INF 3/4130
6. oktober 2005

- Dagens: Kapittel 10, 23, litt fra kap 14, samt matchin i generelle grafer
  - Fra kap 10 : Dybde-først og branch-and-bound søk
  - Fra kap 23: A*-søk
  - Om å finne den største matchingen i en ikke-bipartit graf (Notat om dette kommer!)
  - Utgår: Om å finne tyngste matching i en bipartite graf med kantvekter
- Presisert utgave av oblig 1 lagt ut mandag morgen (3/10).
- Ingen forelesning eller grupper neste uke
- Forelesning om to uker:
  - Dino Karabeg snakker mer om NP-kompletthet etc.
- Ikke alt faglig er bestemt for slutten av kurset
  - Ønsker som for øvrig passer inn vil bli vurdert
- Midtveis-vurdering av kurset kommer

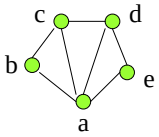
Søk i tilstandsrom

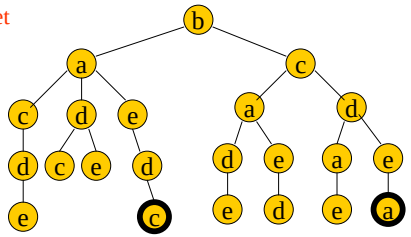
- Kap 10: Herfra skal vi ”minne om” (fra INF 1020)
  - Backtracing algorithms
    - dybde først søk i tilstandsrommet
    - Trenger lite lagerplass
  - Branch and bound
    - Bredde først søk
    - Trenger mye plass: Må holde i lageret alle noder (tilstander) som er ”sett”, men som ikke er studert
    - Kan også gi hver node en ”lovende-het”, og gå videre langs den noden som er mest lovende (heuristikk). Datastruktur: Prioritetskø, likner på Dijkstras ”korteste vei”
  - Et alternativ om man vil gjøre rent bredde-først-søk (ikke i boka):
    - Gjør dybde-først-søk til nivå 1, så nytt søk til nivå 2, osv.
    - Om det er stor forgreningsfaktor tar ikke dette så mye mer tid enn vanlig bredde-først
    - Og det krever mye mindre plass
- Kap 23
  - Herfra skal vi ta om A*-søk.
    - Likner mye på branch-and-bound med prioritetskø

Modeller for avgjørelsessekvenser

- Det er flere måter å ”modellere” avgjørelsessekvenser for et gitt problem
- Finne Hamiltonian Cycle:
  - Start vei i tilfeldig node og forleng veien på alle mulige måter
    - Mulige valg i steget i algoritmen: Alle kanter (ut fra siste vei-node så langt) som ikke går tilbake til allerede brukt node.
  - Start med en kant, og legg stadig til en kant til:
    - Mulige valg i steget i algoritmen: Alle ikkevalgte kanter som gjør at alle de sammenhengende komponentene av valgte kanter fremdeles forblir enkle veier
- Fører til forskjellige ”state space tree” = ”tilstandsrom-treet”
- ”problem state” Tilstander der en del valg er gjort
- ”goal states”: Det gjort et antall valg, og vi står med en løsning.

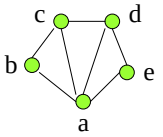
Tilstandstreet til høyre er laget ut fra første modellen over:

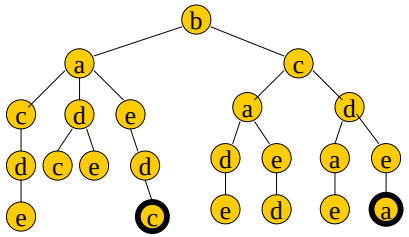




Dybde-først søk/tilbakesporing

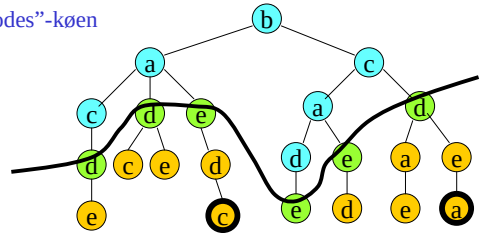
- Gjennomsøker tilstandsroms-treet dybde-først, til vi kommer til en ”mål-tilstand”
- Bruker greiest en rekursiv prosedyre, som har selve problemstillingen som ”globale data”.
- Tar liten plass
- Kan bruke ”heuristikk” til å velge mest lovende vei ut av den noden vi nå står i (f.eks. nærmeste-kant-først, om man vil finne korteste Ham. Cycle).
- Må bruke avskjæring så godt som mulig (pruning, bounding): Ikke gå ned i subtrær som umulig kan inneholde en ”mål-tilstand”





