INF 4130
15. oktober 2008

- Dagens temaer: 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 24 oktober.
  - Det er lov å diskutere den med gruppelærer!
- Forelesning neste uke:
  - Spilltrær, alfa-beta-avskjæring (kap 23.5), samt om sjakkprogrammer
  - Ved: **Rune Djurhuus, STORMESTER i sjakk**, med egen sjakkspalte i Aftenposten, og med Mastergrad fra Ifi.
  - Også folk som ikke er student på dette kurset innviteres.
- Gruppene neste uke
  - Vanlige oppgaver blir lagt ut, men det settes også av litt tid til spørsmål og diskusjon omkring Oblig 2.

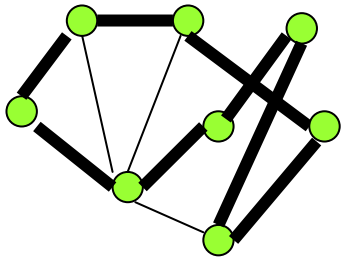
Søk i tilstandsrom - oversikt

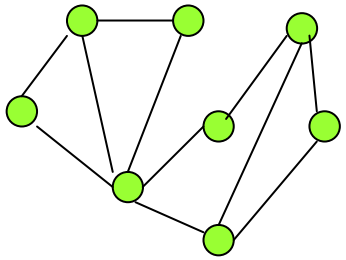
- Kap 10: Herfra skal vi ”minne om” (fra INF 2220: Alg. og datastr.)
  - Backtracing algorithms
    - dybde først søk i tilstandsrommet
    - Trenger lite lagerplass
  - Branch and bound (ble kanskje ikke kalt det i INF 2220?)
    - Bredde først søk, med varianter
    - Trenger mye plass: Må holde i 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, ikke pensum):
    - 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: 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, men som virker raskere.

Modeller for valgsekvenser under søk

- Det er flere måter å ”modellere” valgsekvenser 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



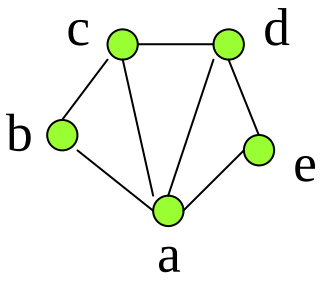


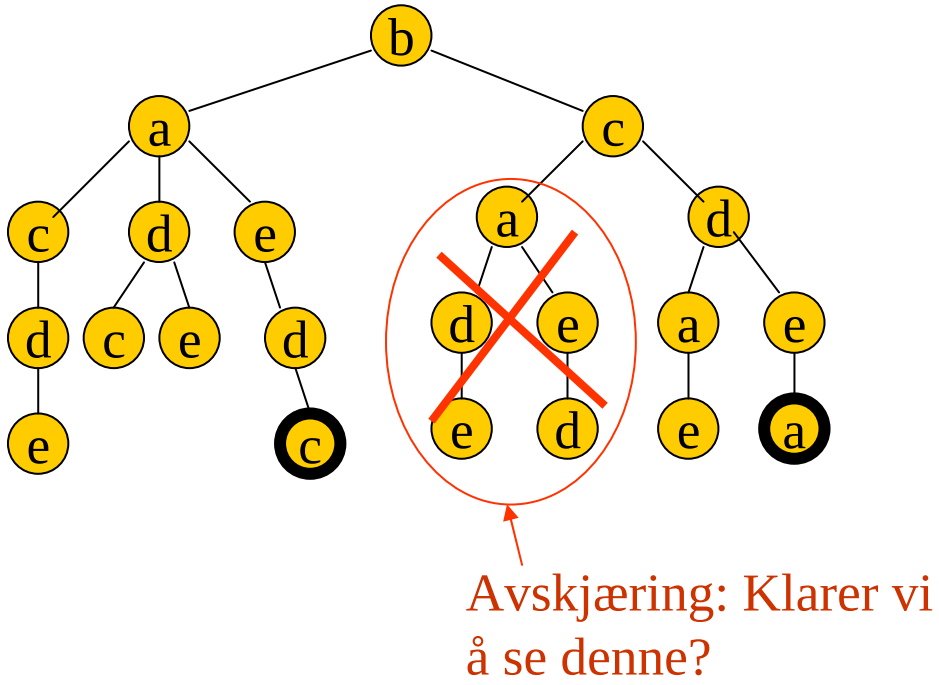
Har opplagt ingen Hamiltonian Cycle

- To måter å her modellere søk 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 valgsekvenser

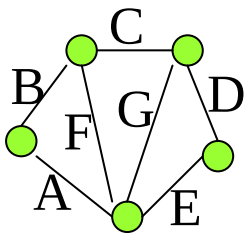
- Tre-struktur ut fra første modell:
  - Velg en node og forleng veien fra denne på alle mulige måter. men ikke bruk kanter (ut fra siste vei-node så langt) som ikke går tilbake til allerede brukt node.

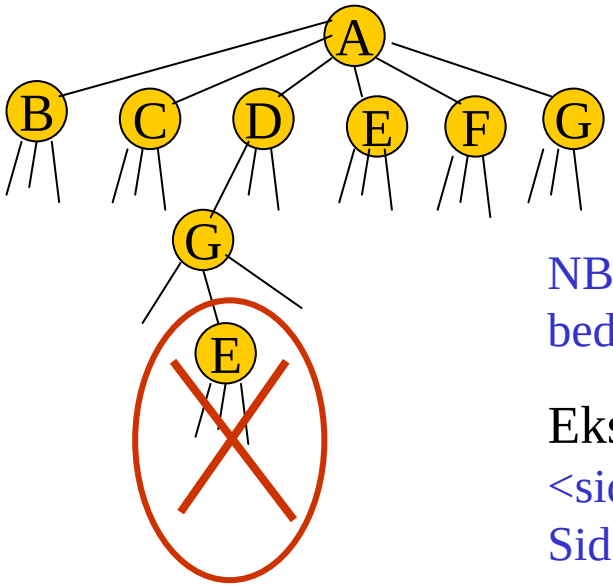




Modeller for valgsekvenser

- 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
    - Avskjæring: Ikke se på kanter som gjør at alle de sammenhengende komponentene av valgte kanter fremdeles forblir enkle veier





NB: Forskjellige modeller kan gi bedre/dårligere muligheter for avskjæring.

