4 INFORMOITU ETSINTÄ

- Sokeilla etsintämenetelmillä ei ole käytössään muuta informaatiota kuin etsintäongelman määrittely.
- *Informoidut etsintämenetelmät* (informed search strategies) ovat sellaisia, jotka voivat hyödyntää tehtäväkohtaista lisäinformaatiota. Niillä ratkaisu voidaan löytää sokeita menetelmiä tehokkaammin.
- Aina lisätietoa ei kuitenkaan ole käytettävissä.
- Yleistä informoitua etsintästrategiaa kutsutaan *paras ensin -strategiaksi* (best first). Siinä laajennettavaksi valitaan solmu *evaluointifunktioon* (evaluation function) f perustuen. Usein evaluointifunktio arvioi kustannusta päämäärään ja laajennettavaksi valitaan solmu, jonka f-arvo on pienin.
- Implementointi tehdään käyttäen kekoa, joka on järjestetty funktion f mukaan nousevaan järjestykseen.

- Paras ensin -nimitys on harhaanjohtava sillä se tarkoittaa tietysti parhaalta vaikuttavaa. Jos paras solmu tiedettäisiin etukäteen, niin mitään etsintää ei tarvittaisi.
- Etsinnässä käytettävä funktio f voidaan valita eri tavoin. Yleensä se perustuu johonkin heuristiseen funktioon h:

$$h(n) = \text{arvioitu halvin kustannus solmusta } n \text{ maalisolmuun.}$$

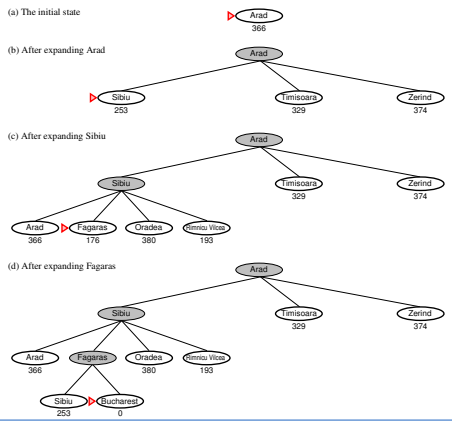
- Esimerkiksi Romanian kartan tapauksessa heuristisena funktiona voi käyttää linnuntie-etäisyyksiä Bukarestiin:

Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Dobreta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

AHNE PARAS ENSIN -STRATEGIA

- Ahne paras ensin -strategia pyrkii laajentamaan solmun, jonka arvioidaan olevan lähinnä maalia. Tässä tapauksessa siis f perustuu suoraan heuristiseen arvioon h, joten $f(n) = h(n)$.
- Paras ensin muistuttaa aiemmin esiteltyä halvin ensin -menetelmää, mutta se pyrkii aina mahdollisimman lähelle maalia, kun taas halvin ensin pysytteli mahdollisimman lähellä lähtösolmua.
- Toisin kuin syvyyssuuntaisella etsinnällä ahne paras ensin -menetelmällä on "suuntavaistoa": hakua ei jatketa ensimmäisestä lapsesta vaan siitä, jonka arvioitu etäisyys tavoitteesta on pienin. Menetelmä kärsii samoista ongelmista kuin syvyyssuuntainen etsintä: se ei ole optimaalinen eikä täydellinen (voi jäädä äärettömään haaraan).

- Pahimman tapauksen aika- ja tilavaatimukset ovat luokkaa $O(b^m)$, missä m on etsintäavaruuden maksimisyvyys. Tilavaatimus on näin suuri, sillä – toisin kuin syvyyssuuntainen etsintä – ahne etsintä joutuu pitämään muistissa kaikki etsintäpuun solmut mahdollista paluuta varten, eikä vain saman haaran solmuja.
- Jos heuristiikka on hyvin valittu, etsintää tehostuu huomattavasti, mutta tämä riippuu ongelmasta ja valitusta heuristiikasta.



A*

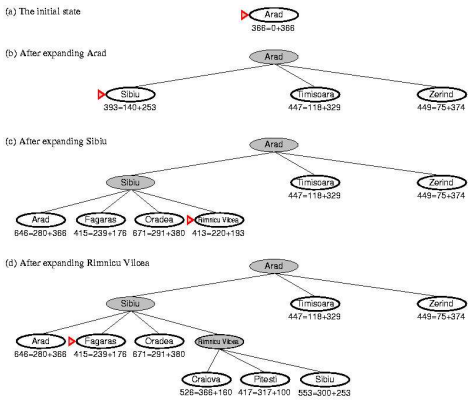
- Tunnetuin paras ensin -strategia on A*, joka evaluoi solmut niiden tämänhetkisen kustannuksen ja arvioidun kustannuksen summan perusteella:

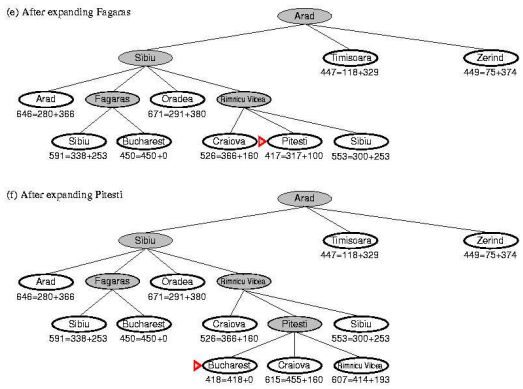
$$f(n) = g(n) + h(n),$$

missä g on kustannus alusta solmuun n ja h arvio halvimman reitin kustannuksesta solmusta n maalisolmuun. Siis $f(n)$ on arvioitu halvimman n:n kautta kulkevan ratkaisun kustannus.