”Branch and bound”

- Bruker en eller annen form for bredde-først-søk
- Blir en mengde av noder som boka kaller ”Live Nodes”
  - Dette er de som er ”sett, men ikke fulgt opp”. NB: Kan bli stor!
- ”Live nodes” vil være et snitt gjennom tilstandsrom-treet (grønne under)
- Steget: Velg en node N fra mengden LiveNodes
  - Er N en mål-node? Om ja: Ferdig! Om nei:
  - Ta N ut av ”liveNodes”-køen
  - Sett alle N’s barn inn i ”Live Nodes”-køen
- Tre strategier:
  - LN-mengden er en FIFO-kø
    - Ekte bredde først
  - LN-mengden er en LIFO-kø
    - Likner på dybde-først
  - LN-mengden er prioriteskø,
    - med en passelig heuristikk som prioritet
    - Likner mye på A*-søk (kommer)

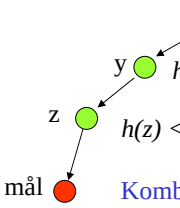


A*-søk

- A*-søk egner seg for problemer der vi har
  - en (eksplisitt eller implisitt) graf av ”tilstander”,
  - Med en start-tilstand, og et antall mål-tilstander
  - Mulige tilstands-overganger (rettede kanter) med en gitt ”kost”.
  - Og: Skal finne en vei fra start til en mål-tilstand med, med minimal kost.
  - Altså logisk sett: korteste-vei-problemet
  - Se figur 23.2, side 718, for deler av en slik graf. Ingen mål-tilstand kom med, og alle overganger kan gåes begge veier (dermed en urettet graf).
- Strategien er et bredde-først-søk
  - Der vi bruker en heuristikk-funksjon *h(x)* for stadig å velge mest lovende vei
  - Altså bredde-først-søk med en priorites-kø for valg av neste fra LiveNodes
  - Minner derved mye om Dijkstras korteste-vei algoritme.
- Krav til heuristikk-funksjonen *h(x):*
  - Den skal være en tilnærmelse til hvor langt det er fra x til nærmeste mål-tilstand
  - Krav for å være A*: *h(x)* skal aldri være større enn den faktisk korteste vei til nærmeste mål.
  - I tillegg kreves gjerne en slags trekant-ulikhet for *h(x)*. Det gjør algoritmen mye raskere.

Krav til *h(x)*

- Dersom *h(x)* aldri er større enn faktisk korteste vei fra x til nærmeste mål,
  - Og vi bruker en Dijkstra-liknende algoritme, med passelig bruk av *h(x)*
  - Da vil vi alltid til slutt få riktig resultat (korteste vei fra start til nærmeste mål)
- Men vi må stadig gå tilbake til noder ”vi trodde vi var ferdig med”
  - og oppdatere lengden, og dermed få mye ekstra-arbeid!
- Vi legger derfor på et ekstra ”monotonitets-krav” på *h(x):*
  - Om det er kant fra x til y med vekt *w(x,y)*, så skal gjelde: $h(x) <= h(y)+w(x,y)$
  - Alle mål-noder *m* har $h(m) = 0$
- Det fine er at dette kravet medfører det første kravet, så vi slipper å tenke på det. Bevis: Vi antar at x -> y -> z -> mål er korteste vei fra x til mål:



$h(x) <= h(y) + w(x,y)$

$h(y) <= h(z) + w(y,z)$

$h(z) <= h(mål) + w(z,mål) = w(z,mål)$

Kombinerer vi disse får vi: $h(x) <= w(x,y) + w(y,z) + w(z,mål)$

Data for selve A* algoritmen (se side 725/726)