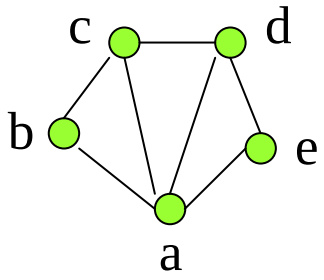
Eksempler fra boka:

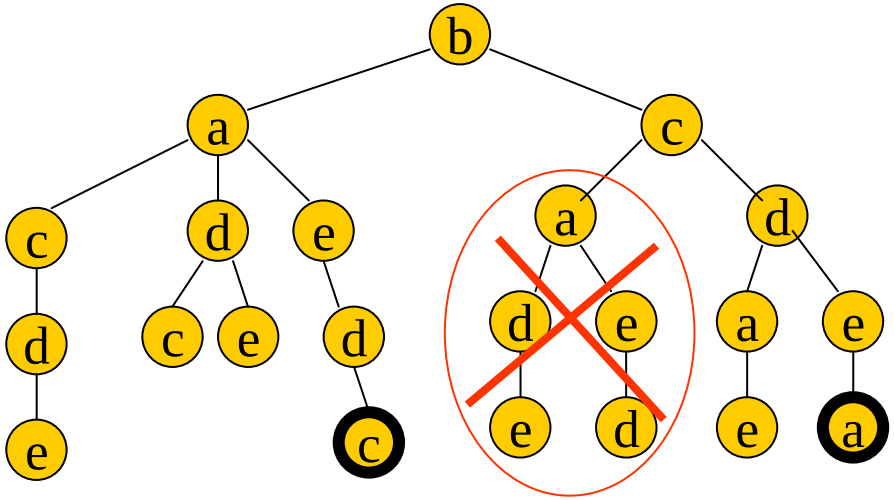
<side 293, figur 10.3 og 10.4 (Subset sum)

Side 719 (8-spill, liten utgave av 15-spill)

Dybde-først søk/tilbakesporing

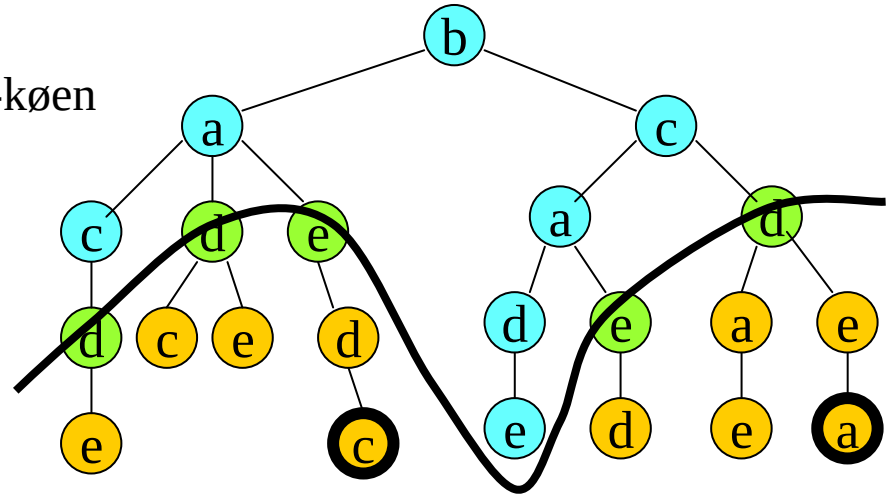
- Gjennomsøker tilstandsroms-treet dybde-først, til vi kommer til en ”mål-tilstand”
  - Bruker greiest en rekursiv prosedyre, som har selve problemstillingen som ”globale data”.
  - Tar liten plass: Holder bare nodene mellom roten og den noden man er i
  - Er ofte i god posisjon til å gjøre avskjæring: 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.





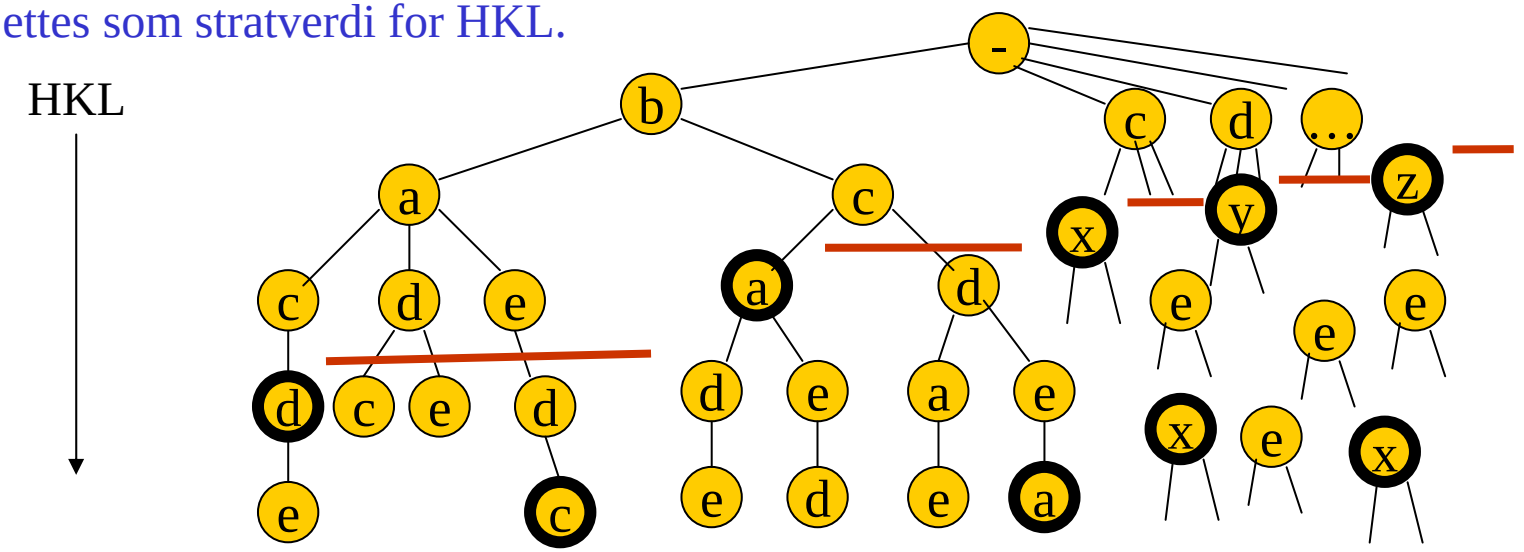
”Branch and bound”