- Tässä siis eräällä tavalla yritetään yhdistää halvin ensin - ja paras ensin -menetelmien hyvät puolet. Jos heuristinen funktio h täyttää tietyt ehdot, A* onkin sekä täydellinen että optimaalinen.

- Puuetsinnässä optimaalisuuteen riittää että funktio h on *luvallinen* (admissible), mikä tarkoittaa sitä, että se ei koskaan yliarvioi jäljellä olevaa kustannusta. Sellainen funktio on siis optimistinen, koska sen mukaan jäljellä oleva työmäärä näyttää pienemmältä (tai korkeintaan yhtä suurelta) kuin mitä se todellisuudessa on. Esimerkiksi etäisyyden arviointi linnuntietä on tällainen luvallinen heuristinen funktio.





Todistetaan, että A*, joka on toteutettu puuetsintänä algoritmin TREE-SEARCH mukaisesti on optimaalinen, jos $h(n)$ on luvallinen:
Olkoon C^* optimaalisen ratkaisun kustannus. Oletetaan että laajennettavien solmujen tietovarastossa on alioptimaalinen maalisolmu G'. Silloin $h(G') = 0$ koska G' on maalisolmu. Silloin

$$f(G') = g(G') + h(G') = g(G') > C^*.$$

Varastossa on myös oltava jokin optimaalisella polulla oleva solmu (jos kerran optimaalinen ratkaisu on olemassa). Nimetään sitä solmua n:llä. Jos h ei yliarvioi kustannusta, niin

$$f(n) = g(n) + h(n) \leq C^*.$$

Koska $f(n) < f(G')$, niin haku ei etene alioptimaaliseen maalisolmuun, vaan laajentaa optimaalisella polulla olevaa solmua. Sama päättely pätee jokaisella askeleella, joten A* palauttaa optimaalisen ratkaisun, jos h on luvallinen. □

Verkkoetsinnässä tämä ei enää päde, koska GRAPH-SEARCH-algoritmi voi hylätä optimaalisen polun toistuvaan tilaan, jos optimaalinen ratkaisu ei ole ensimmäiseksi löydetty solmu. Tämä ongelma voidaan korjata kahdella tavalla:

1. Joko modifioidaan algoritmia siten että se hylkää aina kalliimman toistuvista tiloista → lisäkirjanpitoa.
2. Tai varmistetaan että lyhin reitti löytyy aina ensimmäisenä kuten halvin ensin -menetelmässä. Asetetaan heuristiselle funktiolle lisärajoitus: *konsistenssi* eli *monotonisuus*, joka pätee täsmälleen silloin kun *kolmioepäyhtälö* on voimassa (jokaiselle solmulle n, sen seuraajalle n' ja teolle a):

$$h(n) \leq c(n, a, n') + h(n').$$

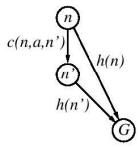
- Jokainen monotoninen heuristiikka on myös luvallinen.
- A* joka käyttää GRAPH-SEARCHiä on optimaalinen jos $h(n)$ on monotoninen!

Jos $h(n)$ on monotoninen, niin $f(n)$:n arvot ovat kasvavia jokaisella polulla.

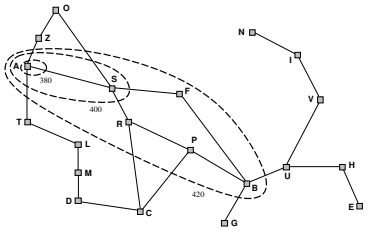
Todistus: Olkoon n' n:n seuraaja. Silloin $g(n') = g(n) + c(n, a, n')$ jollakin teolla a ja saadaan

$$f(n) = g(n)+h(n) \leq g(n)+c(n, a, n')+h(n') = g(n')+h(n') = f(n').\square$$

Tästä seuraa, että GRAPH-SEARCHiä käyttävä A* laajentaa solmut kasvavassa järjestyksessä, joten ensimmäisen laajennettavaksi valitun maalisolmun on oltava optimaalinen.



Koska f:n arvot ovat kasvavia, hakua voidaan havainnollistaa *korkeuskäyrillä* (contours):



Haku etenee varovasti ylöspäin laajentaen solmuja f:n mukaan kasvavassa järjestyksessä ohjautuen kohti optimaalista polkua, jolloin ensimmäinen löytyvä ratkaisu on optimaalinen.
A* on täydellinen, vaikka *karsiikin* ne solmut, joille $f(n) > C^*$ (kuten kuvassa esimerkiksi Timisoaran alla olevan alipuun).