- Vi har en rettet graf G med kantvekter *w(x,y)*, en startnode og et antall mål-noder, samt en monoton heuristikk-funksjon *h(x).*
- Hver node x har i tillegg følgende variable:
  - *g(x)* som stadig vil forandre seg under algoritmen, men som til slutt får lengden av korteste vei fra startnoden til x.
  - *parent(x)* som skal bli foreldre-peker i et tre av korteste veier fra start-noden
  - *f(x)* som hele tiden er lik *g(x)+h(x)*, altså et estimat av veilengden fra start til et mål gjennom x.
- Vi har en prioritets-kø PQ av noder, der prioriteten går på verdien av *f(x)*
  - Denne initialiseres med bare start-noden s, med *g(s)=0*, og *h(s)* vilkårlig. (Dette mangler i selve prosedyre-beskrivelsen i boka, side 725)
- De nodene som for øyeblikket ikke er i PQ deles i to typer
  - Tre-noder: Disse har en foreldre-peker i et tre med startnoden som rot (korteste vei til roten-treet). Disse har alle vært i PQ, og ved starten er det ingen slike tre-noder.
  - Usette noder (de vi ikke har kommet borti så langt)

Selve algoritmen (se side 725/726)

- PQ initialiseres altså med bare start-noden, med $g(s) = 0$ (alle andre er usette)
- Steget, som gjentas så lenge PQ ikke er tom eller vi treffer en mål-node:
  - Plukk den best prioriterte noden x fra PQ (med minst *f*-verdi)
  - Dersom x er en mål-node, slutter herved algoritmen
    g(x) og parent(x) angir korteste vei fra startnoden og den aktuelle veien (baklengs).
  - Ta x ut av PQ, og la den bli en tre-node (den har sin foreldre-peker og g(x) satt riktig)
  - Se på alle naboer til x blant de usette noder, og for hver slik y:
    Sett $g(y) = g(x) + w(x,y)$, $f(y) = g(y) + h(y)$ samt $parent(y) = x$ og sett y inn i PQ
  - Se på alle naboer til x i PQ, og for hver slik y:
    Dersom $g(y) > g(x) + w(x,y)$ så sett $g(y) = g(x) + w(x,y)$ og $parent(y) = x$
  - Vi ser altså *ikke* på de naboene til x som er tre-noder! (At det går bra krever et bevis som kommer på de neste foilene)

Algoritmen kan altså slutte på to måter:

- Ved at PQ blir tom. Det betyr at det ikke går noen vei fra startnoden til en mål-node
- Ved at vi kommer til en mål-node m, og da er $g(m)$ lengden av korteste vei fra startnoden til *m* og $parent(m)$ angir selve veien.

Om vi bruker h(x) = 0 for alle noder, så blir dette Dijkstras korteste-vei-algoritme

A*-søk går raskt med monoton $h(x)$

- Om vi bruker $h(x) = 0$ for alle noder, så blir dette Dijkstras korteste-vei-algoritme.
- Ved å bruke heuristikken håper vi å konsentrere oss mer om de veiene som fører til et mål, slik at algoritmen går raskere.
- Men, vi tar da nodene ut av PQ i en annen rekkefølge enn i Dijkstra-algoritmen
- Dette kunne føre til at riktig veilengde ikke er kommet inn i en node x når den taes ut av PQ og over i treet (slik at vi stadig måtte gå tilbake og oppdatere $g(v)$ og $parent(v)$ for noder *y* i treet (som har forlatt PQ)

Heldigvis gjelder (proposition 23.3.2 i boka):

- Om $h(x)$ er monoton, så vil verdiene av $g(x)$ og $parent(x)$ alltid ha blitt riktige i det øyeblikk x taes ut av PQ over i treet.
- Dermed behøver vi aldri gå tilbake i treet og oppdatere noe.
- Og algoritmen blir av samme orden som Dijkstra-algoritmen
- Bevis på neste foil. Merk: Det er en viktig trykkfeil på side 724, formel 23.3.7:
  - Der det står: … $h(v) + h(v)$ … skal det stå … $h(v) <= g(v) + h(v)$ …

Bevis: Jeg mener vi må bruke induksjon (ikke i boka): Induksjonshypotese: Setningen gjelder for alle y som er flyttet fra PQ til treet *før* x. Vi viser at da gjelder den også for x.