- 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)
- Må selvfølgelig også bruke all mulig avskjæring



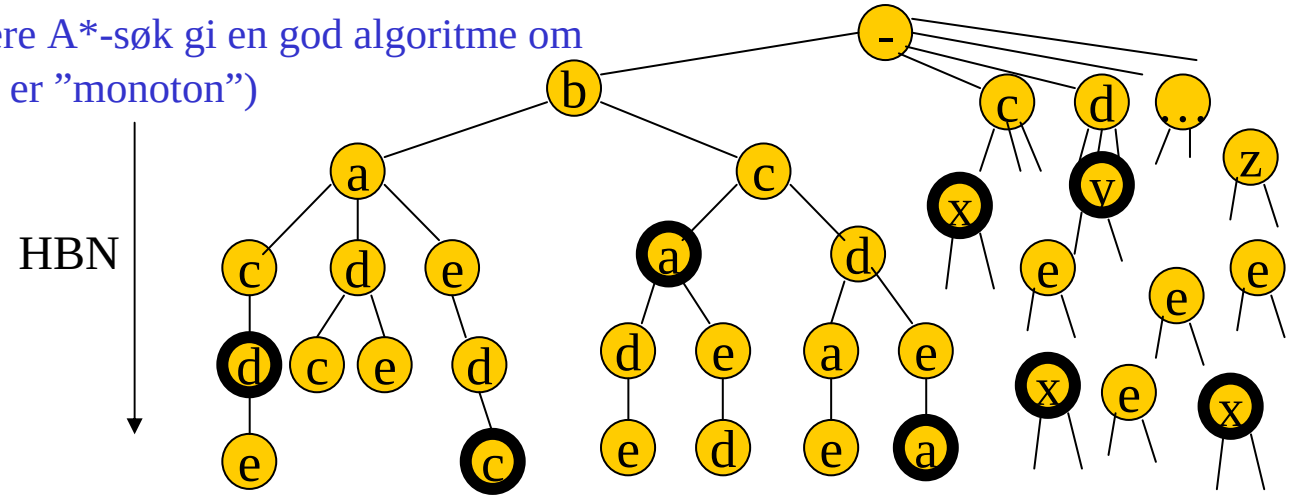
Søk etter ”beste løsning”, idé 1

- 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 kan være. Settes som stratverdi for HKL.



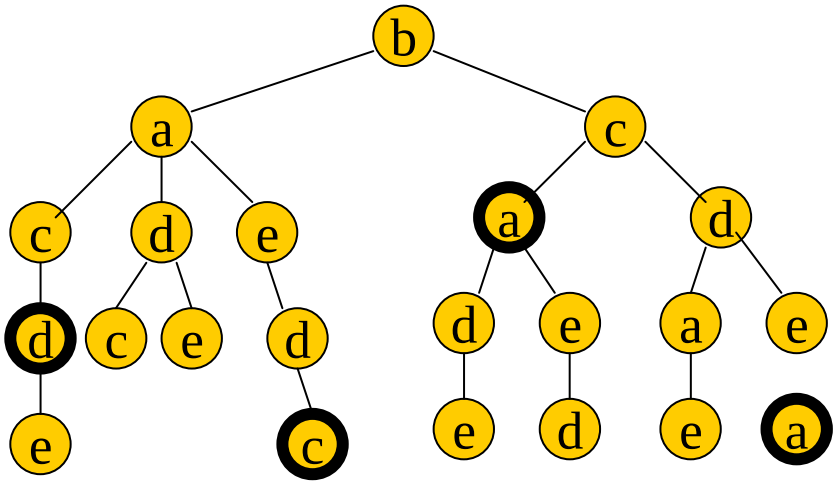
Søk etter ”beste løsning”, idé 2

- 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 senere A*-søk gi en god algoritme om heuristikken er ”monoton”)



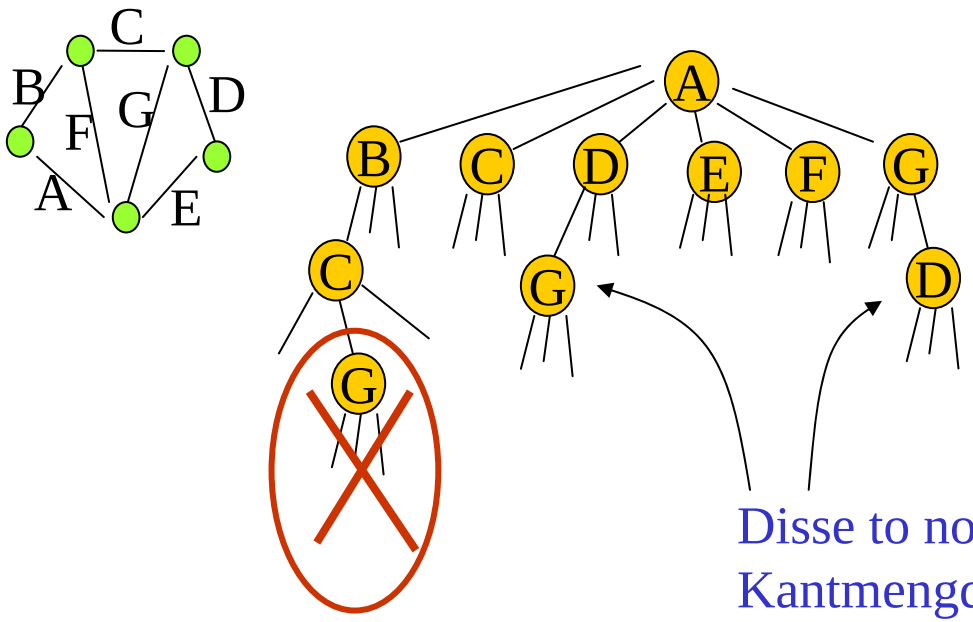
Iterativ bredde først (ikke i boka, ikke pensum)

- 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.