- Tällaisten algoritmien joukossa A* on myös *optimaalisen tehokas* (optimally efficient) millä tahansa heuristisella funktiolla. Mikään optimaalinen algoritmi ei siis selviä vähemmällä solmujen laajentamisella kuin A* (lukuunottamatta sitä että joskus voidaan joutua laajentamaan turhaan joitakin solmuja joiden kustannus on sama kuin optimaalisen ratkaisun).
- Kuitenkaan A* ei aina kelpaa käytännössä, sillä sen etsintäavaruus kasvaa usein eksponentiaaliseksi suhteessa ratkaisun syvyyteen. Näin tapahtuu aina paitsi jos heuristisen funktion virheen suuruus kasvaa korkeintaan logaritmisesti verrattuna todelliseen polun pituuteen. Kasvu pysyy siis hallittuna silloin kun

$$|h(n) - h^*(n)| \leq O(log\ h^*(n)),$$

missä $h^*(n)$ on todellinen kustannus n:stä maaliin johtavalla polulla. Tämä on kuitenkin harvinaista.

- Virhe on lähes aina vähintääkin verrannollisessa suhteessa polkukustannukseen, joten eksponentiaalista kasvua ei yleensä voi välttää. Siksi usein on luovuttava vaatimuksesta löytää optimaalinen ratkaisu. Tällöin voidaan käyttää A*:n variantteja, jotka löytävät nopeasti alioptimaalisen ratkaisun, tai heuristisia funktioita jotka eivät ole luvallisia eivätkä siis takaa optimaalisuutta.
- Toisaalta menetelmän epäkäytännöllisyyteen johtaa myös se seikka, että A*:n tilavaatimus on korkea, koska kaikki käsitellyt solmut säilytetään muistissa kuten aina verkkoetsinnässä (GRAPH-SEARCH). Tarkastellaan seuraavaksi algoritmeja, jotka tulevat toimeen vähemmällä muistilla mutta tuottavat silti optimaalisen ratkaisun.

MUISTIRAJOITETTU HEURISTINEN ETSINTÄ

Useita menetelmiä:

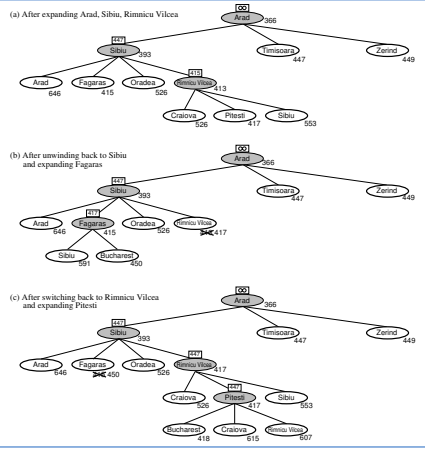
- Iteroivasti syventävä A* eli IDA*
- Rekursiivinen paras ensin -etsintä (RBFS)
- MA* (memory-bounded A*)
- SMA* (simplified memory-bounded A*)

IDA*

- Yksinkertaisin tapa rajoittaa A*-algoritmin muistin käyttöä on soveltaa ideaa iteroivasta syventämisestä. Tällöin saadaan iteroivasti syventävä A* eli IDA*-algoritmi. Sen suurin ero tavalliseen iteroivaan syventämiseen on, että syvyysetsintää ei rajoiteta tietylle syvyydelle, vaan tietyn f-arvon tasolle.
- Käytetään siis syvyysrajoitettua etsintää asettamalla alussa rajaksi juuren f-arvo ja jokaisella iteraatioaskeleella uudeksi f-rajaksi asetetaan f-arvoltaan pienimmän aiemmalla kerralla rajan ylittäneen solmun f-arvo.
- IDA* sopii hyvin tilanteisiin, joissa askeleilla on yksikkökustannukset, mutta se joutuu ongelmiin silloin kun tekojen kustannukset voivat olla reaalilukuja.

Rekursiivinen paras ensin (RBFS)

- Rekursiivinen paras ensin -etsintä (recursive best-first search) pyrkii jäljittelemään tavallisen paras ensin -etsinnän toimintaa, mutta käyttää vain lineaarista muistitilaa.
- Algoritmi toimii kuten tavallinen rekursiivinen syvyyssuuntainen etsintä, mutta merkitsee tarkasteltavaan solmuun muistiin parhaan vaihtoehtoisen löytyneen solmun f-arvoa ja jos tarkasteltavan solmun f-arvo ylittää tämän arvon, rekursiossa palataan takaisin parhaan vaihtoehtoisen reitin polulle.