- Vi lar generelt $g^*(v)$ være lengden av korteste vei fra start-noden til noden v.
- Vi ser på situasjonen når x taes ut av PQ, og vi ser på en nodesekvens P:
  start-noden = $v_0, v_1, v_2, \ldots, v_j = x$
  som er en korteste vei fra start-noden til x (med lengde $g^*(x)$)
- Vi antar at $v_0, v_1, \ldots, v_k$ (men ikke v_{k+1}) er blitt tre-noder når x taes ut av PQ.
- Noden v_{k+1} er altså i PQ når x blir tatt ut av køen.
- Ut fra monotoniteten vet vi (for $i = 0, 1, \ldots, i-1$)
  $g^*(v_i) + h(v_i) <= g^*(v_i) + h(v_{i+1}) + w(v_i, v_{i+1})$
- Siden kanten fra v_i til v_{i+1} er med i en korteste til v_{i+1}, gjelder
  $g^*(v_{i+1}) = g^*(v_i) + w(v_i, v_{i+1})$
- Til sammen gir de to siste: $g^*(v_i) + h(v_i) <= g^*(v_{i+1}) + h(v_{i+1})$
- som så gir, ved å la *i* være $k+1, k+2, \ldots, j-1$
  $g^*(v_{k+1}) + h(v_{k+1}) <= g^*(v_j) + h(v_j) = g^*(x) + h(x)$
- Ut fra induksjonshypotesen vet vi at $g(v_k) = g^*(v_k)$, og dermed må også (ut fra aksjonen når v_k ble tatt ut av PQ) $g(v_{k+1}) = g^*(v_{k+1})$, selv om den ligger i PQ
- Derved har vi:
  $f(v_{k+1}) = g(v_{k+1})+h(v_{k+1}) = g^*(v_{k+1})+h(v_{k+1}) <= g^*(x)+h(x) <= g(x)+h(x) = f(x)$
- Her må imidlertid alle <= være likheter, ellers ville $f(v_{k+1}) < f(x)$, og da ville ikke x blitt tatt ut av køen før v_{k+1}. Derved er $g^*(x)+h(x) = g(x)+h(x)$ og altså $g^*(x) = g(x)$.

Avslutning av kapittel 14

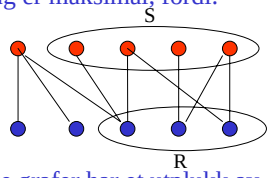
Se på figur 14.10

Den viser sammenhengen mellom matching og flyt i grafer. Vi kan også observere at det å lete etter en flytforbedringsvei i N*f* i figur 14.10 er det samme som å lete etter en forbedringsvei i forhold en tilsvarende matching.

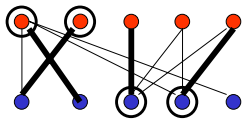
Vi vil lage maks-matching-algoritme for generelle grafer

Det kommer et notat om dette

- Vi trenger et nytt kriterium for å vise at en matching er maksimal, fordi:
- Hall har bare mening for bipartite grafer:



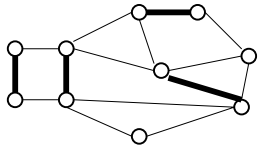
- Kønig-Egervàrys-teorem (fra gruppeøvelsene): Bipartite grafer har et utplukk av noder som
  - Dekker alle kanter
  - og det finnes en matching som er så stor (og som derved er maksimal)

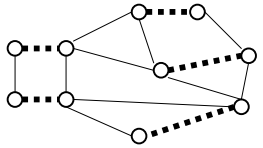


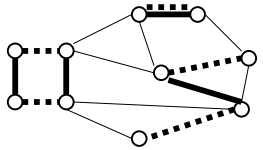
  - Dette gjelder dessverre ikke for ikke-bipartite grafer (grafer med odde løkker):
  - Minste overdekkene utplukk (rødt):  Største matching: 

Nytt kriterium: Forbedringsveier

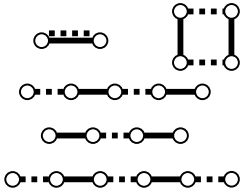
- For generelle grafer: Gitt en matching M. Dersom det finnes en større matching N, så finnes også en forbedrings-vei i M. Bevis:

M 

N 

N og M i samme graf 

Mulige mønstre som inneholder kanter fra M og/eller N:



M-forbedringsvei!

Bare det siste mønstret har flere N-kanter enn M-kanter. Derfor må det finnes minst ett slikt. Og det er altså en forbedringsvei for M!