Vi kan se ting som trær eller grafer

- Tilstands-treet kan ofte ha noder som representerer samme tilstand, selv om man er kommet dit ved forskjellige valgsekvenser, se eksempel under.
  - Man kan oftest velge om man vil slå disse sammen eller ikke
  - Da må tiden til å gjøre ting om igjen veies opp mot det å undersøke om man har sett denne tilstanden tidligere.
- Ved *dfs* kan det være vanskelig å slå sammen. Trenger mye mer plass.
- Ved *bfs* vil det som regel være fordelaktig å slå sammen. Sparer plass (og tid?)
- Noen algoritmer krever at man slår sammen, for eksempel Dijkstra og A*

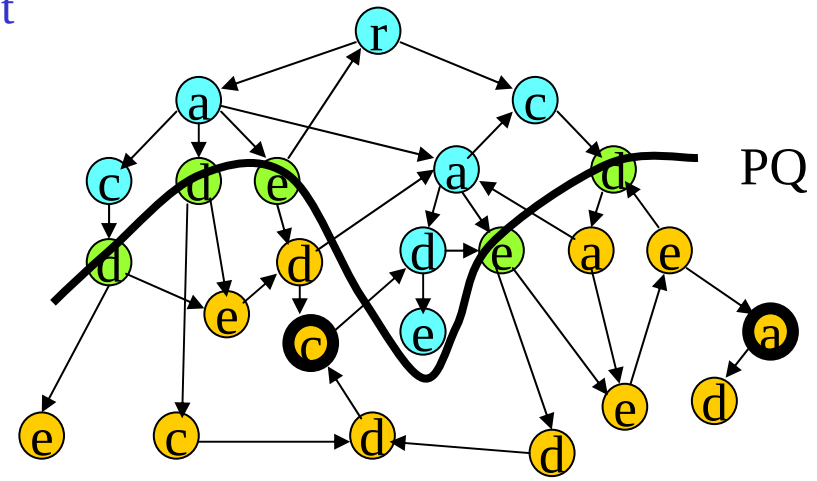


Disse to nodene er essensielt like.
Kantmengden er A, D og G.

A*-søk

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

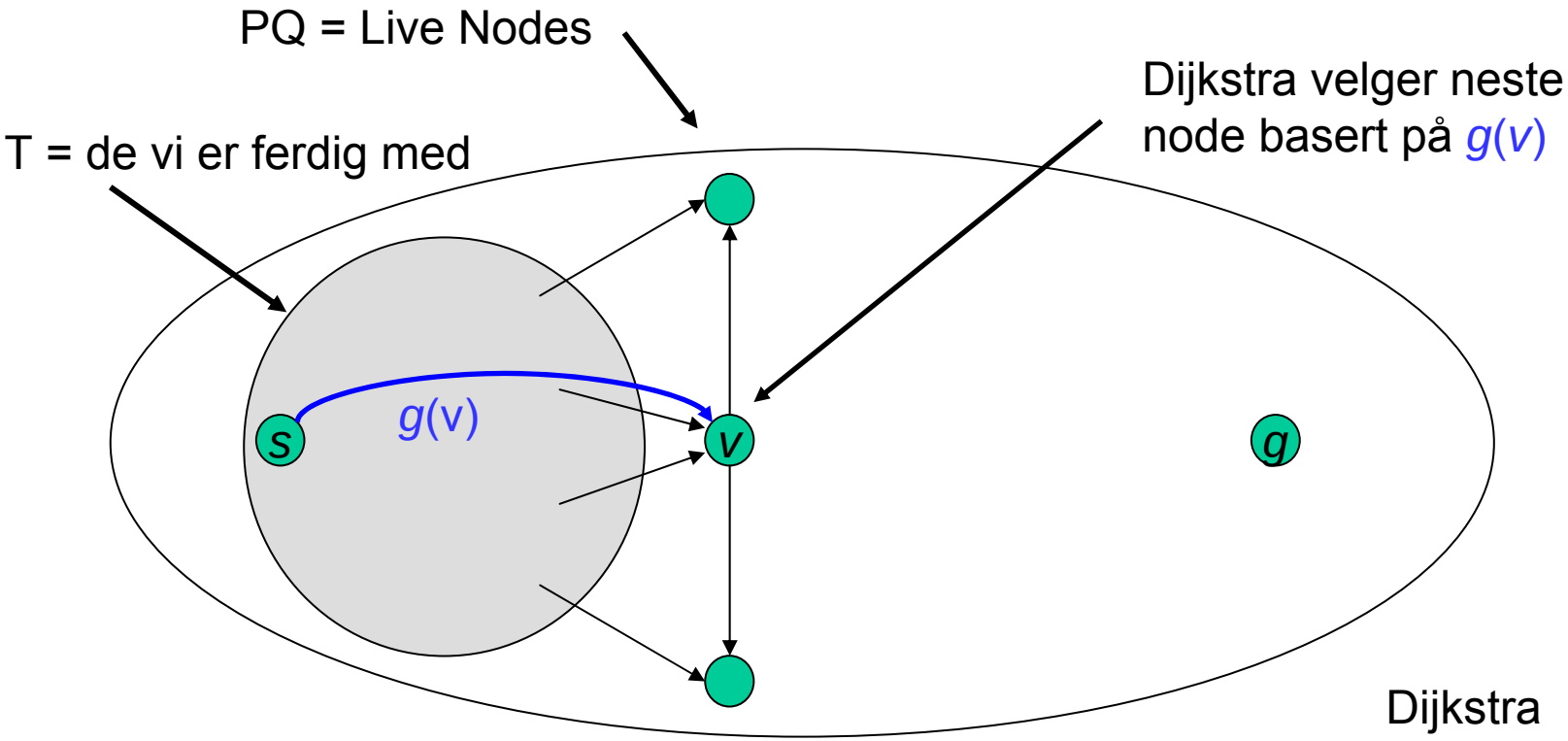
- A*-søk egner seg for problemer der vi har
  - en (eksplisitt eller implisitt) graf av ”tilstander”,
  - Med en start-tilstand, og et antall mål-tilstander
  - Mulige tilstands-overganger (rettede kanter) med en gitt ”kost”.
  - Og: Skal finne en vei fra start til en mål-tilstand, med minimal kost.
  - Altså logisk sett: korteste-vei-problemet



