# INF 3/4130
# 19. oktober 2006

- Dagens: Kapittel 10 og 23 i hovedboka
  - Fra kap 10 : Dybde-først og branch-and-bound søk
  - Fra kap 23: A*-søk
- Oblig 2 har ligget ute en stund. Frist 27 oktober.
- Konkurranse:
  - Kommer i løpet av en uke (15-spill?)
  - Frist ca. 20 november
- Forelesning neste uke:
  - Dino Karabeg starter om NP-kompletthet, uavgjørbarhet, etc.
  - NB: På rommet Alfa-Omega i 4. etasje (NR)



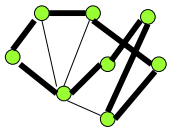
## Søk i tilstandsrom

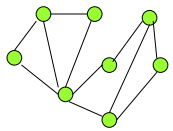
- Kap 10: Herfra skal vi ”minne om” (fra INF 1020)
  - Backtracing algorithms
    - dybde først søk i tilstandsrommet
    - Trenger lite lagerplass
  - Branch and bound (ble kanskje ikke kalt det i INF 1020?)
    - Bredde først søk, med varianter
    - Trenger mye plass: Må holde i lageret alle noder (tilstander) som er ”sett”, men som ikke er studert
    - Varianter: Kan f.eks. 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 til å 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
  - Men om vi setter visse krav til heuristikken, så får vi en algoritme alias Dijkstras ”Korteste vei”-algoritme

## Modeller for avgjørelsessekvenser

- Det er flere måter å ”modellere” avgjørelsessekvenser for et gitt problem
- Gitt modelleringsmåte: De mulige sekvensene danner et tre
- Eksempel, finne mulig Hamiltonian Cycle (innom alle nodene én og bare én gang):

Hamiltonian Cycle



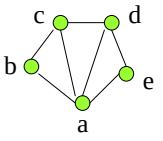


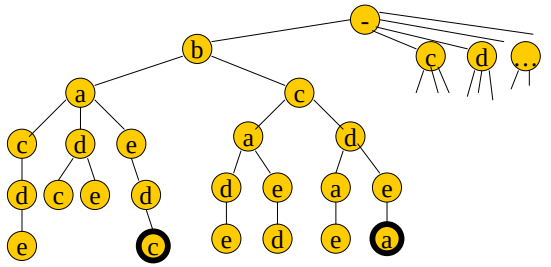
Har opplagt ingen Hamiltonian Cycle

- To måter å modellere dette på:
  - 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.

## Modeller for avgjørelsessekvenser

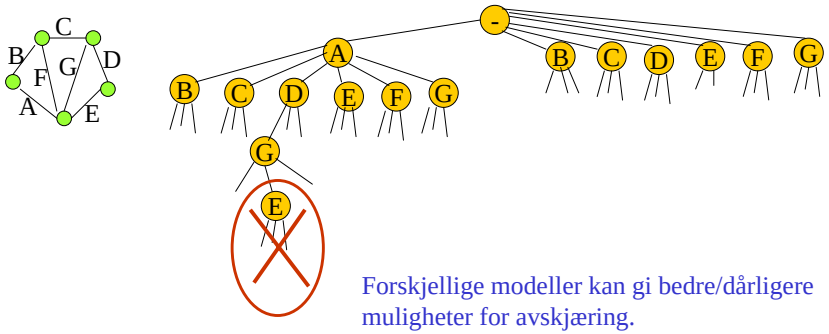
- Tre-struktur ut fra første modell:
  - Velg en node og forleng veien fra denne på alle mulige måter
    - Mulige valg i steget i algoritmen: Alle kanter (ut fra siste vei-node så langt) som ikke går tilbake til allerede brukt node.



# Modeller for avgjørelsessekvenser

- Tilstands-tre ut fra andre modellen:
  - 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

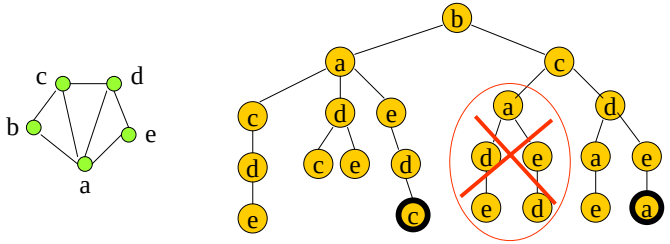



Forskjellige modeller kan gi bedre/dårligere muligheter for avskjæring.

