3 Ongelmanratkaisu ja etsintä

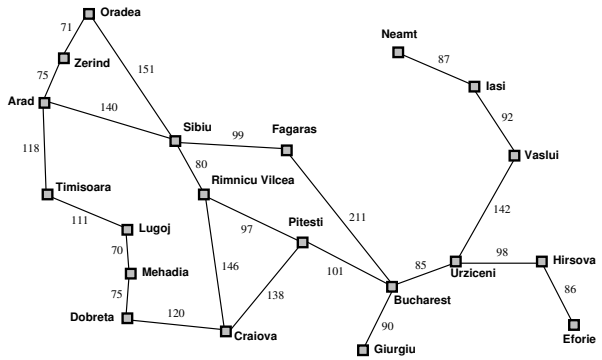
- Keskitytään sellaisiin päämäärähakuisiin toimijoihin, jotka yrittävät ratkoa ongelmia: päämääränä on löytää sellainen tila jossa ongelma on ratkaistu.
- Tarkastellaan aluksi tähän tehtävään sopivia, *sokeita* (uninformed, blind) etsintämenetelmiä.
- Sokealla tarkoitetaan sitä että menetelmällä ei ole käytettävissä mitään ongelmakohtaista tietoa ongelman määrittelyn lisäksi. Se voi ainoastaan käydä läpi etsintäavaruutta systemaattisesti seuraajatiloja generoiden kunnes löytää jonkin päämäärätilan.
- Systemaattinen etsintä on tehotonta, mutta joskus ongelmasta ei ole saatavilla hyödynnettävää tietoa.

Ongelmanratkaisun vaiheet

- *Päämäärän asettaminen* (goal formulation) perustuu toimijan nykyiseen tilanteeseen ja suoritusmittaan (tavoitteena suoritusmitan arvon maksimointi). Päämäärä eli maali on jokin mahdollisten maailman tilojen osajoukko.
- *Ongelman asettaminen* vaatii oikean esitystason valintaa.
- *Etsinnässä* eli *haussa* syötteenä on ongelma ja palautteena on ratkaisu, joka on alkutilanteesta johonkin maalitilaan johtava lista tekoja. Haku vaatii tietoa tekojen seurauksista.
- Kun ratkaisu on löytynyt, se *suoritetaan*.

Siis ensin etsitään ratkaisun tuottava toimintajono ja sitten vasta ryhdytään toimeen. Kyse on siis *offline-* eikä *online-etsinnästä*.

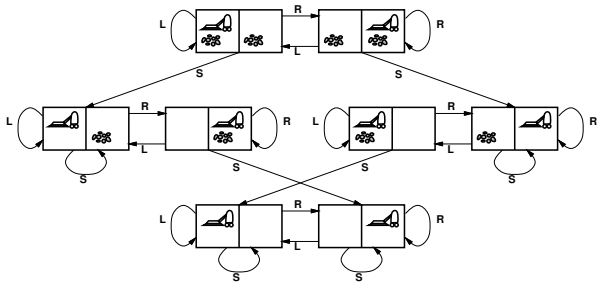
Esimerkki: Romanian tiekartta



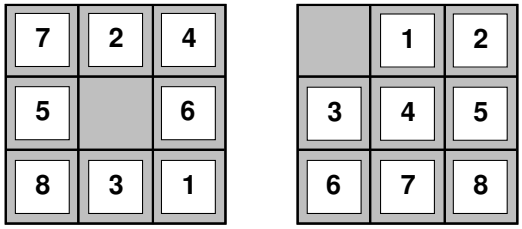
Oletukset ja määrittely

- Oletuksina on ympäristö, joka on staattinen, täysin havaittavissa, diskreetti (toimintojen suhteen) ja deterministinen.
- Ongelman määritelmä neljällä komponentilla:
  1. alkutila (initial state)
  2. toiminnat (actions), seuraajafunktio (successor function)
  3. maaliehto (goal test)
  4. polkukustannus (path cost)
- 1 ja 2 yhdessä määrittelevät *tila-avaruuden*.
- Ongelman ratkaisu on polku alkutilasta johonkin päämäärätilaan. Optimaalinen ratkaisu on sellainen, jolla on matalin polkukustannus.
- Ongelmien muotoilussa täytyy käyttää *abstraktiota*.
- Leluongelmat / todellisen maailman ongelmat