- 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)

Dijkstra – algoritmen

- Strategien er et bredde-først-søk med prioriteskø: PQ = *"Live Nodes"*
- Dijkstras korteste vei algoritme. (Dijkstra er et spesialtilfelle av A*-søk.)
- Dijkstra-algoritmen bøhver aldri se om igjen på noder den har lagt bak seg



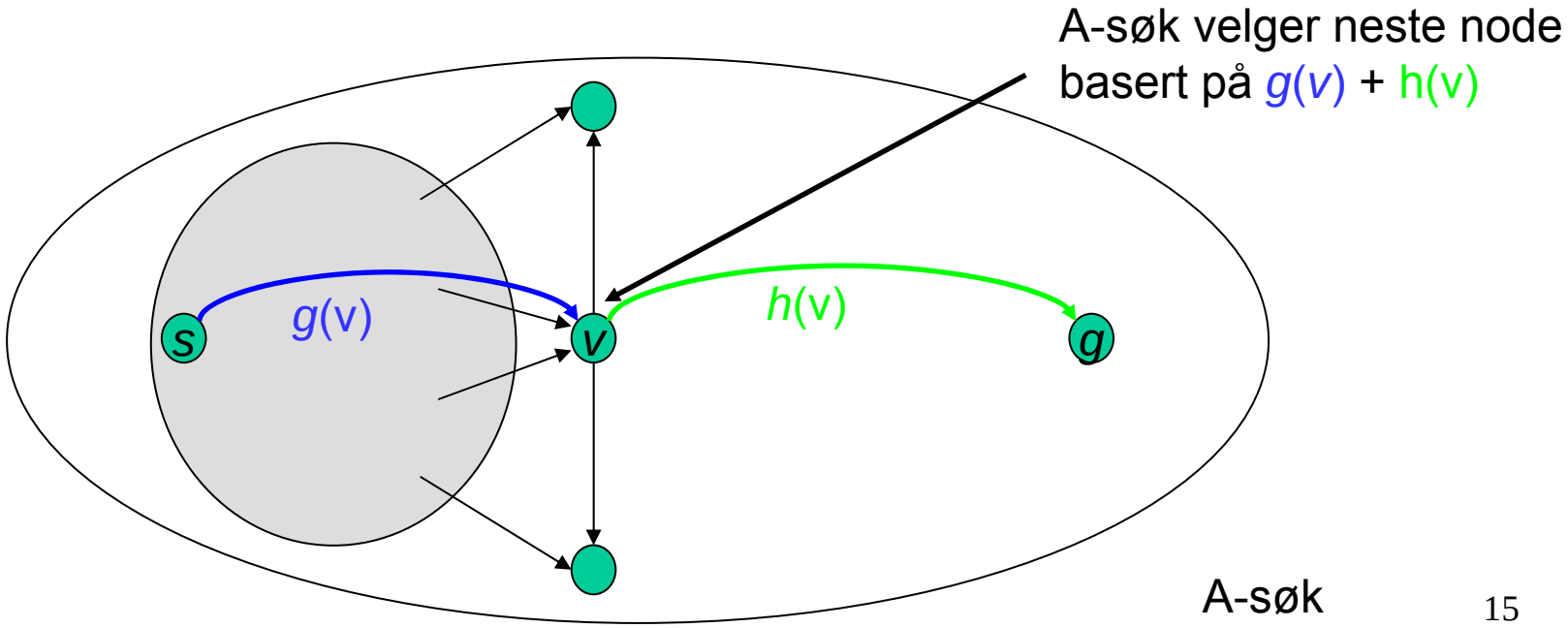
Dijkstras algoritme, korteste vei til alle noder

```
proc Dijkstra(Graph G, Node source)
for each vertex v in Graph do          // Initialisering
   v.dist := ∞                         // Ukjent avstand initielt mellom v og source
   v.previous := NIL                   // Peker for å huske stien
od
source.dist := 0                       // Avstand fra source til seg selv,
PQ := { source }                       // Priorites-køen
T := {  }                              // T settes tom (de ferdigbehandlede nodene)
while PQ is not empty do
  u := extract_min(PQ)                 // Nærmeste node fra prioritetskø, source første
  T :=  T union { u }
  for each neighbor v of u do          // gang. Key i køen er dist-verdien
    if not v in T then                 // Sjekker om den allerede er behandlet ferdig
      v.alt = u.dist + length(u, v)
      if v.alt < v.dist then           // Sjekker om vi nå finner kortere vei
        v.dist := v.alt                // I så fall, sett inn denne
        v.previous := u
      fi
   od
od
```

A*-søk – heuristikk mot *en bestemt* node

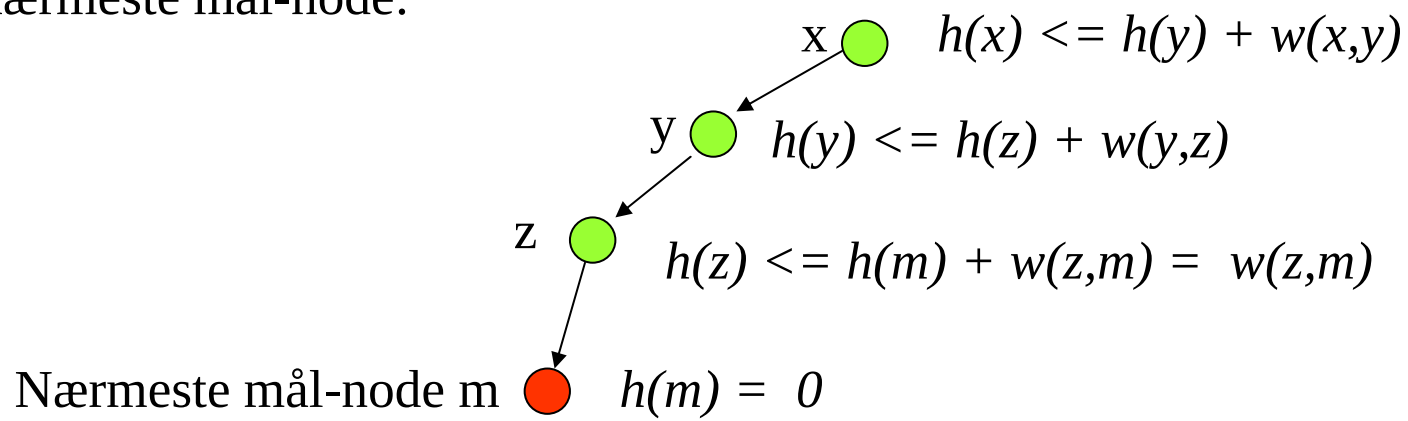
- Strategien er et bredde-først-søk
  - Vi bruker en heuristikk-funksjon *h(v)* for stadig å velge mest lovende vei. Det er denne som hjelper oss å «se» lengre enn et lokalt nabolag.
  - Altså bredde-først-søk med en prioriteskø for valg av neste fra *LiveNodes*.

