4. Lisää ongelmanratkaisumenetelmiä

Jatketaan luvussa 3 esiteltyä etsintäteemaa sellaisissa ongelmissa joissa

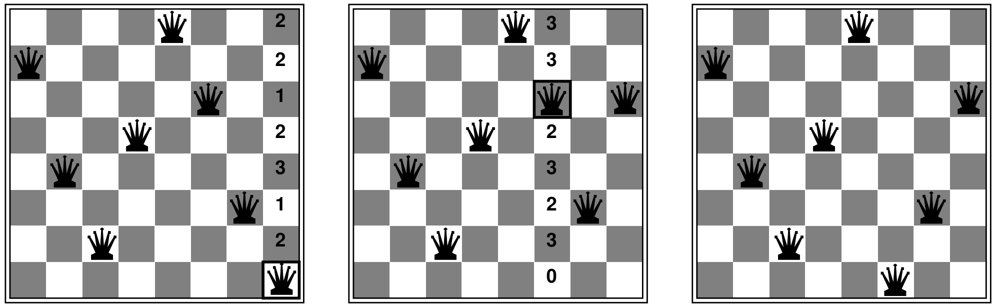
1. vain itse ratkaisu kiinnostaa, ei siihen johtanut tekojono
2. maaliehto ei enää olekaan "musta laatikko", vaan jakautuu erillisiin osaehtoihin, jotka voidaan hoitaa yksi kerrallaan
3. on mukana etsijän lisäksi pahantahtoinen vastustaja, kuten peleissä
4. on mukana etsijän vastustajien lisäksi välinpitämätön "luonto", kuten noppa onnenpeleissä.

Tällöin käytetään erilaisia etsintämenetelmiä kuin aiemmin:

1. Iteratiivisia parannusmenetelmiä.
2. Rajoitteiden levitystä.
3. Etsintää pelipuissa.
4. Etsintää todennäköisyyksin varustetuissa pelipuissa.

4.1. Iteratiiviset parannusmenetelmät [RN95, luku 4.4] [Nil98, luvut 11.1-2]

Esimerkiksi 8 kuningattaren (shakki)ongelmassa ei ratkaisun saavuttanutta tekopolkua kysytä; vain lopputuloksen laatu ratkaisee.



Kuva 38: 8 kuningattaren ratkaiseminen [RN95, kuva 4.16]. (Ruudun numero kertoo moniko sitä uhkaa.)

Tällöin on usein käytännöllisintä *tehdä alkutilaan toistuvasti pienia paikallisia korjauksia* ja toivoa, että näin päästäisiin ratkaisuksi kelpaavaan tilanteeseen.

Esimerkiksi VLSI-piirien optimoinnissa jo alkutila on kelvollinen piiri, jonka komponenttien paikkoja aletaan vaihdella, ja näin yritetään minimoida tarvittavien johtojen pituuksia.

Nämä menetelmät yrittävät *optimoida (minimoida/maksimoida) kohdefunktiota (evaluation function)*, joka mallintaa etäisyyttä ratkaisuun tai nykyisen ratkaisun hyvyyttä.

Menetelmien periaate selviää kuvasta 39: piirretään kohdefunktion $\textsc{Value}\colon \mathbb{R}^2 \mapsto \mathbb{R}$ kuvaaja, ja sijoitetaan nykyinen tila sinne.

Saadaan alppimaisemassa vaeltaja, joka yrittää nousta vuoren huipulle (tai minimointitehtävässä laskeutua laakson pohjalle), vaikka hän. . .

- on niin *likinäköinen*, että näkee vain yhden askeleen verran ympärilleen
- kärsii Korsakovin syndroomasta eli *ei pysty painamaan mitään asioita muistiinsa*.

Yksinkertainen strategia on katsella ympärilleen tuon yhden askeleen verran, ja ottaa niistä *se askel, joka vie korkeimmalle*.

Näin saadaan *vuoristokiipeilyalgoritmi 7 (Hill-Climbing Algorithm)*. Se pysähtyy, kun ylöspäin ei enää päästä, ja ilmoittaa lötäneensä huipun.

Minimointitehtävissä tätä (jyrkimmin alaspäin menevää) menetelmää kutsutaan *gradienttilaskeutumiseksi (Gradient Descent)*.

