

Søk i tilstandsrom

- *Backtracking* (Kap. 10)
 - DFS i tilstandsrommet.
 - Trenger lite lagerplass.
- *Branch-and-bound* (Kap. 10)
 - BFS
 - Trenger mye plass: må lagre alle noder som er «sett» men ikke studert.
 - Kan også gi hver node en «lovende-het», og så gå videre med den som er mest lovende (heuristikk), bruker prioritetskø, likner på Dijkstras algoritme.
- Iterativ fordypning
 - DFS ned til nivå 1, så til nivå 2, osv.
 - Kombinerer plasseffektiviteten til dybde først og komplettheten til bredde først.
- Dijkstras korteste sti-algoritme
- A*-søk (Kap. 23)
 - Likner mye på branch-and-bound med prioritetskø.

Tilstandsrom og avgjørelsessekvenser

- Et *tilstandsrom* er mengden av alle mulige tilstander det vi studerer (problemet) kan befinne seg i – *problem states*.
- Noen tilstander er lovlige løsninger eller måltilstander – *goal states*.

Søkealgoritmene vi skal se på trenger litt struktur på dette tilstandsrommet for å virke. Algoritmene anvendes på problemer hvor løsningen kan sees på som en serie av avgjørelser som taes. (Avgjørelsene definerer en sti fra start til mål.)

- Det er mange måter å modellere avgjørelsessekvenser for et gitt problem. Disse fører til forskjellige tilstandsromstrær (*state space trees*).

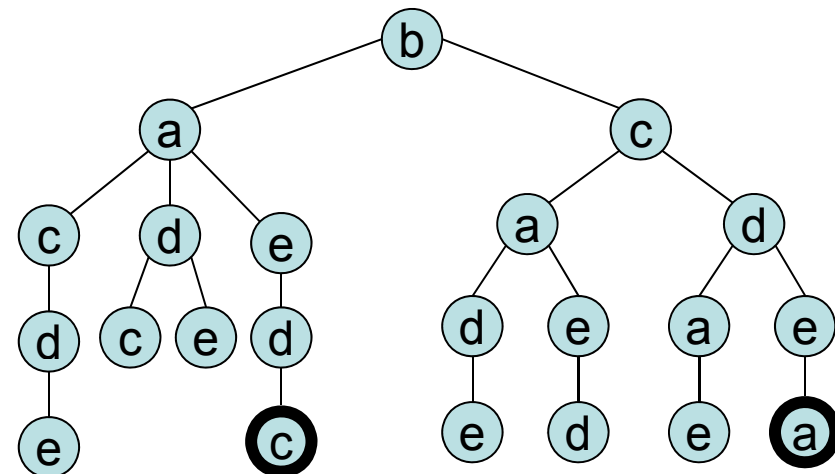
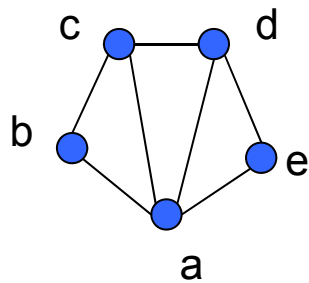
Både backtracking og branch-and-bound baserer seg på søk i disse tilstandsromstrærne.

- Uheldigvis kan tilstandsromstrærne bli store. (Eksponensielt store i forhold til lengden av input.)