- RBFS on jonkin verran tehokkaampi kuin IDA*, mutta laajentaa silti ylimääräisiä solmuja. Esim. kuvassa ensin Rimnicu Vilcea, sitten Fagaras ja jälleen Rimnicu Vilcea.
- Kuten A* myös RBFS on optimaalinen, jos heuristiikka h(n) on luvallinen. Tilavaativuus on luokkaa O(bd), mutta aikavaativuus riippuu heuristiikasta ja siitä kuinka usein etsintäsuunta vaihtuu. Verkkoetsinnässä sekä IDA* että RBFS tulevat helposti eksponentiaalisiksi koska ne eivät voi tarkistaa tilojen toistumista muuten kuin kultakin tarkasteltavalta polulta kerrallaan, joten samoja tiloja voidaan joutua tutkimaan useita kertoja.
- IDA* ja RBFS käyttävät tavallaan liiankin vähän muistia koska ne pyrkivät rajoittamaan muistitilan käyttöä vaikkei siihen olisi tarvettakaan. Jos muistia on, sitä kannattaisi hyödyntää ja rajoittaa sen käyttöä vasta kun on pakko. Näin tekevät MA* (memory-bounded A*) ja SMA* (simplified memory-bounded A*).

SMA*

- SMA* toimii muuten kuten A*, laajentaen parasta lehteä, niin kauan kuin muistitilaa riittää. Muistitilan loppuessa se ei voi lisätä uutta solmua hakupuuhun pudottamatta ensin jotakin vanhaa solmua. Silloin SMA* unohtaa huonoimman lehtisolmun, siis sen, jonka f-arvo on korkein. Unohdetun lehtisolmun f-arvo talletetaan kuitenkin sen vanhempaan siltä varalta että sinne tarvitsisikin joskus palata. Tämä tapahtuu silloin, kun kaikki muualta saadut f-arvot ovat huonompia.
- Jos kaikilla lehtisolmuilla on sama f-arvo, SMA* laajentaa viimeiseksi löydetyn ja unohtaa ensimmäiseksi löydetyn. (Tällä vältetään tilanne jossa laajennettavaksi ja unohdettavaksi valittaisiin sama solmu, esim. satunnaisvalintaa käytettäessä.) Jos käy niin huonosti että muisti loppuu kun vain yksi lehtisolmu on jäljellä, niin silloin on pakko lopettaa etsintä siihen, vaikka ratkaisuun asti ei ole vielä päästy.

- Jos ratkaisun etäisyys on muistitilan rajoissa, SMA* on täydellinen. Jos parhaan ratkaisun etäisyys on muistitilan rajoissa, SMA* on optimaalinen. Jos jokin ratkaisu löytyi muistitilan rajoissa niin silloin palautetaan paras löytyneistä.
- Jos muistia on kohtuullisesti, SMA* pystyy ratkomaan paljon mutkikkaampia ongelmia kuin A*.
- Käytännön tilanteissa SMA* onkin ehkä paras yleiskäyttöinen etsintäalgoritmi optimaalisten ratkaisujen löytämiseen. Erityisesti silloin, jos etsintäavaruus on verkko, tekojen kustannukset vaihtelevat ja solmujen laajentaminen on kallista verrattuna avoimen ja suljetun listan ylläpitoon.
- Toisaalta hyvin vaikeissa ongelmissa SMA* voi joutua hyppimään edestakaisin polulta toiselle, jolloin se tuhlaa aikaa tuomalla samoja solmuja toistuvasti muistiin.
- Vaikka aika/tila-kysymykset ovatkin vaikeita, näyttää siltä että jos halutaan optimiratkaisu, niin ainakin toista kuluu.

Etsinnän tehostaminen oppimalla

- Nyt on esitelty erilaisia etsintästrategioita, mutta olisiko mahdollista että agentti voisi dynaamisesti muunnella etsintästrategiaa ja näin oppisi etsimään tehokkaammin?
- Tämä on mahdollista. Kun agentti suorittaa etsintää *objektitason* tila-avaruudessa, etsinnän mahdollisia tiloja voidaan pitää *metatason* tila-avaruutena.
- Varsinaisen etsinnän vaiheet (puut) ovat siis metatason tiloja. Kun haku silloin tällöin joutuu tiloihin jotka eivät ole hyödyllisiä vaan joista esim. joudutaan vaihtamaan etsintäsuuntaa, metatason oppimisalgoritmi voi tehdä yleistyksiä tällaisista tilanteista ja oppia välttämään niitä.
- Tavoitteena on tällöin *yhteiskustannuksen* (total cost) minimointi, jossa tavoitellaan parasta tasapainoa ratkaisun polkukustannuksen ja etsintään käytettävien resurssien välillä.

Heuristiset funktiot

- Heuristinen funktio voidaan valita monella tavalla. Esimerkiksi 8-liukupelissä voidaan käyttää vaikkapa väärässä paikassa olevien palojen lukumäärää tai ns. Manhattan-etäisyyttä. Molemmat ovat luvallisia, sillä ne eivät yliarvioi todellista kustannusta. (Katso kirjan kuva 4.8.)
- Heuristiikan valinta vaikuttaa suorituskykyyn *efektiivisen haarautumiskertoimen* kautta.
- Dominanssi: Olkoot h_1 ja h_2 luvallisia heuristisia funktioita. Jos $h_2(n) \geq h_1(n)$ kaikilla n, niin h_2 *dominoi* h_1:tä. Dominoiva heuristiikka laajentaa vähemmän solmuja ja on siksi parempi etsinnän kannalta.