Koulumatematiikassa funktion $f\colon \mathbb{R} \mapsto \mathbb{R}$ *derivaatta* f' (alias $\mathbf{D}f$, $f\frac{x}{dx}$, . . .) pisteessä $x_0 \in \mathbb{R}$ oli geometrisesti:

- Piirrä funktion $y = f(x)$ kuvaaja κ 2-ulotteisessa (suorakulmaisessa) koordinaatistossa.
- Piirrä kuvaajalle κ se *tangenttisuora* τ joka kohtaa kuvaajan pisteessä x_0.
- Derivaatan arvo $f'(x_0)$ on *tangenttisuoran* τ *kulmakerroin*. . .
- eli *funktion* $f(x)$ *hetkellinen muutosnopeus kohdassa* x_0.
  Esimerkiksi fysiikassa (hetkellinen)
  - nopeus on paikan
  - kiihtyvyys on nopeuden

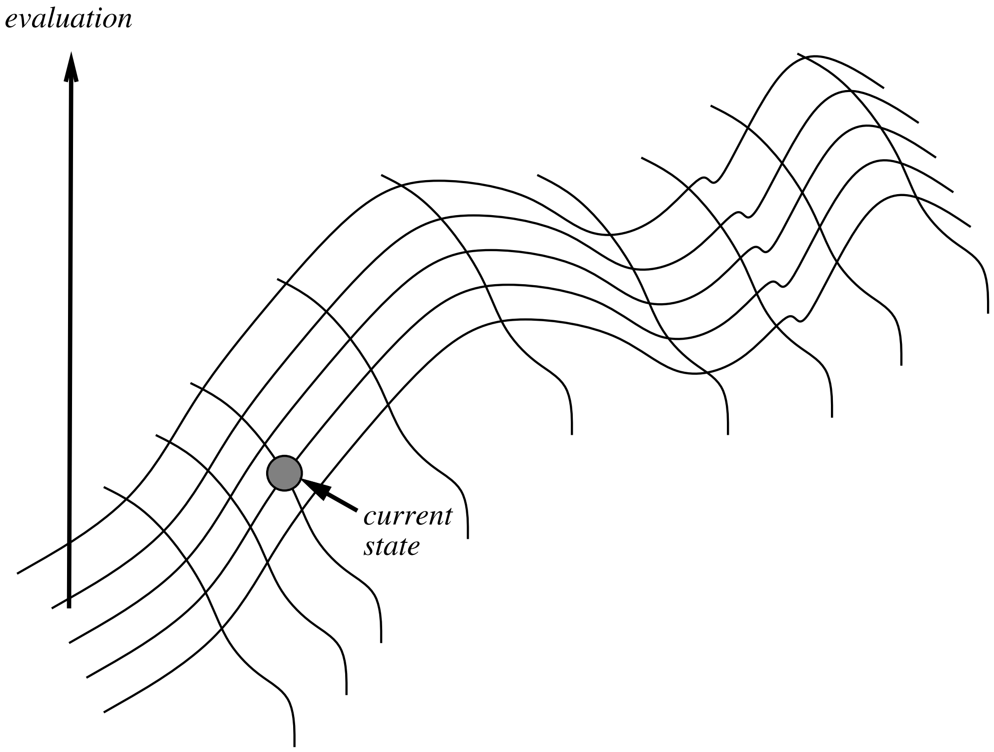
  derivaatta ajan suhteen.

Matematiikan kurssin "Differentiaali- ja integraalilaskenta II" gradientti on sama idea m-ulotteisten avaruuksien funktioille $g\colon \mathbb{R}^m \mapsto \mathbb{R}$.

3-ulotteisessa kuvassa 39 on maiseman $z = g(x, y)$ kuvaaja K.

Funktion arvon z hetkellinen muutosnopeus ja -suunta saadaan nyt siitä *tangenttitasosta* T, joka kuvaajalla K on nykyisessä pisteessä $\mathit{current\ state} = \langle x_0, y_0 \rangle$.

Maisemafunktion g gradientti on *tangenttitason* T *normaalivektori*.