Eksempler fra boka:
Figur 10.3 og 10.4 (Subset sum)
Side 719 (8-spill, liten utgave av 15-spill)

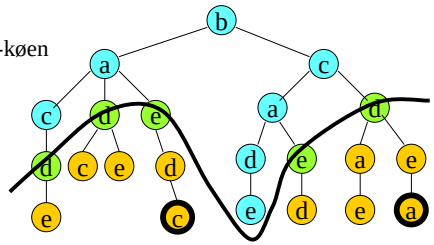
# Dybde-først søk/tilbakesporing

- 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: Holder bare nodene mellom roten og den noden man er i
  - Kan bruke ”heuristikk” til å først 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”. Her: Kan være ”lur”
  - F.eks.: Når man har valgt b, c og a, kan man se at alle ”tilbake-kanter til b er sperret, og at dette ikke kan føre til en Ham. Cycle.



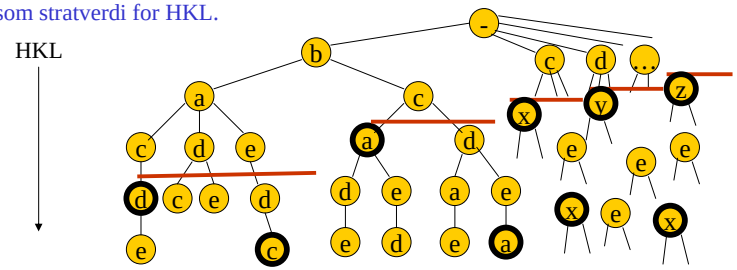

# ”Branch and bound”

- 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)
  - Alle over (nærmere roten) er man ferdig med (blå)
  - Alle under er ikke sett enda (gule)
- 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 (hvor lovende er noden)
    - Likner mye på A*-søk (kommer)
- Kan selvfølgelig også bruke tradisjonelle avskjæring (bounding, pruning)




# Søk etter ”beste løsning”, idé 1

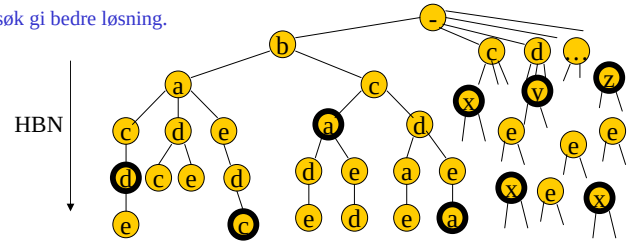
- For enkelhets skyld: Beste løsning er den mål-noden som er nærmest roten, regnet i antall kanter (men lar seg lett generalisere)
  - Da må vi *ikke* tenke på Ham. Cycle som eksempel (alle mål-noder like dype)
- Idé 1:
  - Bruk dybde først, og bruk all vanlig avskjæring slik som før
  - Hold en global variabel ”hittil korteste lengde”: HKL
  - Gå aldri dypere en det beste vi har sett til nå, se tegning
  - Om vi også kan beregne et minstemål for hvor langt det er til nærmeste mål-node får vi enda bedre avskjæring
  - Kan ofte *på forhånd* beregne en øvre grense for hvor langt unna beste målnode er. Settes som stratverdi for HKL.