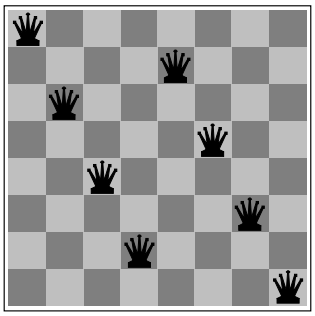
Pölynimurimaailma



8-liukupeli

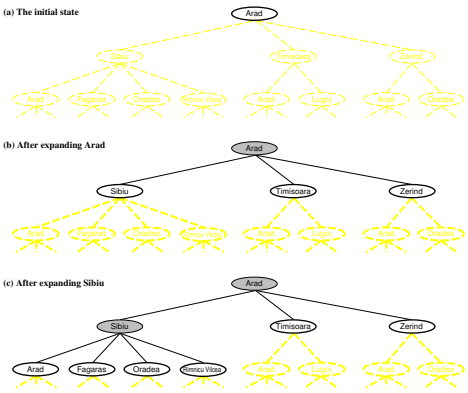


Kahdeksan kuningattaren ongelma



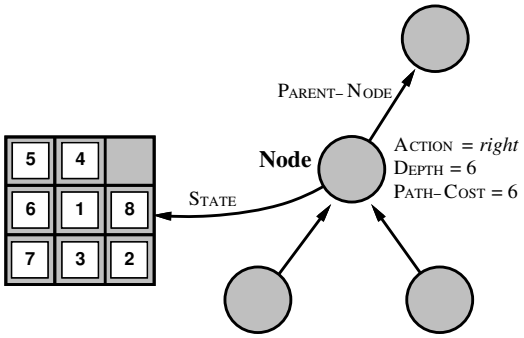
Etsintäprosessi

- Haku etenee *tila-avaruudessa*, mutta se voi edetä monella eri tavalla. Nyt käsitellään sellaisia, jotka muodostavat *hakupuun* alkutilasta seuraajafunktion avulla. Puun solmut eli hakusolmut vastaavat ympäristön tiloja. Hakupuun juuri vastaa alkutilaa.
- Haku noudattaa seuraavaa kaavaa:
  1. Tarkistetaan onko tämänhetkinen solmu maalisolmu.
  2. Laajennetaan tämänhetkinen solmu generoimalla sen seuraajasolmut seuraajafunktiota käyttäen.
  3. Valitaan tämänhetkiseksi solmuksi jokin generoiduista solmuista.
- *Hakustrategiassa* on kyse siitä, mikä solmu valitaan kohdassa 3.



TIETORAKENTEET

- Huom: Tila-avaruus ja hakupuu ovat eri asioita! Esim. Romanian kartassa 20 tilaa mutta hakupuu ääretön.
- Solmu on tietorakenne, jossa on seuraavat komponentit:
  1. tila (ympäristön tila)
  2. vanhempisolmu
  3. toiminta (jolla vanhemmasta päästiin tähän)
  4. polkukustannus $g(n)$ (alkusolmusta tähän)
  5. syvyys (askelten lukumäärä puussa alkusolmusta lähtien)
- Lisäksi tarvitaan tietorakenne generoitujen solmujen tallettamiseen.
- Kirjassa puhutaan jonosta (queue) yleisenä tietorakenteena ja sitten erotellaan FIFO- ja LIFO-jonot (eli pinot). Käytetään tässä mieluummin laitoksella tavallista terminologiaa ja sanotaan että varasto voidaan toteuttaa joko jonona (FIFO) tai pinona (LIFO).



ETSINTÄMENETELMIEN ARVIOINTI

Etsintäongelmien ratkaisumenetelmää M voidaan arvioida ainakin neljällä kriteerillä:

Täydellisyys: Löytääkö M ratkaisun aina kun sellainen on olemassa? (vai voiko se sivuuttaa oikean ratkaisun ja pysähtyä vastaukseen "ei ole" tai jäädä kokonaan pysähtymättä?)

Optimaalisuus: Löytääkö M aina kustannukseltaan edullisimman ratkaisun? (Tai kuinka lähelle parasta ratkaisua M yleensä pääsee?)

Aikavaativuus: Kuinka paljon aikaa menetelmä käyttää ratkaisun löytämiseen? (Mitataan generoitujen solmujen määrällä.)

Tilavaativuus: Kuinka paljon muistia menetelmä tarvitsee? (Mitataan muistissa yht'aikaa pidettävien solmujen maksimimäärällä.)