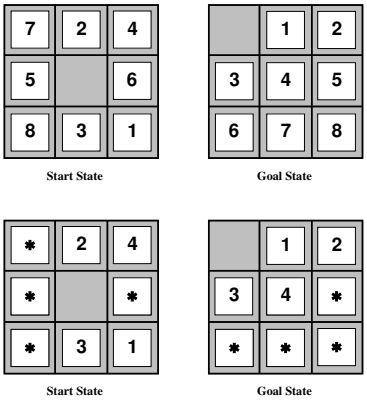
LUVALLISTEN HEURISTIIKKOJEN KEKSIMINEN

Heuristiikkojen keksiminen:

- ongelmien väljentäminen (relaxation)
- osaongelmat (subproblems)
- hahmotietokannat (pattern databases)

Heuristiikkojen oppiminen:

- induktiivinen oppiminen
- piirteet (features)

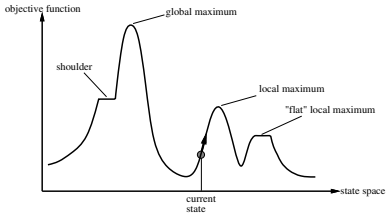


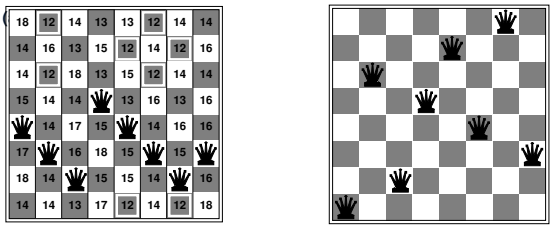
PAIKALLINEN ETSINTÄ JA OPTIMOINTI

- Jos ratkaisun löytymisreitistä (polusta) ei tarvitse välittää, vaan ainoastaan ratkaisun lopputulos on merkitsevä, voidaan käyttää *paikallisia* (local) etsintästrategioita.
- Käsitellään yhtä ainoata tilaa (tai ratkaisua) ja muunnellaan sitä paikallisesti toisenlaiseksi → naapuruston (neighbourhood) idea.
- Haussa kuljettuja polkuja ei tarvitse varastoida, joten muistitilan tarve on vähäinen, vain vakio.
- Yleensä käsitellään täydellisiä tiloja (complete state) kuten kahdeksan kuningattaren asetelmia tai kokonaisia kaupunkireittejä, joihin sitten tehdään pieniä paikallisia muutoksia.
- Löytävät hyviä ratkaisuja suurissakin etsintäavaruuksissa, joissa systemaattiset hakumenetelmät eivät ole käyttökelpoisia.

Tärkeitä käsitteitä:

- optimointiongelmat
- kohdefunktio (objective function)
- ”tilamaisema” (state space landscape)
- paikallinen/lokaali optimi (maksimi/minimi)
- globaali optimi (maksimi/minimi)





(a) **(b)**

(a) Tila, jonka arvo (uhkausten lukumäärä) on 17. Ruutuihin on merkitty mahdollisten seuraajatilojen (saman sarakkeen kuningattaren siirtäminen kyseiseen ruutuun) arvot.

(b) Paikallinen minimi, jonka arvo on 1.

VUORIKIIPEILYALGORITMIT

- Vuorikiipeilyalgoritmit (hill-climbing algorithms) ovat ahneita paikallisia etsintämenetelmiä: valitaan aina nykyisen tilan naapureista se, jolla kohdefunktion arvo on korkein.
- Jos tilaa ei voi enää parantaa, haku päättyy.
- Tavanomainen vuorikiipeily juuttuu helposti paikalliseen optimiin (joka ei ole globaali) tai tasangolle. Tämän korjaamiseksi on kehitetty useita variantteja:
  - sivuttaissiirtymien salliminen → varottava juuttumista
  - stokastinen vuorikiipeily: valitaan satunnaisesti nykytilannetta parantavista naapureista
  - ensimmäisen valinnan vuorikiipeily (first-choice hill climbing): tutkitaan naapureita satunnaisesti ja valitaan ensimmäinen joka on nykyistä parempi
  - vuorikiipeily satunnaisella uudelleenaloituksella (random restart): aloitetaan uudestaan satunnaisesta alkutilasta kunnes ratkaisu löytyy → täydellisyys


```
function HILL-CLIMBING( problem) returns a state that is a local maximum
  inputs: problem, a problem
  local variables: current, a node
                   neighbor, a node

  current ← MAKE-NODE(INITIAL-STATE[problem])
  loop do
      neighbor ← a highest-valued successor of current
      if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
      current ← neighbor
  end
```


SIMULOITU JÄÄHDYTYS

- Jos sallitaan vain nykytilan parantaminen, juututaan paikalliseen optimiin eikä haku ole täydellinen
- Toisaalta *satunnaiskulku* (random walk) on täydellinen mutta tehoton etsintämenetelmä.
- Yksi ratkaisu on sallia tilanteen huononeminen joskus: *Simuloitu jäähdytys* (simulated annealing) sallii siirtymisen huonompaan vaihtoehtoon mutta sallimisen todennäköisyys pienenee eron kasvaessa.
- Lisäksi pidetään yllä "lämpötilaa", joka kuvaa tilannetta huonontavien siirtymien todennäköisyyttä. Lämpötila pienenee ajan myötä joten alussa sallitaan suuriakin huononnuksia, mutta lopulta vain hyvin pieniä.
- Jos lämpötilan jäähtyminen on riittävän hidasta, niin menetelmä päätyy globaaliin optimiin ykköstä lähestyvällä todennäköisyydellä.