- Minner derved mye om Dijkstras korteste vei-algoritme. (Dijkstra er faktisk et spesialtilfelle av A*-søk.)



Krav til *h(x) (monotonitet)*

- 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 krav 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)

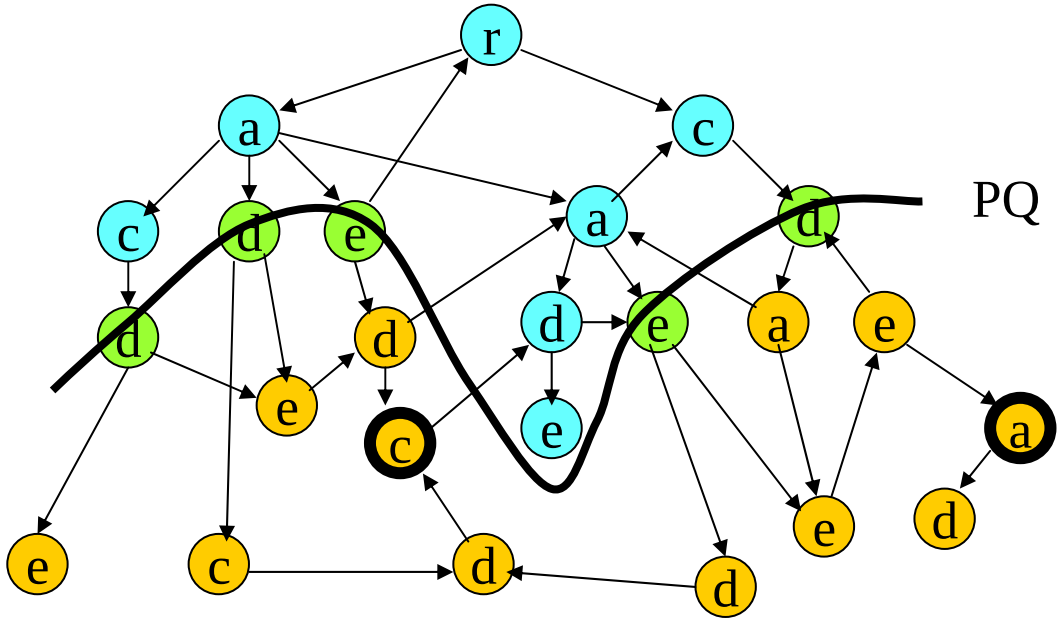
- De nodene som for øyeblikket ikke er i PQ deles i to typer
  - Tre-noder: Disse har en foreldre-peker i et tre med startnoden som rot (korteste vei til roten-treet). Disse har alle vært i PQ, og ved starten er det ingen slike tre-noder.
  - Usette noder (de vi ikke har kommet borti så langt)

Figur for A* algoritmen

Tre-noder (mengden T): Blå
PQ-noder: Grønne
Usette noder: Gule

Alle noder *v* har:

h(v): En fast heuristikk-verdi for hver node. Må være monoton
g(v): Foreløpig korteste vei fra r. Forandrer seg, men ikke etter at den er blitt tre-node (blå)
foreldrepeker: Viser vei tilbake til r, ut fra nåværende *g*-verdi)
f(v): *g(v)* + *h(v)*. Forandrer seg heller ikke etter at den er blitt blå.