Tilstandsrom og avgjørelsessekvenser

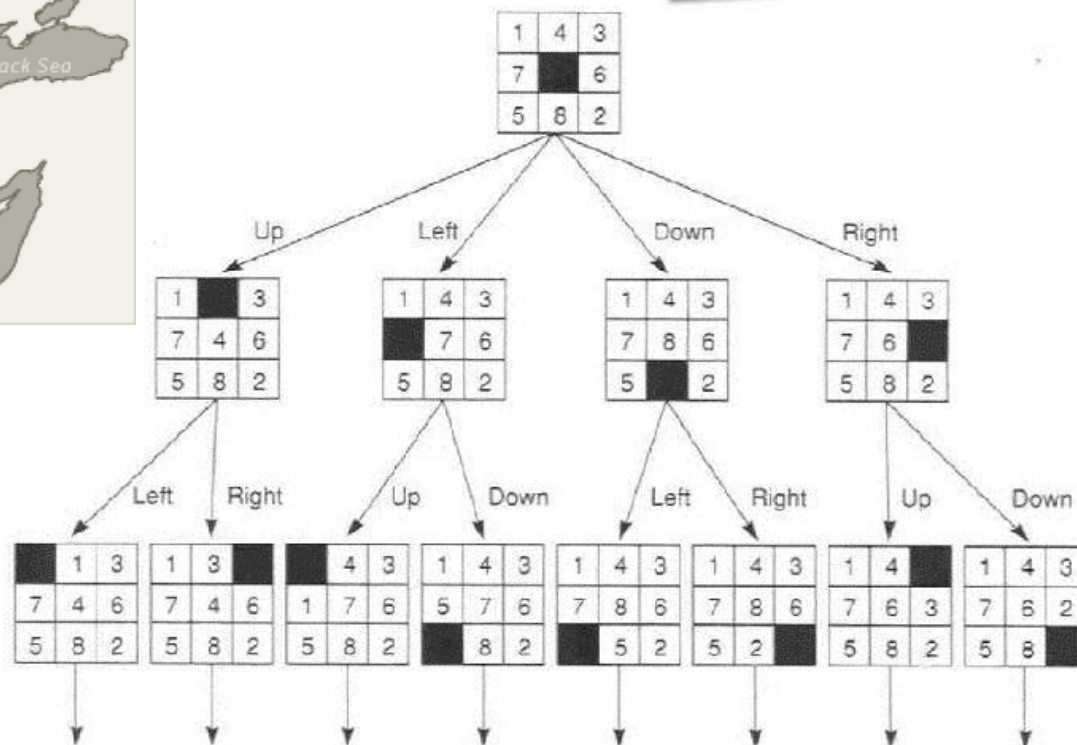
Eksempel: Hamiltonsk sykel

1. Start sykelen i tilfeldig valgt node og gå videre på alle mulige måter.
 - Mulige valg i steget i algoritmen: alle kanter ut fra siste valgte node som ikke går tilbake til allerede valgte node.
2. Start med en kant og legg stadig til en kant til.
 - Mulige valg i steget i algoritmen: alle ikke-valgte kanter som gjør at alle de sammenhengende komponentene av valgte kanter fremdeles forblir enkle veier.



Tilstandsromstre for modell 1 over.

Tilstandsrom og avgjørelseseske



Backtracking / DFS

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

DFS

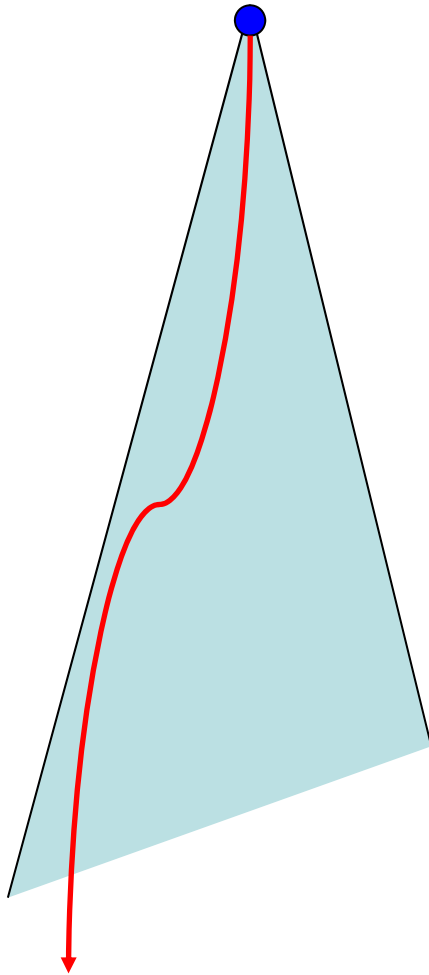
Malen for et dybde først-søk ser slik ut (rekursivt):

```
proc DFS(v)
{
    v.visited=TRUE
    for <hver nabo w av v> do
        if not w.visited then DFS(w)
    od
}
```

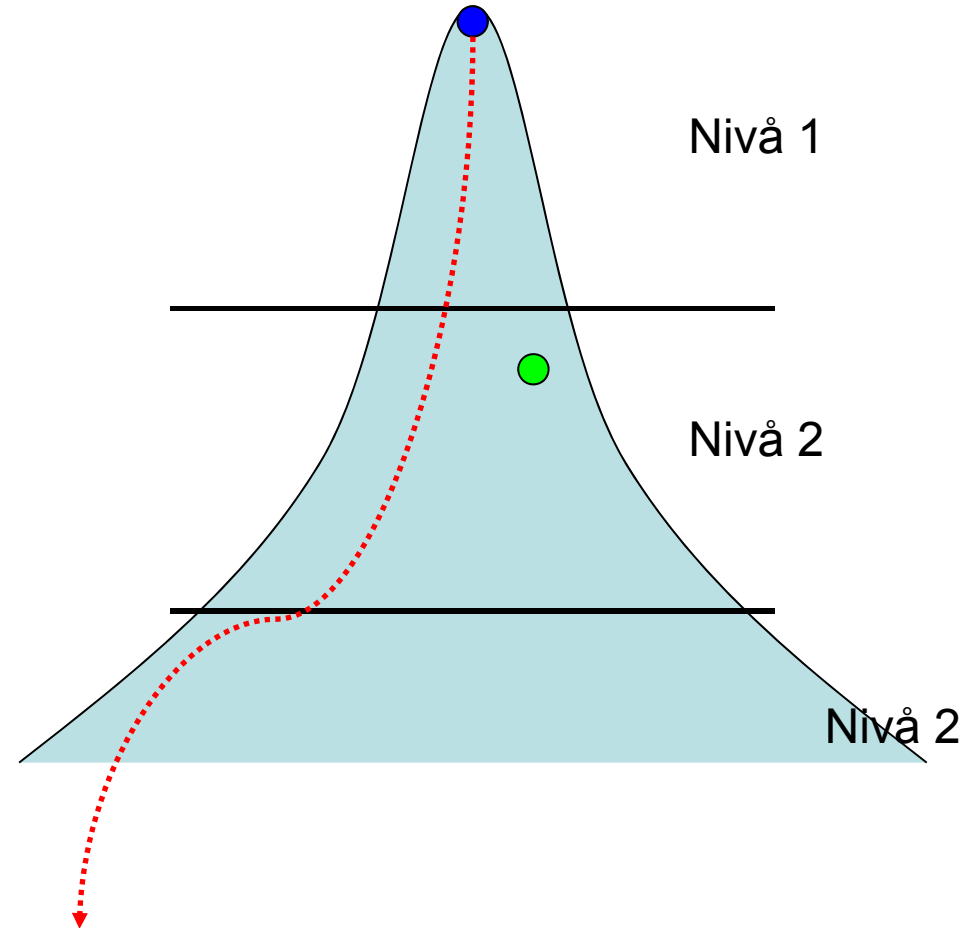
Dette er en meget anvendelig teknikk.