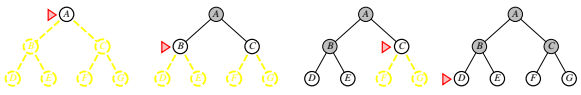
SOKEAT ETSINTÄSTRATEGIAT

Tarkastellaan seuraavia sokeita etsintästrategioita:

- Leveyssuuntainen etsintä (breadth-first search)
- Halvin ensin -etsintä (uniform cost search)
- Syvyyssuuntainen etsintä (depth-first search)
- Syvyysrajoitettu etsintä (depth-limited search)
- Iteratiivinen syventäminen (iterative deepening depth-first search)
- Kaksisuuntainen etsintä (bidirectional search)

LEVEYSSUUNTAINEN ETSINTÄ

- Jos solmuvarastoksi otetaan jono, niin saadaan etsintäpuun leveyssuuntainen läpikäynti. Ensin laajennetaan juurisolmu, sitten sen lapset, sitten niiden lapset jne taso kerrallaan.
- Etsintä on täydellinen, kunhan haarautumisaste (branching factor) on äärellinen.
- Löytää lähimmällä tasolla olevan maalin, mutta se ei välttämättä ole optimaalinen jos tekojen kustannukset vaihtelevat.
- Ongelmana on eksponentiaalinen (ajan ja) tilan tarve. Jos haarautumisaste on $b > 1$ ja d lyhimmän ratkaisun polun pituus eli avaruuden halkaisija (diameter), niin läpikäydyn puun koko (=laajennusten määrä) voi olla $b + b^2 + b^3 + \ldots + b^d + b^{d+1} - b)$, joka on luokkaa $O(b^{d+1})$.
- Jokainen generoitu solmu on jossain muistissa koska se on joko jonossa (laajennettavien varastossa) tai jonkun siellä olevan solmun esi-isä $\Rightarrow$ Muistivaatimus kohtuuttomaksi.



Oletetaan että haarautumisaste on 10, solmuja läpikäydään 10000 sekunnissa ja yhden solmun talletustila on 1000 tavua:

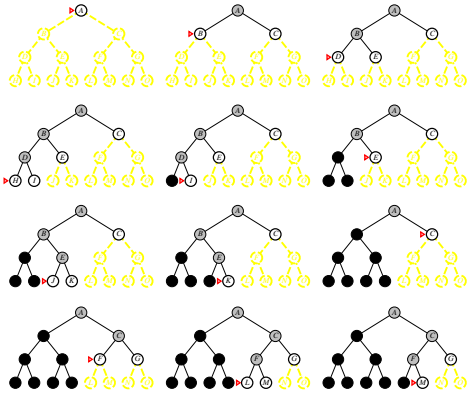
Depth	Nodes	Time	Memory
2	1100	0.11 s	1 megabytes
4	111100	11 s	106 megabytes
6	10^7	19 min	10 gigabytes
8	10^9	31 h	1 terabytes
10	10^{11}	129 d	101 terabytes
12	10^{13}	35 a	10 petabytes
14	10^{15}	3523 a	1 exabyte

HALVIN ENSIN -ETSINTÄ

- Halvin ensin -menetelmä ei välitä polun askelten määrästä, vaan polun kokonaiskustannuksesta: se valitsee tarkasteltavaksi aina sen solmun, johon johtaa halvin polku.
- Tämä voidaan toteuttaa käyttämällä tallennettujen solmujen varastona *kekoa* (heap), jossa järjestys määräytyy kuhunkin solmuun liittyvän polun kokonaiskustannuksen mukaan, ja siitä valitaan aina pienin.
- Algoritmi on täydellinen ja optimaalinen, jos kustannukset eivät voi olla negatiivisia (=polun kokonaishinta ei laske).
- Aika- ja tilavaatimusta on vaikea luonnehtia b:n ja d:n avulla, mutta jos C^* on parhaan ratkaisun kustannus, niin tila- ja aikavaatimus on pahimmassa tapauksessa luokkaa $O(b^{\lceil C^*/\epsilon \rceil})$, missä ϵ on yhden toiminnon minimikustannus. Tämä voi olla paljon enemmän kuin b^{d+1}, sillä algoritmi voi tutkia suuren määrän halpoja vääriä toimintoja ennen kuin löytää kalliin oikean. Jos tekojen kustannukset ovat aina samat, tämä on sama kuin b^{d+1}.