# Søk etter ”beste løsning”, idé 2

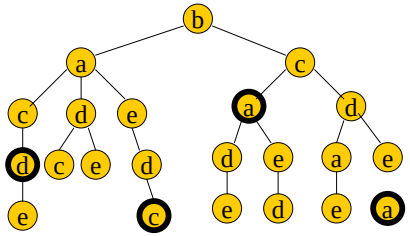
Litt forandret fra forelesningen

- SOM FØR: Beste løsning er den mål-noden som er nærmest roten, regnet i antall kanter.
- Idé 2: Bruk ren bredde først.
  - Dette vil helt rett fram gi beste løsning uten noen gang å gå for dypt
- Alternativ, om optimalitetskriteriet er litt mer komplisert enn bare ”nærmest mulig roten”: Bruk prioritert bredde-først søk.
  - Prioritet: Estimat for hvor sannsynlig det er at beste node ligger i ”mitt” subtre.
  - Også her: Hold en global variabel ”hittil beste node”: HBN
  - Om du kommer til en node *der det er sikkert at ingen i hele nodens subtre er bedre enn HBN*, så skjær av.
  - MEN: Man må generelt fortsette søket inntil *alle* grener er avskåret som beskrevet i forrige punkt, altså til prioriteskøen av ”sette, men ikke behandlede” noder er tom.
  - Her vil A*-søk gi bedre løsning.




# Iterativ bredde først (ikke i boka)

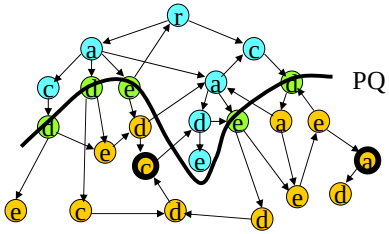
- Et alternativ om man vil gjøre rent bredde-først-søk
  - Bruker like lite plass som dybde først søk
  - Men må gjøre litt arbeid om igjen
- Idé: Gjør dybde-først-søk til nivå 1, så et helt nytt søk til nivå 2, osv.
  - Om det er stor forgreningsfaktor tar ikke dette så mye mer tid enn vanlig bredde-først
  - Og det krever altså mye mindre plass
- Kan også brukes med prioriteter/heuristikker etc.



# A*-søk

Eller: Dijkstras *korteste vei algoritme*, med heuristikk!

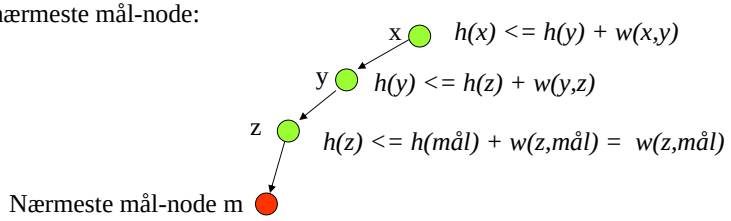
- A*-søk egner seg for problemer der vi har
  - en (eksplisitt eller implisitt) graf av ”tilstander”,
  - Med en start-tilstand, og et antall mål-tilstander
  - Mulige tilstands-overganger (rettede kanter) med en gitt ”kost”.
  - Og: Skal finne en vei fra start til en mål-tilstand med, med minimal kost.
  - Altså logisk sett: korteste-vei-problemet
  - Se side 728/29, figurer 23.8 – 23.10.
    Her er *h(x)* = avstanden i rett linje




- Strategien er et bredde-først-søk, med heuristikk
  - 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
  - Men: Må ha spesielle krav på h(x) for at algoritmen skal bli like enkel som Dijkstra
- Finnes varianter til fullt A*-søk (A-søk, A*-søk uten alle *h(x)*-krav oppfylt)

# Krav til *h(x)*

- Krav (monotonitet):
  1. Heuristikk-funkjsonen *h(x)* er mindre-eller-lik lengden av faktisk korteste vei fra x til nærmeste mål-node
  2. Om det er kant fra x til y med vekt *w(x,y)*, så skal gjelde: *h(x) <= h(y)+w(x,y)*
  3. Alle mål-noder *m* har *h(m) = 0, og ellers må vi ha h(m)>= 0.*
- Om *h(x)* alltid er 0, så er disse oppfylt. Da får vi Dijkstras algoritme.
- Det fine er at krav 2 og 3 medfører krav 1, så vi slipper å tenke på krav 1.
- Bevis: Vi antar at *x -> y -> z -> mål-node* er korteste vei fra x til nærmeste mål-node:




Kombinerer vi disse får vi: *h(x) <= w(x,y) + w(y,z) + w(z, m)*
Altså: *h(x)* <= korteste vei til nærmeste målnode.