Iterativ fordypning

Ulempen med dybde først er at man kan «gå seg bort» i en håpløs gren.



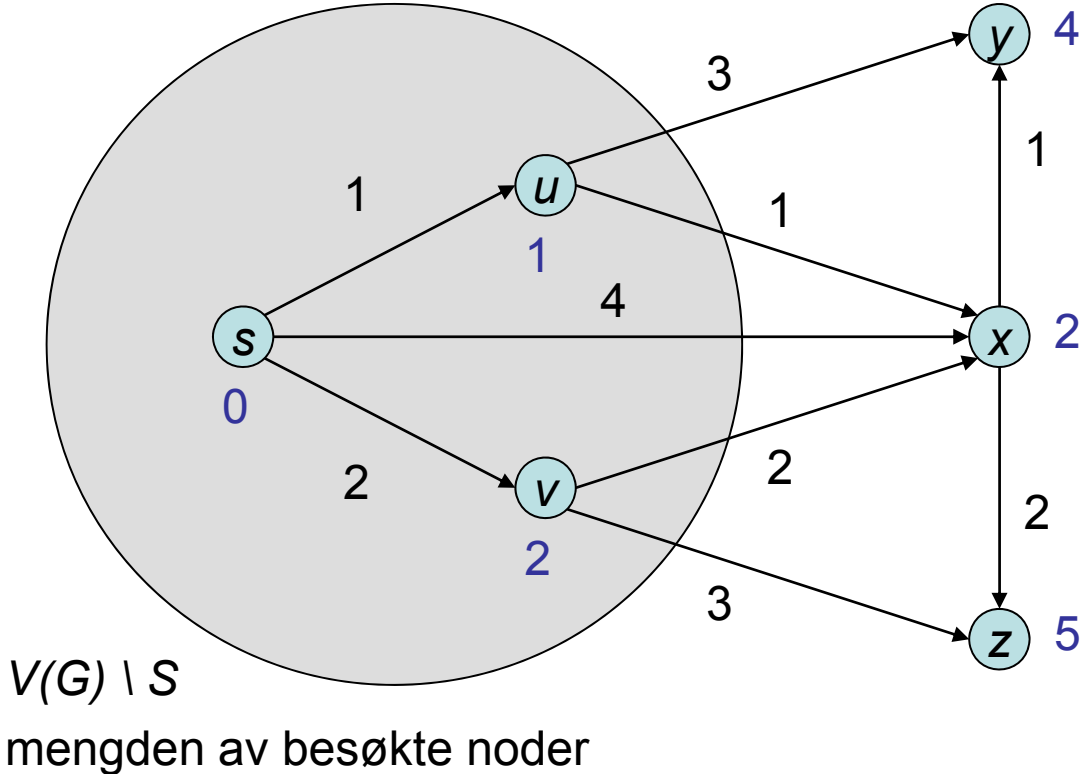
Unngår dette med iterativ fordypning, på bekostning av litt ekstraarbeid.



Dijkstras algoritme

```
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
S := V(G)                                // Mengden av ubesøkte noder, initielt alle
while S is not empty do
    u := extract_min(S)                    // Nærmeste node fra prioritetskø, source første
    for each neighbor v of u do          // gang. Key i køen er dist-verdien
        v.alt = u.dist + length(u, v)
        if v.alt < v.dist then           // «Relax» (u,v)
            v.dist := v.alt
            v.previous := u
        fi
    od
od
```