Syvyyssuuntainen etsintä

- Syvyyssuuntainen etsintä laajentaa aina syvimmällä olevan laajentamattoman solmun.
- Tämä voidaan toteuttaa valitsemalla solmuvarastoksi *pino* (stack). Toinen tapa on käyttää rekursiota.
- Muistivaatimus on vähäinen: muistissa vain yksi polku kerrallaan sekä (varastossa) laajentamattomat lapsisolmut.
- Jos hakuavaruudessa on maksimisyvyys m ja haarautumisaste b, niin muistitilan tarve on $bm + 1$ solmua. Pahimman tapauksen aikavaatimus on $O(b^m)$.
- Muistin määrä on vielä pienempi ($O(m)$) *peruuttavassa syvyyssuuntaisessa etsinnässä* (backtracking search), jossa vain yksi seuraaja generoidaan kerrallaan ja solmuista talletetaan kohta, josta jatketaan myöhemmin.
- Ei ole optimaalinen eikä täydellinen, koska jos m on ääretön, etsintä saattaa joutua äärettömään etsintähaaraan.

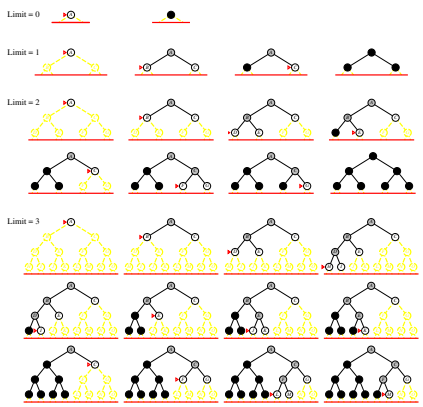


Syvyysrajoitettu etsintä

- Äärettömien oksien ongelma voidaan välttää määrittämällä haulle syvyysraja l, jota syvemmälle syvyyssuuntainen etsintä ei mene.
- Valitettavasti l:n valinta on vaikeaa ja menetelmä jää epätäydelliseksi jos rajaksi valittu l on pienempi kuin d. Toisaalta jos $l > d$, menetelmä ei ole optimaalinen.
- Aikavaativuus on $O(b^l)$ ja tilavaativuus $O(bl)$.
- Joskus syvyysrajaa voidaan arvioida ongelmaa koskevan tietämyksen perusteella. Esimerkiksi Romanian kartassa, jossa on vain 20 kaupunkia, tiedetään heti ettei polku voi olla pitempi kuin 19 ja tarkemmin katsoen huomataan että kaikki kaupungit ovat saavutettavissa toisistaan korkeintaan yhdeksällä siirtymällä (= tila-avaruuden halkaisija). Yleensä sopivaa rajaa ei kuitenkaan tiedetä ennen kuin ongelma on ratkaistu.

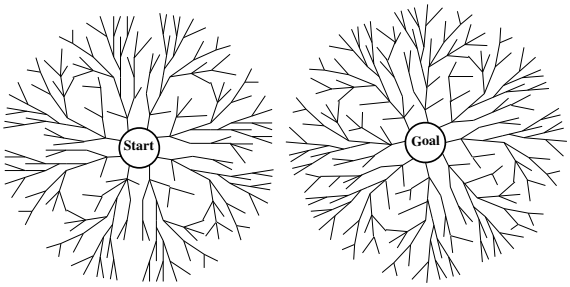
Iteratiivinen syventäminen

- Iteratiivinen syventäminen kutsuu syvyysrajoitettua etsintää asettamalla syvyysrajaksi ensin 0:n, sitten 1:n, 2:n ja niin edelleen kunnes päämäärä löytyy tasolla d.
- Menetelmä yhdistää leveys- ja syvyyssuuntaisen etsinnän hyvät puolet: tilavaatimus on vain luokkaa $O(bd)$ ja haku on täydellinen kunhan haarautumiskerroin on äärellinen ja optimaalinen kun polkukustannus on syvyyden kasvava funktio. Aikavaatimus on $O(b^d)$, sillä generoitujen solmujen määrä on $d \cdot b + (d-1) \cdot b^2 + ... + 1 \cdot b^d$.
- Menetelmä vaikuttaa tuhlaavaiselta, mutta ei ole sitä, koska ylemmät kerrokset ovat pieniä verrattuna alempiin. Varsinkin jos b on suuri, voi menetelmä generoida selvästi vähemmän solmuja kuin leveyssuuntainen etsintä, joka joutuu usein generoimaan myös tason $d + 1$ solmuja.
- Yleisesti ottaen iteratiivista syventämistä onkin pidettävä suositeltavana sokeana etsintämenetelmänä silloin kun etsintäavaruus on suuri eikä ratkaisun syvyyttä tunneta!