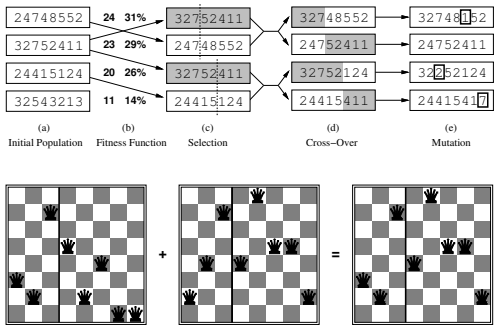
```
function SIMULATED-ANNEALING( problem, schedule) returns a solution state
  inputs: problem, a problem
          schedule, a mapping from time to "temperature"
  local variables: current, a node
                   next, a node
                   T, a "temperature" controlling prob. of downward steps

  current ← MAKE-NODE(INITIAL-STATE[problem])
  for t ← 1 to ∞ do
      T ← schedule[t]
      if T = 0 then return current
      next ← a randomly selected successor of current
      ΔE ← VALUE[next] – VALUE[current]
      if ΔE > 0 then current ← next
      else current ← next only with probability e^(Δ E/T)
```


PAIKALLINEN KEILAHAKU

- Paikallinen keilahaku (local beam search) pitää yllä k kappaletta tarkasteltavia tiloja, joiden kaikki seuraajatilat generoidaan.
- Jos mikään niistä ei ole maalitila, valitaan seuraajista k parasta laajennettaviksi seuraavalla kierroksella.
- Ei ole sama asia kuin vuorikiipeily satunnaisella uudelleenaloituksella, koska rinnakkaiset haut riippuvat toisistaan → haku ohjautuu nopeasti lupaaville alueille jolloin huonot polut hylätään. Tämä voi olla ongelmakin.
- *Stokastinen keilahaku* toimii muuten samoin, mutta ei valitakaan parhaita vaan käytetään satunnaisuutta siten että paremmilla on huonompia suurempi todennäköisyys tulla valituksi.

GENEETTISET ALGORITMIT

- Geneettisissä algoritmeissa (genetic algorithms) on samantapainen idea kuin edellisessä, mutta jatkoon ei valita tilojen seuraajia vaan tilaparien yhdistelmiä.
- Terminologia biologiasta: puhutaan *populaatiosta* ja sen *yksilöistä*, jotka ovat yleensä merkkijonoja. Yksilöiden hyvyyttä mitataan *kelpoisuusfunktiolla* (fitness function) ja niiden todennäköisyys päästä *valinnassa* jatkamaan seuraavaa sukupolvea riippuu niiden kelpoisuudesta. Yhdistämistä kutsutaan *risteytykseksi* (crossover) ja se tapahtuu valitsemalla kummastakin merkkijonosta jokin katkaisukohta josta niiden loppuosat vaihdetaan keskenään.
- Lopuksi jälkeläiset saattavat pienellä todennäköisyydellä joutua *mutaation* kohteeksi, jossa jokin merkki vaihdetaan toiseksi. Tämän jälkeen sama toistuu uudella sukupolvella.
- Toimivuus edellyttää sopivaa merkkijonokoodauksen valintaa ja risteytys- ja mutaatio-operaatioiden rajoittamista sellaiseksi että jälkeläiset ovat mielekkäitä.



PAIKALLINEN ETSINTÄ JATKUVISSA AVARUUKSISSA

- Jos kohdefunktio on jatkuva-arvoinen, naapuruston idea ei enää suoraan toimi.
- Jos tila-avaruus voidaan *diskretisoida*, aiemmat menetelmät toimivat.
- Toinen mahdollisuus on soveltaa vuorikiipeilymenetelmiä käyttäen hyväksi kohdefunktion f *gradienttia* $\bigtriangledown f$, joka on vektori joka kertoo jyrkimmän nousun suunnan ja suuruuden.
- Esimerkiksi kuuden jatkuva-arvoisen muuttujan funktiolle $f(x_1, y_1, x_2, y_2, x_3, y_3)$ gradientti

$$\nabla f = (\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial y_1}, \frac{\partial f}{\partial x_2}, \frac{\partial f}{\partial y_2}, \frac{\partial f}{\partial x_3}, \frac{\partial f}{\partial y_3})$$

- (Lokaalissa) maksimissa gradientti on nolla. Yleensä yhtälöä ei voi ratkaista globaalisti, mutta sitä voidaan hyödyntää lokaalisti.