Dijkstras algoritme



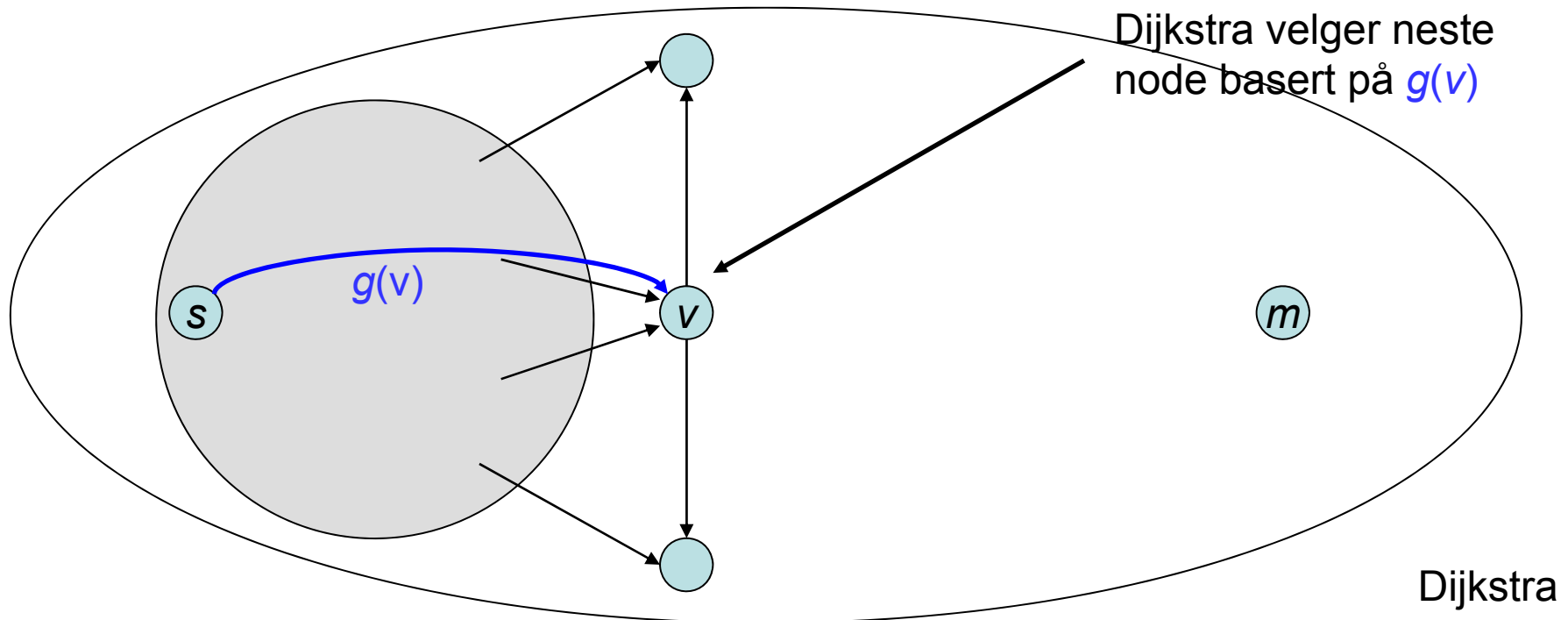
A*-søk (Hart, Nilsson, Raphael 1968)

- Backtracking / dybde først, LIFO / FIFO *branch-and-bound*, bredde først og Dijkstra bruker bare lokal informasjon når man velger, og ekspanderer, en node. Man ser alltid bare på en node og dens naboer.
- A*-søk er en type heuristisk søk som forsøker å se lengre enn bare til en nodes umiddelbare naboer.
- Mye brukt innen AI og kunnskapsbaserte systemer, og i dataspill.
- A*-søk er egnet i problemer hvor vi har
 - en (eksplisitt eller implisitt) tilstandsgraf,
 - med en start-tilstand, og et antall mål-tilstander, hvor
 - mulige tilstands-overganger er rettede kanter med en gitt kost,og hvor vi skal finne en vei fra start til en måltilstand, med minimal kost.

Altså logisk sett: korteste-vei-problemet

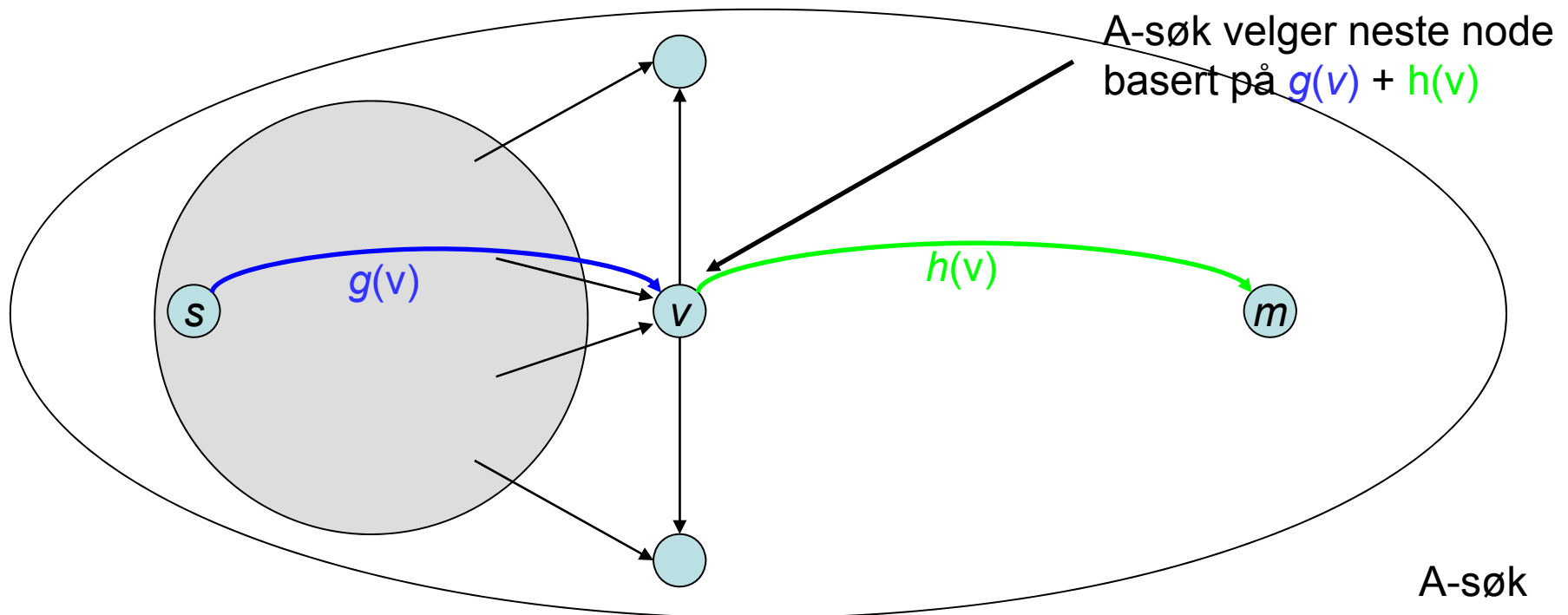
A*-søk – heuristikk

- 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 sti-algoritme. (Dijkstra er faktisk et spesialtilfelle av A*-søk.)