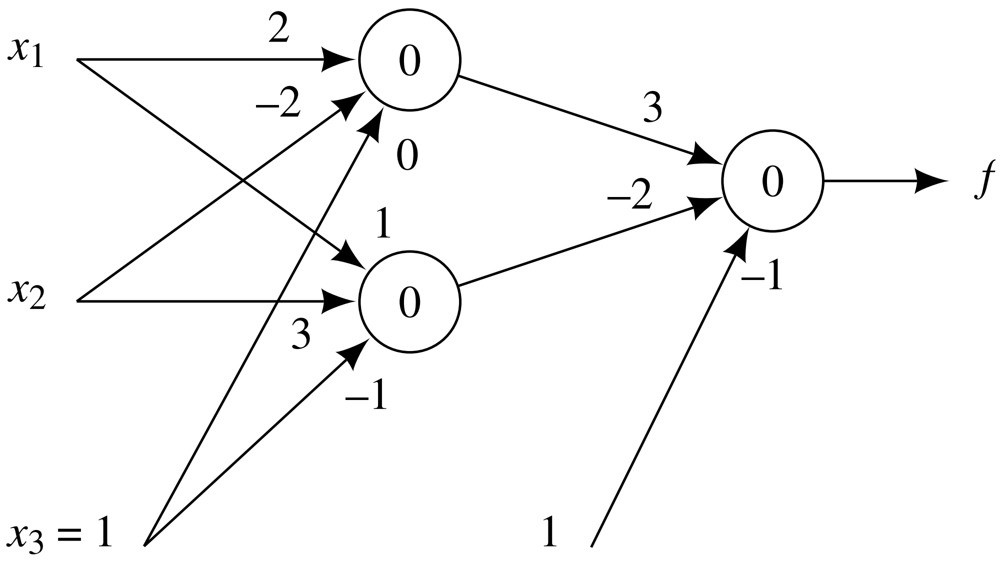
Kuva 39: Optimoitavan funktion maisemaa [RN95, kuva 4.13].

```
function HILL-CLIMBING(problem) returns a solution state
  inputs: problem, a problem
  static: current, a node
          next, a node

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

Algoritmi 7: Vuoristokiipeily [RN95, kuva 4.14].

Gradienttilaskeutumisella voi siis suorittaa etsintää *jatkuvissa* avaruuksissa kuten reaalilukuja sisältävissä ongelmissa. (Luvun 3 mentelmät olivat diskreettejä.)



Kuva 40: Pieni monitasoinen hermoverkko [Nil98, kuva 3.6].

Luvussa 1.5 mainittu monisoluisten hermoverkkojen oppimisalgoritmi on gradienttilaskeutumisen eräs sovellus:

- Perustehtävässä annetaan
  - hermoverkko H jonka kaarien (reaali)painoja voi muutella
  - esimerkkiaineisto A joka verkon H pitää oppia.
    Aineisto on valmis lista pareja ⟨annettu syöte, haluttu tulos⟩.
- Minimoitava funktio g mittaa sitä *virhettä*

  $$|H(\text{annettu syöte}) - \text{haluttu tulos}|$$

  jonka nykyinen H aineistosta poimitulla esimerkillä tekee.
  Yleensä valitaan mitaksi virheen neliö.

- Maisema $\mathcal{M}$ koostuu kaikista mahdollisista H:
  Jokainen muuttuva kaaripaino on oma ulottuvuutensa.
- Johdetaan gradientin lauseke funktio(i)lle $f_a : \mathcal{M} \mapsto \mathbb{R}$ joka kertoo, miten kaaripainoja pitää päivittää, kun halutaan muuttaa nykyinen verkko sellaiseksi, joka muistaa (myös) seuraavan esimerkin a.
  Esimerkki a on funktiossa ikään kuin tuntematon vakio.
- Opettamisessa
  1. valitaan jokin esimerkki
  2. muistetaan se päivittämällä kaaripainoja gradienttisäännöllä

  kunnes jäljelle jää enää pieni virhe.

Vuoristokiipeilyalgoritmilla 7 on kolme tunnettua ongelmamaastoa:

Paikallinen huippu (Local Maximum) on pieni ympäristöään korkeampi mäennyppylä.
Vaeltaja kiipeää sille luullen päässeensä koko vuoren huipulle, koska sen ympärillä on pelkkää alamäkeä niin pitkälle kuin vaeltajan lähes sokea silmä kantaa.

Tasanko (Plateau) on alue, jolla korkeus ei vaihtele niin pitkälle kuin silmä kantaa.
Tasangolla vaeltajamme hoipertelee *satunnaiskävelyn (random walk)*.

Harjanne (Ridge) viettää sekä "oikealle" että "vasemmalle" jyrkästi alaspäin.
Jos vaeltajan askelvalikoimasta puuttuu juuri jyrkänteen harjaa "eteenpäin" vievä suunta, niin vaeltaja kulkee epätaloudellisesti hyppien rinteen puolelta toiselle.

Kaikissa tapauksissa vaellus saavuttaa sellaisen maaston, jossa etenemistä ei enää tapahdu tai se on tuskallisen hidasta.

Yksi tapa yrittää välttää näitä ongelmia on *aloittaa satunnaisesti (Random-Restart)* uusi vaellus jostakin toisesta etsintäavaruuden satunnaisesta kohdasta sen jälkeen kun edellinen on pysähtynyt, ja kokeilla löytääkö se aiempaa korkeamman huipun.

Milloin satunnaisen aloittamisen voi lopettaa?

- Kun vaelluksia on tehty ennalta kiinnitetyt L kappaletta.
- Kun uusia entistä korkeampia huippuja ei enää ole löytynyt viimeisten L vaelluksen aikana.

Optimiratkaisun löytyminen on

- todennäköistä kun L on "riittävän suuri"
- nopeaa jos maaston huippuja on vähän ja ne erottuvat selvästi ympäristöstään.

Vaikeiden etsintäongelmien maasto

- sisältää eksponentiaalisen monta paikallista huippua mihin takertua ja mitä väistellä
- eli on piikikäs kuin siilin selkä.

Simuloitu jäähdytys/mellotus (Simulated Annealing) tarjoaa systemaattisemman tavan väistellä paikallisia huippuja.

Simuloidussa jäähdytyksessä tarkastellaan joka hetki *satunnaista* mahdollista askelta $next$:

- Jos $next$ veisi ylöspäin, niin otetaan se.
- Jos $next$ veisikin *alaspäin*, niin *arvotaan* otetaanko se vai jäädäänkö paikoilleen.

  Tässä arvonnassa askeleen $next$ todennäköisyys on sitä pienempi mitä
  - suurempi on vastaava pudotus ΔE, ja
  - mitä kauemman aikaa t vaellus on jo kestänyt.

Silloin vaelluksen

alussa pyritään väistelemään paikallisia huippuja laskeutumalla usein alaspäin

lopussa on todennäköisesti päädytty jonkin korkean huipun lähistölle, joten aletaan yrittämään useammin sitä ylöspäin.

Siirtyminen alkuvaiheesta loppuvaiheeseen on *asteittainen*.

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  inputs: problem, a problem
          schedule, a mapping from time to "temperature"
  static: current, a node
          next, a node
          T, a "temperature" controlling the probability 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^(ΔE/T)
```