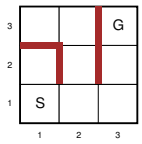
- Jyrkimmän nousun vuorikiipeilyä voi tehdä päivittämällä nykyistä tilaa $\vec{x} \leftarrow \vec{x} + \alpha \nabla f(\vec{x})$, missä α on pieni vakio.
- Menetelmä on herkkä α:n arvolle, sillä jos se on liian pieni, etsintä kestää kauan, ja jos se on liian suuri, maksimikohta ohitetaan helposti.
- Jos kohdefunktio ei ole derivoituva, voidaan yrittää muodostaa *empiirinen gradientti* kokeilemalla kohdefunktion arvoa nykyisen tilan lähistöllä tekemällä hyvin pieniä muutoksia eri koordinaattien arvoihin. Tämä vastaa aiemmin mainittua diskretisointia.
- Joskus kohdefunktiolle ja ratkaisuille voidaan asettaa *rajoitteita*. Esimerkiksi *lineaarisessa ohjelmoinnissa* kohdefunktio on lineaarinen funktio ja ratkaisujen täytyy toteuttaa tietyt lineaariset epäyhtälöt jotka muodostavat konveksin alueen. Tällöin ratkaisu voidaan tehdä polynomisessa ajassa muuttujien määrän suhteen.

ONLINE-ETSINTÄ JA TUNTEMATTOMAT YMPÄRISTÖT

- Aiemmin käsiteltiin offline-tyyppisiä etsintämenetelmiä, jotka laskevat ratkaisun kokonaan valmiiksi ennen kuin suorittavat mitään tekoja todellisessa maailmassa.
- Nyt käsitellään online-menetelmiä, jotka vuorottelevat toimintojen etsimistä ja suorittamista. Saatuaan havaintoja seurauksista online-toimijat jatkavat etsintää.
- Online-etsintä sopii hyvin dynaamisiin tai semidynaamisiin ympäristöihin, joissa tilanne (tai suoritusfunktion arvo) voi muuttua etsinnän aikana. Samoin stokastiset ympäristöt, joissa tekojen vaikutukset vaihtelevat, sopivat online-etsinnälle.
- Online-etsintä on välttämätöntä *tutkimisongelmissa* (exploration problem) joissa agentti ei tunne ympäristön tiloja ja tekojen seurauksia vaan sen täytyy tunnustella ympäristöään ja kokeilla toimintojaan. (Esim. robottinavigointi tuntemattomassa ympäristössä.)

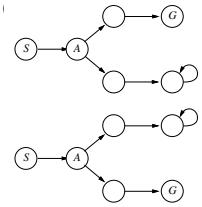
ONLINE-ETSINTÄONGELMAT

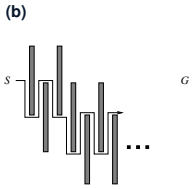
- Online-etsintäongelmiksi kutsutaan sellaisia ongelmia, joita ei voi ratkaista ainoastaan laskemalla etukäteen vaan ainoastaan vuorottelemalla etsintää ja kokeiluja.
- Oletetaan, että toimija tietää ainoastaan
  - mahdolliset teot, ACTIONS(s)
  - tekojen kustannukset, $c(s, a, s')$ (ei voi käyttää ennen kuin s' tunnetaan)
  - maalitestin, GOAL-TEST(s)
- Toimija ei siis tiedä toimintojensa seuraajatilaa ennen kuin se suorittaa toiminnan. Voi olle esim. ettei agentti seuraavan sokkelossa tiedä että toiminto Up ruudussa (1,1) johtaa ruutuun (1,2) ja vaikka se kokeilisi sitä, niin se ei tiedä, että toiminto Down vie sen takaisin ruutuun (1,1). Joskus tosin tällaiset suhteet toimintojen välillä voivat olla tunnettuja, jolloin ainoastaan ympäristö on tuntematon.



- Oletetaan nyt, että toimija tunnistaa tilan, jossa se on ollut aikaisemmin ja että teot ovat deterministisiä. Toimijalla voi lisäksi olla käytössään heuristinen funktio h, joka kertoo sen etäisyyden maalitilasta. (Yllä esim. Manhattan-etäisyys.)
- Tehtävänä voi olla joko koko ympäristön kartoittaminen tai päämäärän löytäminen kustannuksia minimoiden. Kustannuksena voi olla esim. koko kuljetun reitin pituus. Tätä verrataan tavallisesti optimaalisen reitin pituuteen (joka siis voitaisiin saavuttaa jos alue tunnettaisiin).

- Agentin löytämän ratkaisun ja optimaalisen ratkaisun suhdetta kutsutaan *kilpailusuhteeksi* (competitive ratio). Suhde on joskus ääretön: esimerkiksi jos teot eivät ole peruutettavia, online-etsintä voi päätyä umpikujaan. Mikään etsintäalgoritmi ei voi välttää umpikujia kaikissa tila-avaruuksissa: Kuvitellaan vastustaja joka voi sijoittaa maalit ja umpikujat haluamallaan tavalla (adversary argument).