A*-søk – heuristikk

- 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 sti-algoritme. (Dijkstra er faktisk et spesialtilfelle av A*-søk.)



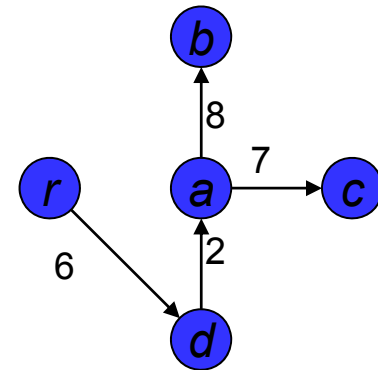
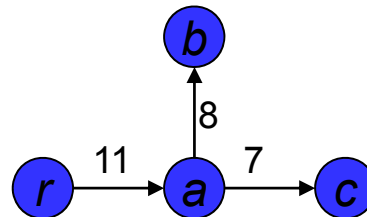
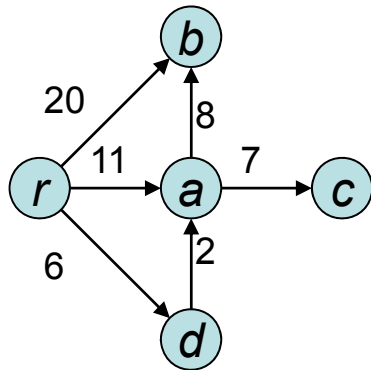
A*-søk – heuristikk

Heuristikk-funksjonen $h(v)$ skal være et estimat på hvor langt det er fra v til nærmeste mål-tilstand.

- For at algoritmen skal kalles et A*-søk forlanger vi at $h(v)$ skal aldri være større enn den faktisk korteste vei til nærmeste mål. Altså et underestimat – et optimistisk estimat.
- I tillegg kreves gjerne en slags trekant-ulikhet for $h(v)$. Det gjør algoritmen mye raskere. Vi slipper oppdatering av noder vi allerede har behandlet, og som ligger i spenntreet.

A*-søk – heuristikk

- Dersom $h(v)$ aldri er større enn faktisk korteste vei fra v til nærmeste mål,
 - og vi bruker en Dijkstra-liknende algoritme, med passelig bruk av $h(v)$,
 - 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!

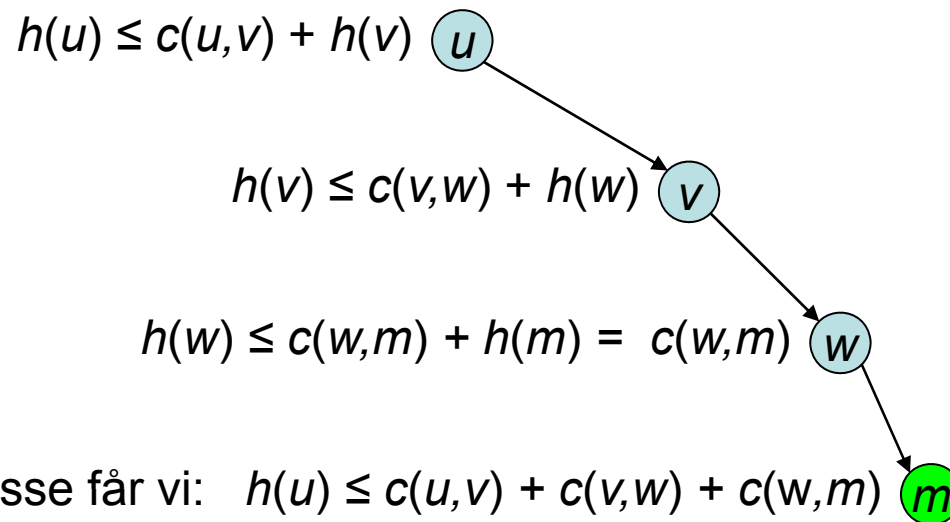


v	r	a	b	c	d
$h(v)$	-	23	20	15	29

A*-søk – heuristikk

- For å unngå stadig oppdateringer av treet, legger vi et ekstra «monotonitets-krav» på h :
 - Om det er kant fra v til w med vekt $c(v,w)$, så skal gjelde: $h(v) \leq c(v,w) + h(w)$.
 - Alle mål-noder m har $h(m) = 0$.
- Det fine er at dette kravet medfører det første kravet (understimatskravet), så vi slipper å tenke på det.

