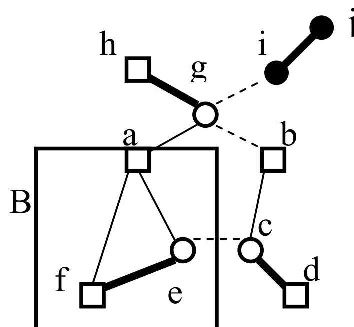
og at ra = (1,1) og rb = (2,2), og at målet r2 = (3,3). Da blir h(rb) = 1 og h(ra) = 1+2 = 3. Siden kost(ra,rb) = 1, så gjelder ikke h(ra) <= h(rb) + kost(ra, rb).

Oppgave 7 Matching

Gitt grafen under, med den angitte matching. Vis hvordan algoritmen for maksimal matching i generelle grafer vil avgjøre om dette er en maksimal matching eller om den kan økes. Gi et eksempel på hvordan nodene kan være merket når algoritmen tar avgjørelsen. Merk blomster som oppstår med en passelig boks/ring rundt dem, og bruk stipling på kanter som hverken er med i noe tre eller i matchingen. Bruk ellers F(irkant)-noder, P(rikk)-noder og S(irkel)-noder, som i kompendiet. Beskriv hva som er avgjørende for at algoritmen tar den avgjørelsen den tar.



Svarforslag på oppgave 7 (Sjekkes av Stein):



Det første som skjer er at alle de umatchede nodene, nemlig a og b, merkes som røtter i hvert sitt tre, altså som F-noder, og at alle de andre merkes som P-noder (fylte). Deretter ser vi etter (umatchede) kanter fra F-noder til enten andre F-noder, eller til P-noder (noder vi ikke har sett på).

La oss si at vi først finner (a,e). Dette fører også (e,f) inn i treet, med e som S-node og f som F-node.

Vi kan så tenke oss at vi ser F-til-F-kanten (f,a). Begge disse nodene er allerede i samme tre, og det vil si at vi får en blomst B, angitt ved en firkant i figuren. Siden en blomst er en F-node kunne vi nå ha valgt kanten (e,c), eller egentlig (B,c), og dermed fått (e,c) og (c,d) inn i det samme treet. I stedet kan vi si at vi finner kanten (b,c), og dermed blir c en S-node og d en F-node, og de kommer inn i treet med rot b. Så kan vi tenke oss at vi ser på kanten (a,g), eller altså egentlig (B,g), og da kommer g og h inn i det første treet slik som angitt. Vi kunne like gjerne valgt kanten (b,g), og da ville g og h kommet i det andre treet (men med samme merking).

Nå ser vi at det ikke går noen kanter vi ikke har sett på fra F-noder (eller blomster) til F-noder eller til noder vi ikke har sett på (P-noder). Nodene i og j kommer altså aldri inn i noe tre, og forblir P-noder. Dermed stopper trebyggingen, og siden det ikke er blitt riktig type kontakt mellom de to trærne viser det at det ikke finnes noen større matching.