Kaksisuuntainen etsintä

- Etsitään sekä alku- että maalitilasta samanaikaisesti (tav. leveyssuuntaisesti) ja lopetetaan jos haut kohtaavat. Säästetään aikaa, sillä $b^{d/2} + b^{d/2}$ paljon pienempi kuin b^d.
- Muistia tarvitaan paljon, koska ainakin toisen puun kaikki lehdet täytyy pitää muistissa jotta toisen puun senhetkistä solmua voidaan verrata niihin. Tarkistus voidaan tehdä hajautustaululla vakioajassa, joten aikavaativuus pysyy luokassa $O(b^{d/2})$, mutta tilavaativuus on samaa luokkaa.
- Etsintä taaksepäin ei aina ole helppoa sillä teon edeltäjien joukko *Pred*(n) pitäisi voida laskea. Jos toiminnot ovat *kumottavissa* (reversible) kuten esim. liukupeleissä, etsintä taaksepäin on helppoa, mutta muuten se voi olla vaikeaa.
- Maalitilojen lukumäärä voi vaikeuttaa tilannetta: jokaisesta maalitilasta ei voi aloittaa taaksepäin, jos maalitilat määritellään jollakin maaliehdolla joka tarkastelee vain jotain tilan piirrettä (esim. shakkipelin mattitilanne).



Vertailu

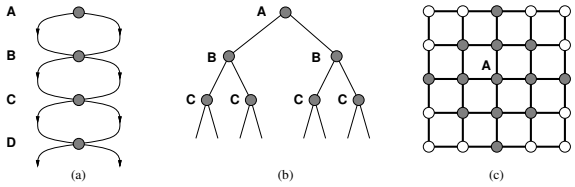
Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(bl)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

Evaluation of search strategies. b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit. Superscript caveats are as follows: [a] complete if b is finite; [b] complete if step costs $\geq \epsilon$ for positive ϵ; [c] optimal if step costs are all identical; [d] if both directions use breadth-first search.

(AIMA, Fig. 3.17)

SAMOJEN TILOJEN VÄLTTÄMINEN

- Joissain tapauksissa samoihin tiloihin ei voi joutua uudestaan (esim. 8 kuningattaren ongelma kirjan muotoilun mukaan), mutta yleensä joudutaan huolehtimaan samojen tilojen esiintymisestä. Esimerkiksi silloin kun teot ovat kumottavia, hakupuut ovat äärettömiä, mutta jos toistuvat tilat poistetaan, puu saadaan pysymään äärellisenä.
- Äärellisissäkin tapauksissa voidaan toistojen poistamisella joskus saada jopa eksponentiaalinen vähennys etsintäkuluihin. Vrt. seuraavan kuvan kohtia (a) ja (b).
- Syvyyssuuntaisessa etsinnässä ainoat muistissa olevat solmut ovat tutkittavalla polulla olevat (ja niiden lapset) joten uuden solmun vertaaminen niihin estää haarojen kasvamisen äärettömäksi koska toistuvat tilat voidaan hylätä. Näin ei voi välttää hakuavaruuden kasvamista eksponentiaaliseksi kohdassa (c). Ainoa keino välttää kasvu on pitää käsiteltyjäkin solmuja muistissa, jolloin tilavaatimus kasvaa.



- Jos kaikki käsitellyt solmut pidetään muistissa, haun voidaan ajatella käsittelevän verkkoa puun sijaan. Otetaan käyttöön toinen tietovarasto käsitellyille solmuille. Kutsutaan sitä suljetuksi listaksi ja aiempaa laajennettavien solmujen varastoa avoimeksi listaksi. Nyt tämänhetkistä solmua verrataan aina kaikkiin suljetun listan solmuihin ja jos sama tila löytyy sieltä, tämänhetkinen solmu voidaan hylätä. Etsintä tehostuu huomattavasti jos tilat toistuvat usein.
- Optimaalisuus tulee ongelmalliseksi, sillä tilan toistuessa hylätään aina uusimman löydetyn esiintymä vaikka se olisi halvempi kuin aiemmin löydetty. Joissakin tapauksissa optimaalisuus voidaan kuitenkin säilyttää.
- Suljettujen listan käyttö saa aikaan sen, että syvyyssuuntaisen ja iteroivan syventämisen tilavaatimukset eivät enää pysy lineaarisina, vaan kaikki solmut täytyy pitää muistissa. Tilavaatimus kasvaakin tällöin helposti epäkäytännölliseksi.