## Om A-søk og varianter av A*-søk

- Om du har en heuristikk *h(v)* for hvor langt det er til en mål-node
  - Og *h(v)* kan være både litt for stor og litt for liten
  - Da kalles dette A-søk (veldig likt prioritert bredde-først-søk)
- Om vi vet at *h(x)* aldri vil være større enn den virkelig korteste vei til målet
  - Men *ikke* tilfredstiller det *fulle* monotoniteskrav
  - 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!
- Her er boka dessverre ikke helt god her (se trykkfeil-listen).
  - Vi tar derfor ikke dette med som pensum

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

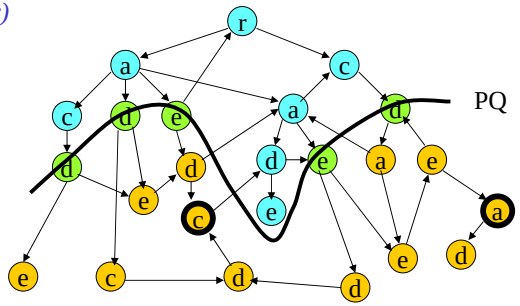
- Vi har en rettet graf G med kantvekter *w(x,y)*, en startnode og et antall mål-noder, samt en monoton heuristikk-funksjon *h(x)*.
- Hver node x har i tillegg følgende variable:
  - *g(x)* = foreløpig korteste vei fra startnoden. Denne vil stadig forandre seg under algoritmen, men vil til slutt få lengden 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)

## Figur for A* algoritmen

Tre-noder: Blå
PQ-noder: Grønne
Usette noder: Gule

Alle noder *v* har:

*h(v)*: En fast heuristikk-verdi for hver node
*g(v)*: Foreløpig korteste vei fra r (forandrer seg)
foreldrepeker: Viser vei tilbake til r, ut fra nåværende *g*-verdi)
*f(v)*: *g(v) + h(v)*




Alle noder med pekere fra blå noder, må være grønne eller blå

## Selve algoritmen

- 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 ut fra PQ (med minst *f*-verdi)
  - Dersom x er en mål-node, slutter herved algoritmen
    g(x) og parent(x) angir da korteste vei fra startnoden og den aktuelle veien (baklengs).
  - Ta ny miste ut av PQ, kall den x, og la den bli en tre-node
    Den har allede nå sin foreldre-peker og *g(x)* satt riktig, bevis kommer
  - 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 *h(x)* = 0 for alle noder, så blir dette altså Dijkstras korteste-vei-algoritme

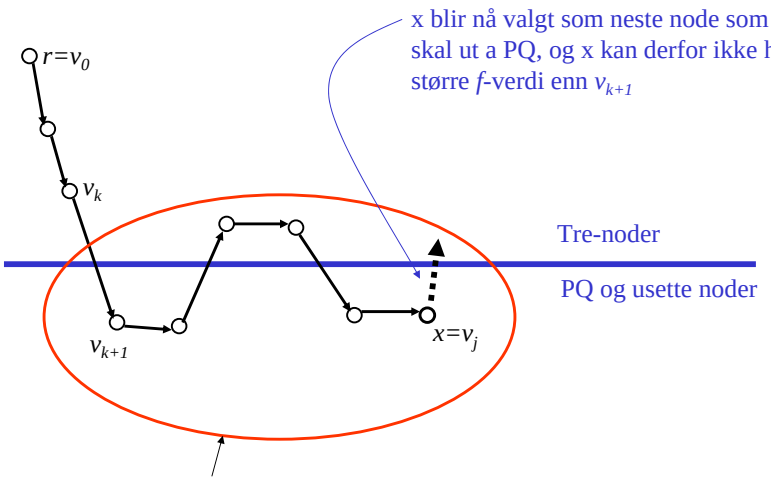
## A*-søk går "uten tilbakelegging" med monoton *h(x)*

- 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)* …

## Figur til beviset for at noder har fått riktig *g*-verdi når de taes ut av PQ




Vi viser essensielt at alle disse må ha samme *f*-verdi, og dermed også samme *g*-verdi

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, …, j-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, …, 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 $g^*(x) = g(x)$.

## Eksempler på A*-søk (kopiert opp på trad. foil)

- Eksempel i kap. 23.3.2: Finn korteste løsning i 8-spill
- Figurer side 719 og 727