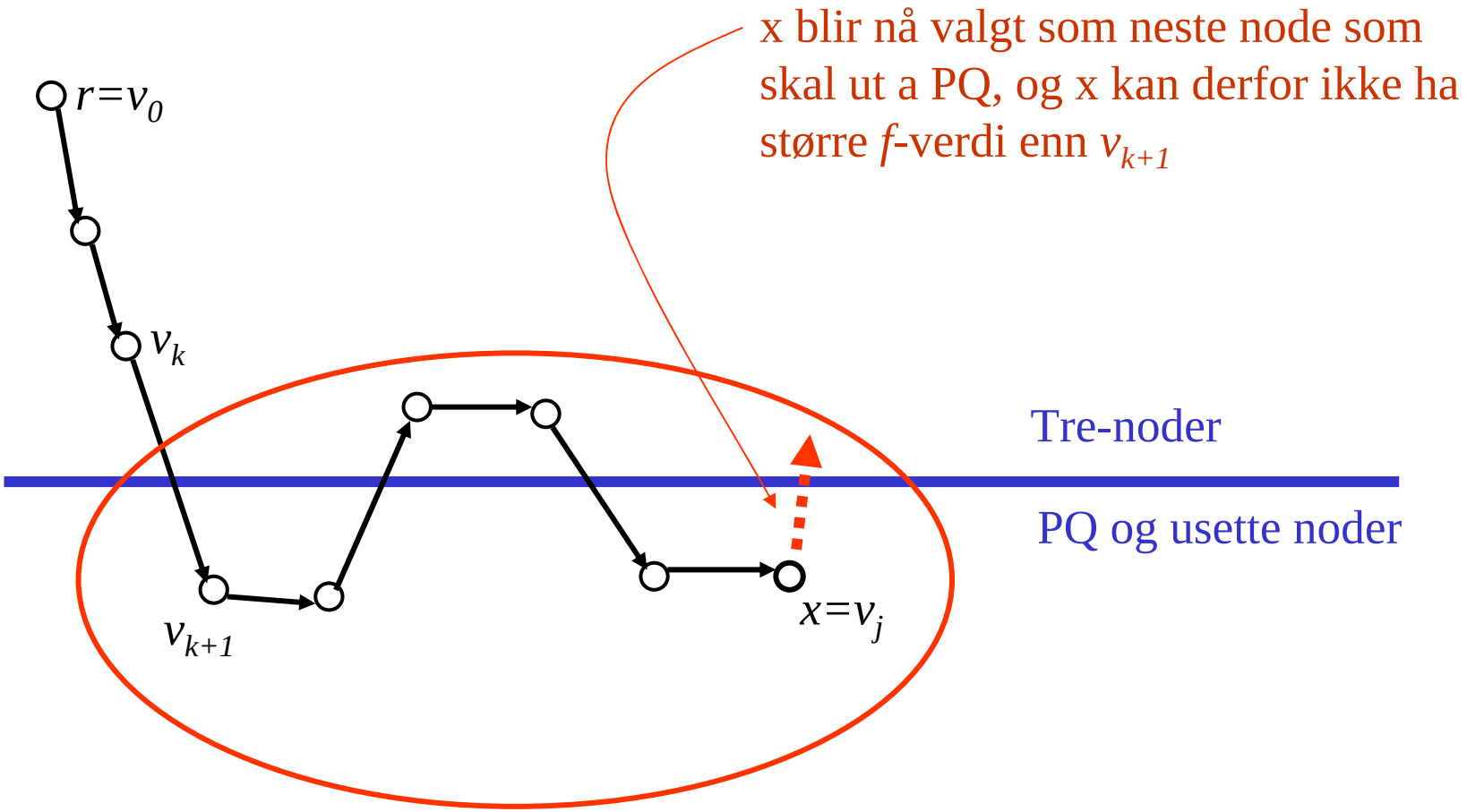
Selve A*-algoritmen

- PQ initialiseres altså med bare start-noden, med $g(s) = 0$ (alle andre er usette)
- Steget, som gjentas inntil PQ er tom :
  - Plukk den best prioriterte noden x ut fra PQ (med minst *f*-verdi)
  - Dersom x er en mål-node, slutter herved algoritmen
    - $g(x)$ angir da lengden av korteste vei fra startnoden
    - $x, parent(x), parent(parent(x)), \ldots$ er den aktuelle veien (baklengs).
  - Ta ny minste ut av PQ, kall den x, og sett den inn i T (tre-nodene vi er ferdige med)
    Den har allede nå sin foreldre-peker og $g(x)$ satt riktig, **bevis kommer**
  - Se på alle naboer til x **utenom de i T** (de ferdigbehandlede), og for hver slik y:
    Dersom $g(y) > g(x) + w(x,y)$ så sett $g(y) = g(x) + w(x,y)$ og $parent(y) = x$
  - 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 noen 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

Vi må vise (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 T. Vi fører bare beviset for *g(x)*
- 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



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 (altså 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 (og derfor gjelder $f(v_{k+1}) >= f(x)$)
- Ut fra monotoniteten vet vi (for alle $i = 0, 1, \ldots, j-1$, *altså helt fram til x*)

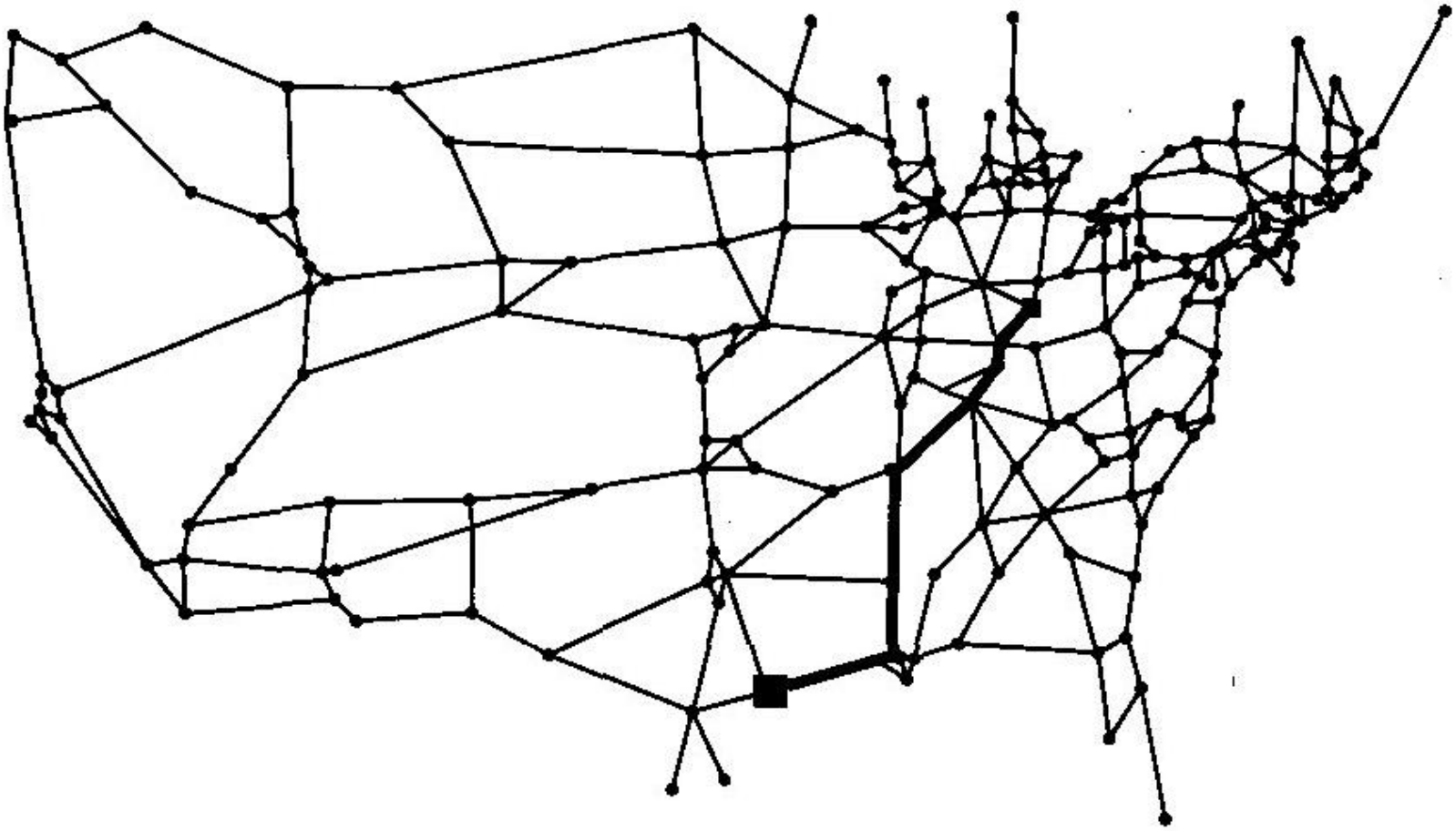
  $$g^*(v_i) + h(v_i) <= g^*(v_i) + h(v_{i+1}) + w(v_i, v_{i+1})$$
- Siden kanten fra v_i til v_{i+1} er med i en korteste vei til v_{i+1}, gjelder ($i = 0, 1, \ldots, j-1$)