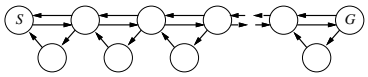
- *Turvallisesti tutkittavia* (safely explorable) alueita ovat sellaiset, joissa umpikujiin ei voi joutua, vaan jokaisesta saavutettavasta tilasta päästään johonkin päämäärätilaan.
- Erityisesti sellaiset tila-avaruudet, joissa teot ovat peruutettavia, ovat turvallisesti tutkittavia, esimerkiksi 8-liukupeli ja sokkelot.
- Tällaisissakaan ympäristöissä ei voi antaa yleistä rajaa kilpailusuhteelle vaan etsintä voi olla mielivaltaisen vaikeaa. Katso edellisen kuvan kohta (b).

ONLINE-ETSINTÄAGENTIT

- Etsinnän ja toiminnan vuorottelun seurauksena online-algoritmit ovat hyvin erilaisia kuin offline-algorimit. Esimerkiksi A* voi laajentaa solmuja eri puolilta etsintäavaruutta, mutta ympäristöön sijoittuneen online-agentin on tehtävä tekoja siinä paikassa (siinä kohdassa etsintäavaruutta), jossa se kulloinkin on. Siksi esimerkiksi syvyysuuntainen etsintä sopii paremmin online-agentin etsintäalgoritmiksi, koska solmuja laajennetaan paikallisesti (kunnes peruutetaan).
- Esim. aiemmassa sokkelossa voidaan joutua kulkemaan lähes kaikkia kaaria pitkin edestakaisin. Tutkittaessa tämä on hyväksyttävää, mutta reitin löytämisessä kilpailusuhde voi kasvaa mielivaltaisen huonoksi.
- Peruutus täytyy tehdä fyysisesti ja tekojen olisi hyvä olla kumottavissa. Jotkut algoritmit toimivat ilmankin mutta kilpailusuhteilla ei ole ylärajaa.

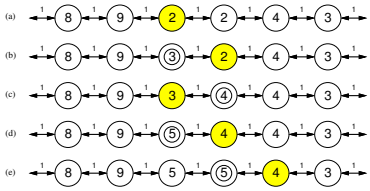
PAIKALLINEN ONLINE-ETSINTÄ

- Vuorikiipeilyalgoritmikin etenee hakuavaruudessa paikallisesti ja koska se pitää vain yhden tilan muistissa, se on jo sellaisenaan online-algoritmi. Yksinkertaisessa muodossaan se ei kuitenkaan ole kovin hyödyllinen, koska se jättää agentin paikalliseen maksimiin eikä satunnaisia uudelleenaloituksia voida tehdä.
- Niiden sijasta voidaan käyttää *satunnaiskävelyä* (random walk). Siinä kuljetaan satunnaiseen suuntaan (mahdollisesti) preferoiden niitä tekoja, joita ei vielä ole kokeiltu. Näin voidaan taata että maali joskus löytyy, jos etsintäavaruus on äärellinen (tai korkeintaan kaksiulotteinen ristikko tai hila (grid)). Tämä voi kuitenkin viedä eksponentiaalisen ajan, kuten esim. kuvassa:



LRTA*

- Parempi idea on ottaa käyttöön muisti, jossa pidetään arviota kunkin käydyn tilan etäisyydestä maaliin $H(n)$. Tämä vastaa heuristista funktiota h mutta sitä päivitetään etsintäavaruutta tutkittaessa, kuten seuraavassa kuvassa:



- Prosessi "tasoittaa" (flatten) paikallisen minimin, minkä jälkeen etsintää voidaan taas jatkaa normaalisti.

- Tällaista menetelmää kutsutaan nimellä "oppiva tosiaikainen A*" (LRTA*) (learning real-time A*).
- LRTA* löytää varmasti maalin missä tahansa äärellisessä turvallisesti tutkittavassa ympäristössä. Toisin kuin A* se ei kuitenkaan ole täydellinen äärettömissä etsintäavaruuksissa vaan voi jäädä äärettömään haaraan.
- Pahimmassa tapauksessa alueen tutkiminen voi vaatia $O(n^2)$ askelta kun n on tilojen määrä, mutta yleensä algoritmi suoriutuu nopeammin.

OPPIMINEN ONLINE-ETSINNÄSSÄ

- Toimija oppii "kartan" ympäristöstä tallettamalla tekojensa seuraukset. LRTA*:ssä toimija oppii lisäksi eri tilojen arvot, jotka konvergoituvat kohti todellisia arvoja, minkä jälkeen optimaalisia toimintoja voidaan tehdä yksinkertaisella vuorikiipeilyllä. Jotta agentti voisi oppia toimintojensa yleisiä vaikutuksia ja niiden välisiä suhteita, kuten sen että Up kasvattaa y-koordinaattia ja Down vähentää sitä, tarvitaan
  1. tapaa, jolla tällaisia asioita voidaan esittää: tietämyksenesityskieltä, ja
  2. algoritmeja, joilla tällaisia yleisiä sääntöjä voidaan oppia havainnoista.