Algoritmi 8: Simuloitu jäähdytys [RN95, kuva 4.15].

Asteittaiseen menetelmään päädyttiin *fysiikan* kautta:

1. Lämmitetään kiinteää rautamöykkyä R sellaiseen korkeaan lämpötilaan T, jossa se sulaa hyvin juoksevaksi nesteeksi.

   Tehdään tämä upottamalla R sitä itseään paljon suurempaan lämpöhauteeseen, joka on lämmitetty lämpötilaan T.
2. Silloin möykyn R jokainen atomi a pääsee liikkumaan (melko) vapaasti.
3. "Luonnonlakien mukaan" kukin atomi a on satunnaisessa lämpöliikkeessä, jossa atomin a oma energiatila voi

   nousta ottamalla lämpöenergiaa hauteesta (joka ei moisesta kylmene)

   laskea luovuttamalla lämpöenergiaa hauteeseen (joka ei moisesta lämpene).

Energiatilan laskeminen on nousemista todennäköisempää.

4. Näin koko R todennäköisesti "rauhoittuu" aikanaan tilaan, jossa valtaosa sen atomeista a on energiaminimissään.

   Tämä energiaminimi riippuu nykyisestä lämpötilasta T.
5. Jäähdytetään rautaa R laskemalla hauteen lämpötilaa T hieman.

   Odotetaan taas että valtaosa sen atomeista a ehtii jälleen etsiytyä uuteen energiaminimiinsä.
6. Jäähdytystä jatketaan varovasti, kunnes R on taas kiinteää.

   Nyt sen atomit a ovat hyvin todennäköisesti hyvin lähellä kiinteän olomuodon energiaminimiään, eli raudan luontaista hilarakennetta.

Näin saadaan *meltorautaa*, jonka pitäisi olla rakenteeltaan hyvin homogeenistä sekä vapaata sisäisistä jännityksistä ja (hius)halkeamista.

Simuloitu jäähdytys saadaan tästä prosessista siten, että kohdefunktion vastaluku $-\textsc{Value}(R)$ kertoo möykyn R *kokonaisenergian* joka pyrkii siis minimoitumaan.

Jos. . .

- jäähdytys aloitetaan "tarpeeksi kuumasta" alkulämpötilasta, ja
- lämpötilaa lasketaan "tarpeeksi hitaasti"

niin. . .

- simuloitu jäähdytys päätyy lopuksi globaaliin maksimikohtaan todennäköisyydellä 1

mutta. . .

- "todennäköisyydellä 1" ei ole sama asia kuin "varmasti", ja
- vastaava *jäähdytysstrategia* (jonka määrää algoritmin 8 funktio $schedule$) on eksponentiaalisen hidas.

Simuloitu jäähdytys on osoittautunut *käytännössä tehokkaaksi optimointimenetelmäksi* varsinkin silloin kun ongelma on hyvin suuri, esimerkiksi mutkikkaiden VLSI-piirien optimoinnissa.

Eräänlainen iteratiivinen parannusmenetelmä ovat myös *geneettiset algoritmit (Genetic Algorithms, GA)*:

- Niissä simuloidaan jäähtymisen sijasta
  - geenien sekoittumista suvullisessa lisääntymisessä
  - mutatoitumista yksilöissä.
- Simuloitu luonnonvalinta ohjaa populaatiota sukupolvi sukupolvelta soveltumaan paremmin ympäristöönsä eli ratkaistavaan ongelmaan.

Näitä menetelmiä ei kurssilla kuitenkaan käsitellä.