Bevis: Vi antar at $u \rightarrow v \rightarrow w \rightarrow m$ er korteste vei fra v til måltilstanden m :



A*-søk – algoritmen

- Vi har en rettet graf G med kantvekter $c(u, v)$, en startnode og et antall mål-noder, samt en monoton heuristikk-funksjon h .
- Hver node v har i tillegg følgende variable:
 - $g(v)$ som stadig vil forandre seg under algoritmen, men som til slutt får lengden av korteste vei fra startnoden til v .
 - $parent(v)$ som skal bli foreldre-peker i et tre av korteste veier fra start-noden.
 - $f(v)$ som hele tiden er lik $g(v) + h(v)$, altså et estimat av veilengden fra start til et mål gjennom v .
- Vi har en prioritets-kø PQ av noder, der $f(v)$ er nøkkel (prioriteten).
 - 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)

A*-søk – algoritmen

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

A*-søk – monoton heuristikk

- Om vi bruker $h(v) = 0$ 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 v når den taes ut av PQ og over i treet (slik at vi stadig måtte gå tilbake og oppdatere $g(w)$ og $parent(w)$ for noder w i treet (som har forlatt PQ)

MEN heldigvis gjelder (proposition 23.3.2 i boka):

- Om h er monoton, så vil verdiene av $g(v)$ og $parent(v)$ alltid være riktige i det øyeblikk v taes ut av PQ over i treet.
- Dermed behøver vi aldri gå tilbake i treet og oppdatere noe.

Merk: Det er en viktig trykkfeil på side 724, formel 23.3.7:

Der det står: ... $h(v) + h(v)$... skal det stå ... $h(v) \leq g(v) + h(v)$...

A*-søk – monoton heuristikk

Bevis: Vi bør bruke induksjon (ikke som i boka):

Induksjonshypotese: Setningen (Proposition 23.3.2) gjelder for alle w som er flyttet fra PQ til treet *før* v . (Verdiene $g(w)$ og $parent(w)$ riktige for alle w .)

Steg: Vi må nå vise at den da også gjelder da også for v . (Vi ser kun på $g(v)$.)

Vi lar generelt $g^*(u)$ være lengden av korteste vei fra start-noden til en node u . (Vi skal altså ha $g(u)=g^*(u)$ når nodene legges i treet.)

Vi ser på situasjonen når v taes ut av PQ, og ser på nodesekvensen P :

start-noden = $v_0, v_1, v_2, \dots, v_j = v$

som er en korteste vei fra start-noden til v (med lengde $g^*(v)$)

Vi antar at v_0, v_1, \dots, v_k (men ikke v_{k+1}) er blitt tre-noder når v taes ut av PQ, slik at v_{k+1} er i PQ når v blir tatt ut av køen.

A*-søk – monoton heuristikk

Ut fra monotonisiteten vet vi at (for $i = 0, 1, \dots, j-1$)

$$g^*(v_j) + h(v_j) \leq g^*(v_j) + c(v_j, v_{j+1}) + h(v_{j+1})$$

Siden kanten fra (v_j, v_{j+1}) er med i en korteste sti til v_{j+1} , gjelder

$$g^*(v_{j+1}) = c(v_j, v_{j+1}) + g^*(v_j)$$

Til sammen gir de to:

$$g^*(v_j) + h(v_j) \leq g^*(v_{j+1}) + h(v_{j+1})$$

som så, ved å la i være $k+1, k+2, \dots, j-1$, gir

$$g^*(v_{k+1}) + h(v_{k+1}) \leq g^*(v_j) + h(v_j) = g^*(v) + h(v)$$

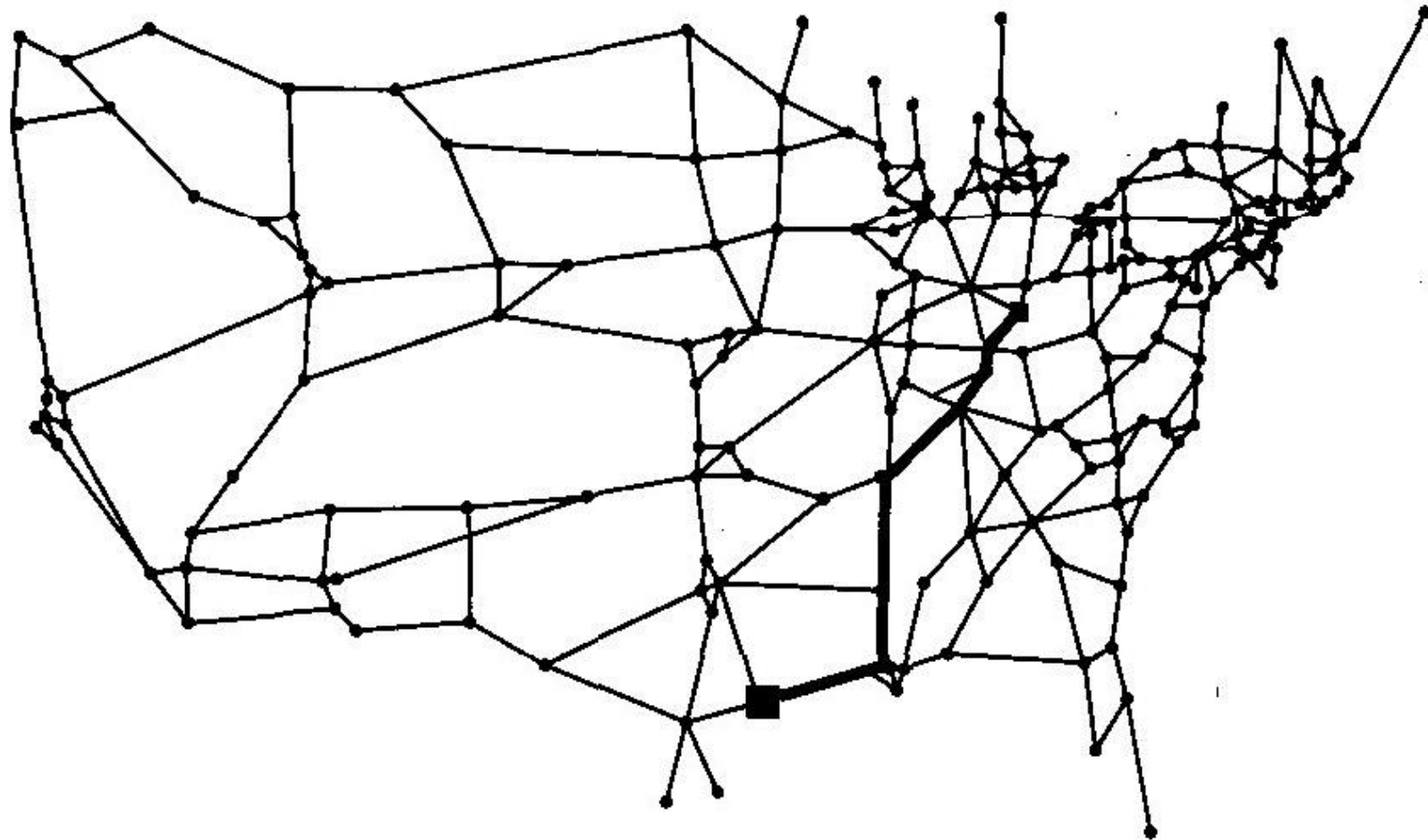
Ut fra induksjonshypotesen vet vi at $g(v_k) = g^*(v_k)$, og dermed må også (ut fra aksjonen når v_k ble tatt ut av PQ) $g(v_{k+1}) = g^*(v_{k+1})$, selv om den ligger i PQ.

Derved har vi:

$$f(v_{k+1}) = g(v_{k+1}) + h(v_{k+1}) = g^*(v_{k+1}) + h(v_{k+1}) \leq g^*(v) + h(v) \leq g(v) + h(v) = f(v)$$

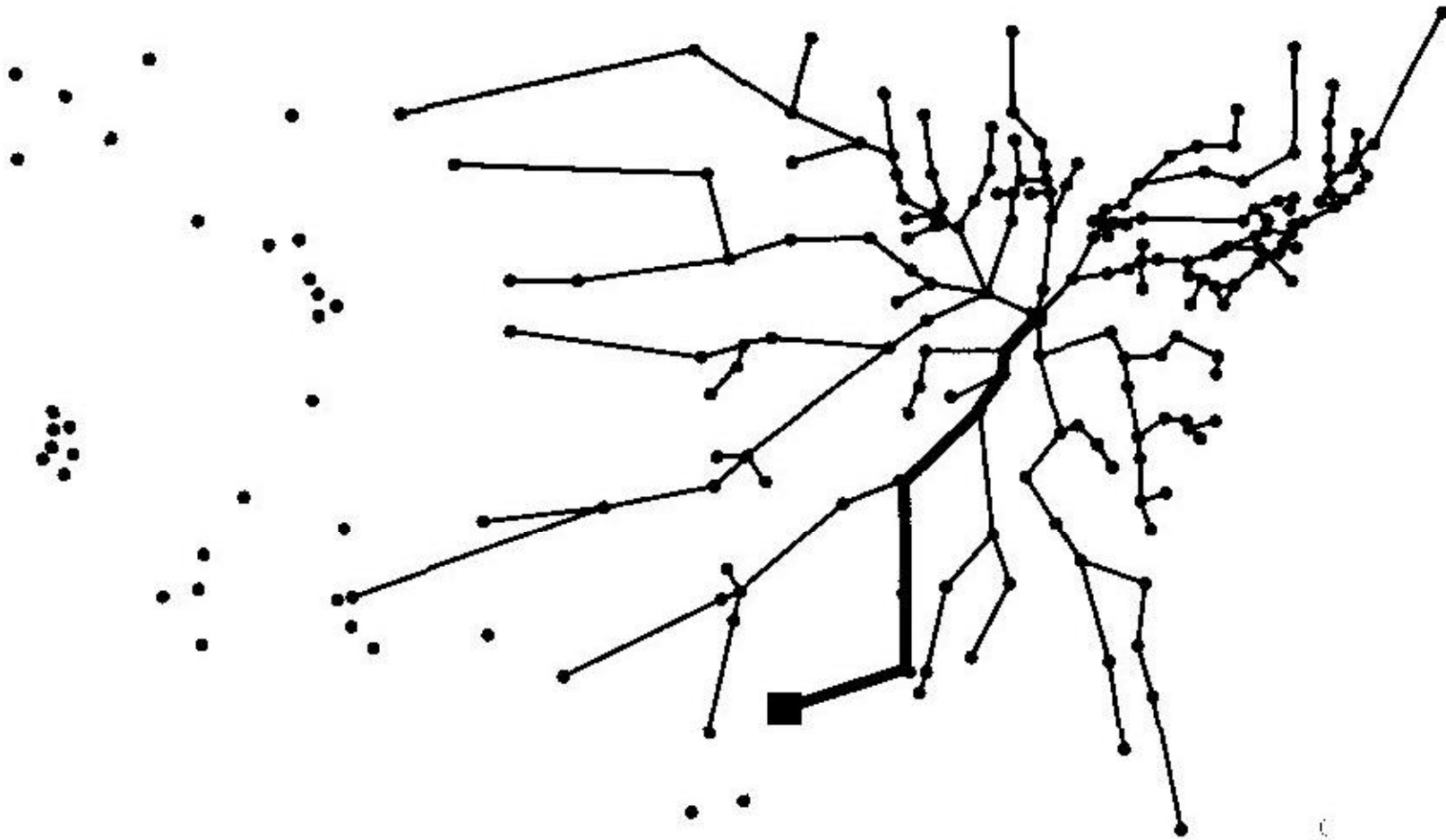
Her må imidlertid alle \leq være likheter, ellers ville $f(v_{k+1}) < f(v)$, og da ville ikke v blitt tatt ut av køen før v_{k+1} . Derved er $g^*(v) + h(v) = g(v) + h(v)$ og altså $g^*(v) = g(v)$.