  $$g^*(v_{i+1}) = g^*(v_i) + w(v_i, v_{i+1})$$
- 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* etter tur være $k+1, k+2, \ldots, j-1$, og vi setter sammen ulikhetene:

  $$g^*(v_{k+1}) + h(v_{k+1}) <= g^*(v_j) + h(v_j) = g^*(x) + h(x)$$
- 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 ikke ligger i T
- Derved har vi:

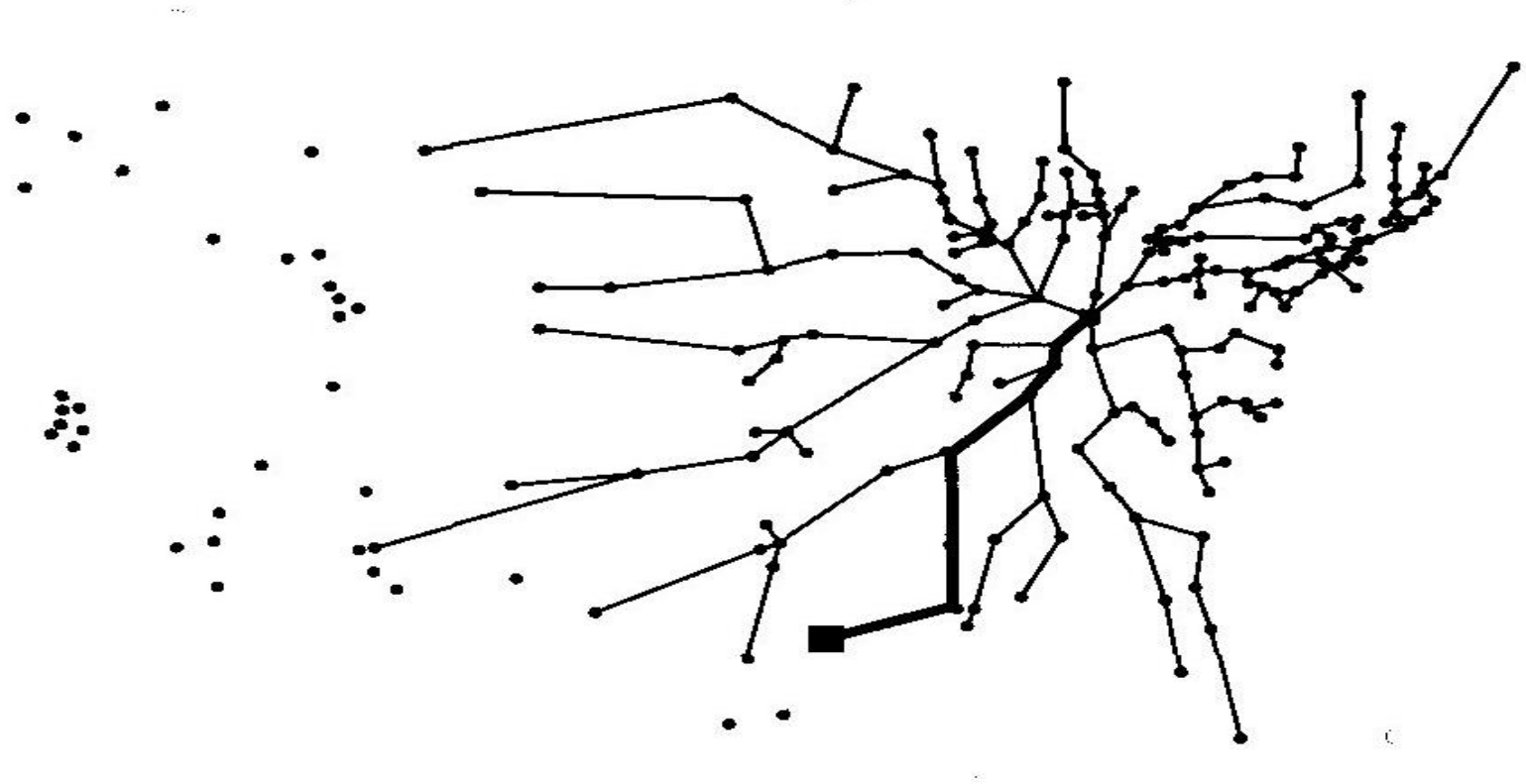
  $$f(v_{k+1}) = g(v_{k+1})+h(v_{k+1}) = g^*(v_{k+1})+h(v_{k+1}) <= g^*(x)+h(x) <= g(x)+h(x) = f(x)$$
- Her må imidlertid alle <= være likheter, ellers ville $f(v_{k+1}) < f(x)$, og da ville ikke x blitt tatt ut av køen før v_{k+1}. Derved er $g^*(x)+h(x) = g(x)+h(x)$ og altså $g^*(x) = g(x)$.

Eksempel på A*-søk



Amerikanske highways, korteste vei Cincinatti - Houston uthevet.

Treet generert av Dijkstras algoritme (stopper i Houston).



Treet generert av A*-søk.