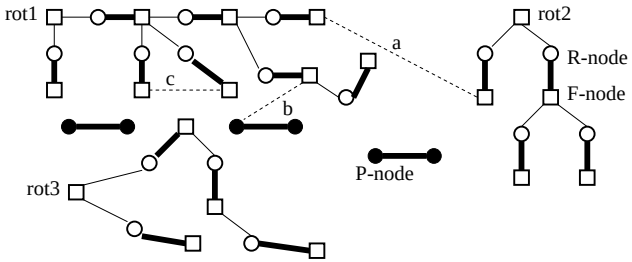
Altså, nytt kriterium: Om en matching ikke har forbedringsveier, så er den maksimal

Nok en maks-matching-algoritme for bipartite grafer

(som lar seg generalisere til generelle grafer)

- Ide: Gro trær på vanlig måte, men fra *alle* umatchede noder.
- Nodene i trærne er F(irkant)-noder og S(irkel)-noder, og alle noder utenfor trærne er P(rikk)-noder. Alle røttene i trærne er F-noder.
- Når vi går fra en rot utover en vei i treet vil annenhver node være F-node og S-node, og alle kanter fra S-node til F-node vil være matching-kanter.
- Initialisering: Alle ikke-matchede kanter settes til F-noder. Resten blir P-noder. Alle P-noder er altså matchede.
- Steget: Se etter kanter fra F-node (vi overser de til R-noder):

- (a): Kanten går til F-node i annet tre: Da er forb.vei funnet
- (b): Kanten til P-node. Utvid treet på vanlig måte
- Kant til F-node i eget tre (c) finnes ikke (gir odde løkke)



Avslutningssituasjonen har ikke forbedringsvei

Anta at algoritmen stopper uten å finne en F- til F-kontakt mellom to trær. Gitt en alternerende vei V, som starter i en rot. Vi vil vise at V aldri komme til en annen rot (og derved ikke kan være en forbedrings-vei). Vi beviser dette gjennom to lemmaer:

Lemma 1: Om V starter i et tre i en rot-node (som er en F-node) eller kommer utenifra og inn i et tre i en R-node, så vil V, så lenge den går mellom noder i dette treet, alltid følge matchede kanter fra en R- til en F-node, og den vil ikke komme til roten av treet.

Bevis:

- Innen *ett og samme tre* kan det ikke gå kanter mellom to F-noder eller to R-noder
- Så lange V er innenfor samme treet må den annenhver gang være i en F-node og en P-node.
- Om V starter i roten av treet, og forblir i treet må den først følge en umatchet kant til en R-node.
- Om V kommer utenfra treet kommer den langs umatchet kant, til en R-node i treet.
- I begge disse tilfellene går den altså langs en umatchet kant til en R-node i treet. Den neste kanten i V må da være matchet, og gå til en F-node, så må V følge en umerket kant til R-node osv.
- Dermed vil V, så lange den forblir i dette treet, følge matchede kanter fra R- til F-noder, og umatchede fra F- til R-noder. Dermed vil den heller ikke kunne komme til roten av treet, for da måtte den følge en umatchet kant til roten, som er en F-node.

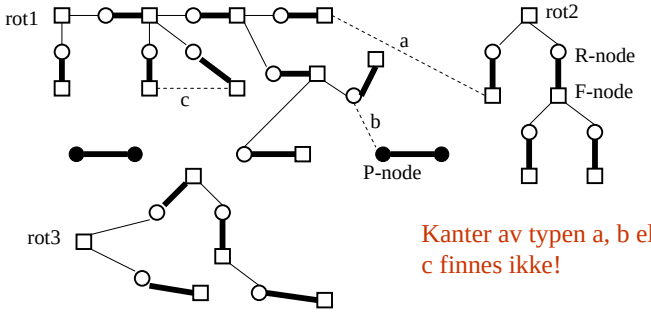
Avslutningssituasjonen har ikke forbedringsvei - II

Lemma 2: Dersom V startet i roten av et tre T eller kom inn i treet T til i R-node, så kan V bare forlate T i en F-node, langs en (umatchet) kant som fører til en R-node i et annet tre.

Bevis:

- Den siste kanten V fulgte i T måtte være en matchet kant.
- Ut fra Lemma 1: Så lenge V er i treet, må den følge matchede kanter fra en R- til en F-node. Dermed vil den altså måtte forlate treet i en F-node.
- Den kommer da til en R-node siden algoritmen stoppet nettopp fordi det ikke var noen kanter fra F-noder i T til F-noder i andre trær, eller fra F-noder til P-noder.