TREE-SEARCH-ALGORITMI

```
function TREE-SEARCH( problem, fringe) returns a solution, or failure
  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
      if fringe is empty then return failure
      node ← REMOVE-FRONT(fringe)
      if GOAL-TEST(problem, STATE(node)) then return node
      fringe ← INSERTALL(EXPAND(node, problem), fringe)

function EXPAND( node, problem) returns a set of nodes
  successors ← the empty set
  for each action, result in SUCCESSOR-FN(problem, STATE[node]) do
      s ← a new NODE
      PARENT-NODE[s] ← node; ACTION[s] ← action; STATE[s] ← result
      PATH-COST[s] ← PATH-COST[node] + STEP-COST(STATE[node], action, result)
      DEPTH[s] ← DEPTH[node] + 1
      add s to successors
  return successors
```


GRAPH-SEARCH-ALGORITMI

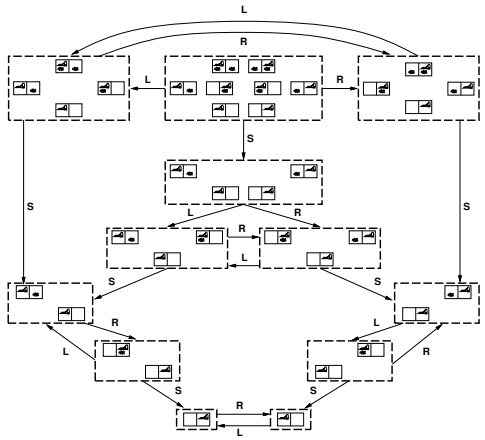
```
function GRAPH-SEARCH( problem, fringe) returns a solution, or failure

  closed ← an empty set
  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
      if fringe is empty then return failure
      node ← REMOVE-FRONT(fringe)
      if GOAL-TEST(problem, STATE[node]) then return node
      if STATE[node] is not in closed then
          add STATE[node] to closed
          fringe ← INSERTALL(EXPAND(node, problem), fringe)
  end
```


ETSINTÄ PUUTTEELLISELLA TIEDOLLA

Mitä jos ympäristö ei olekaan deterministinen tai täydellisesti havaittavissa? Seuraa ongelmia:

1. Havaintojen puutteellisuus
   - Jos ympäristö ei ole täysin havaittavissa tai ääritapauksessa jos sensoreita ei ole lainkaan, toimija voi olla tietämätön alkutilanteesta ja on tehtävä päätelmiä useiden mahdollisten tilojen joukoista → *uskomustilat* (belief states). Tilanne ei ole toivoton, jos tekojen seuraukset tunnetaan varmasti.
   - Ongelma on ratkaistu, jos tavoitetaan uskomustilaksi sellainen mahdollisten maailmojen joukko, jonka kaikki tilat ovat maalitiloja (ks. kuva). Tämä ei aina onnistu.
   - Jos ympäristö ei ole deterministinen, vaan tekojen vaikutuksista ei voi olla varma, tilanne on pahempi. Jos "Murphyn laki" (imuri saattaa imuroitaessa päästää likaa puhtaalle lattialle) on voimassa, niin agentti joka ei havaitse ympäristöä, ei voi ratkaista ongelmaa varmasti.



2. Varautumisongelmat
   - Stokastisessa, dynaamisessa tai moniagenttiympäristössä uudet havainnot voivat tuoda informaatiota joka vaatiikin tekojonon muuttamista.
   - *Vuorotellaan* (interleaving) tekojonon etsintää ja toteutusta: etsitään siihen asti kunnes ympäristöä on taas havainnoitava ja toteutetaan löydetyistä parhaalta vaikuttava tekojono. Sitten taas havainnoidaan ja jatketaan tulosten perusteella etsintää. Näin päästään toimeen nopeammin, mutta voidaan epäonnistuakin.
   - *Suunnitellaan* yksinkertaisen tekojonon sijasta havaintojen ehdollistamia toimintaohjelmia. Esim. while dirt suck end right while dirt suck end. Toimii paremmin mutta suunnitelman löytäminen voi olla vaikeaa.
   - Jos toimija ei tiedä etukäteen edes omien tekojensa vaikutusta, sen on *kokeiltava* eri tekoja satunnaisesti ja *opittava* erottamaan hyvät ratkaisut huonoista.