A*-søk



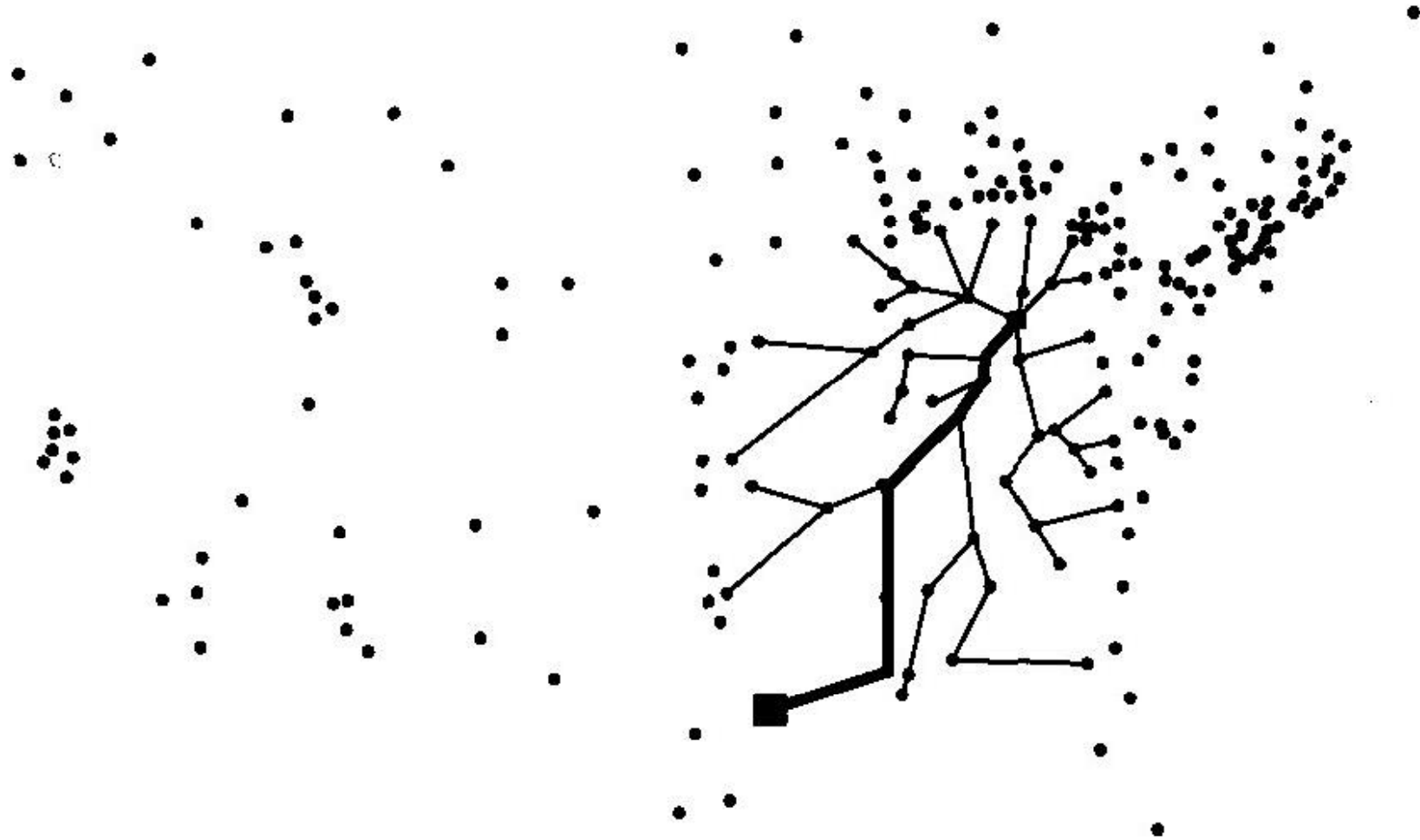
Amerikanske highways, korteste vei Cincinatti - Houston uthevet.

A*-søk



Treet generert av Dijkstras algoritme (stopper i Houston).

A*-søk



Treet generert av A*-søk.