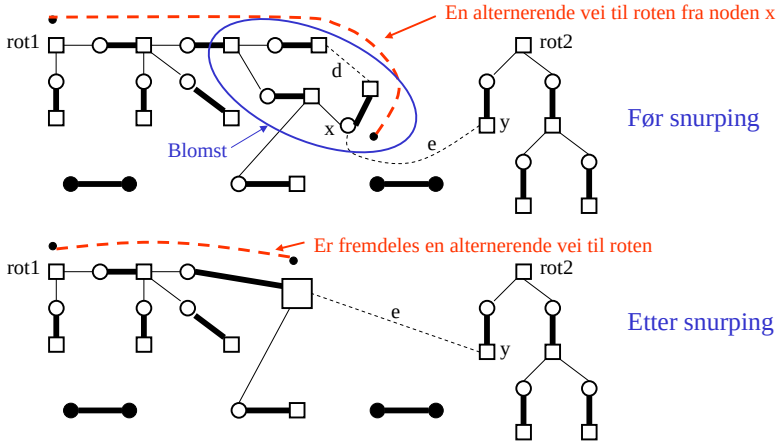
Dermed: Lemma 1 og 2 sier til sammen at V ikke kan nå en rotnode i et annet tre, og altså ikke være en forbedringsvei.



Kanter av typen a, b eller c finnes ikke!

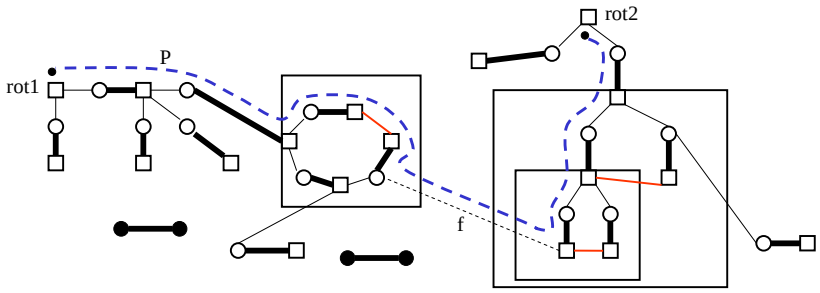
Algoritmen for generelle grafer

- Da kan det finnes kanter (som *d* i figuren) mellom to F-noder i samme tre. Da dannes odde løkker med treet, og disse kalles "blomster".
- Merk at det da går alternerende veier fra *alle* noder i blomsten tilbake til roten
- Dermed vil kanten *e* gi en forbedringsvei fra rot2 til rot1
- Derfor: "Snurp" en blomst sammen til en firkantnode så fort den oppstår.



Forbedringsveier i den opprinnelige grafen

- Det kan etter hvert bli mange blomster, også inne i hverandre
- Det er da viktig at en forbedringsvei som oppstår i den snurpete grafen, også angir en forbedringsvei i den opprinnelige grafen.
- Figuren viser hvordan en slik vei kan finnes:



Ingen forbedringsveier ved stopp

(Se figur neste foil)

- Anta at denne algoritmen stopper uten at det blir funnet noen forbedringsvei.
- Må da vise at ingen forbedringsveier finnes (i den opprinnelige grafen), slik at matchingen altså er maksimal.
- På den sammensnurpede grafen går argumentet som før. Der er ingen forbedringsveier
- Om da alle forbedringsveier V i den opprinnelige grafen også blir forbedringsveier V' i den snurpede grafen, ville vi være i mål.
- Dette igjen ville være riktig dersom V bare kan være innom en gitt blomst én gang, slik at sammensnurpingen bare fører til at seksjoner med partall antall kanter blir skåret ut fra V, og forskjellige avsnitt av V aldri ble snurpet sammen.
- Dette stikker i at Lemma 1 og 2 gjelder for den snurpede grafen, noe som kan vises på nøyaktig samme måte som tidligere.
- Om vi følger V fra en av endene, så vil følgende gjelde:
  - Nå V forlater en blomst går den ikke umiddelbart inn i en annen blomst. Da ville vi hatt en F-til-F-kant i den snurpede grafen (mellom blomster), og dermed ikke stoppet. Når den forlater en blomst går den altså til en S-node i den snurpede grafen.
  - V kan bare gå inn i en gitt blomst én gang, siden den er avhengig av å gjøre det langs den ene matchede kanten inn i blomsten. Her er det viktig at Lemma 1 og 2 gjelder for den snurpede grafen, slik at V bare vil følge matchingkanter i denne fra S- til F-kanter.

Ingen forbedringsveier ved stopp - figur

Figur til beviset (på forrige foil) for at det ikke er forbedringsveier i den opprinnelige grafen når algoritmen stopper uten å finne F-til-F kant mellom trær i den snurpede grafen.