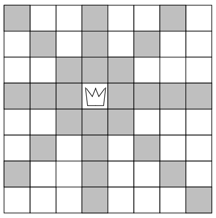
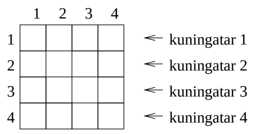
3.5 Puut ongelmanratkaisussa

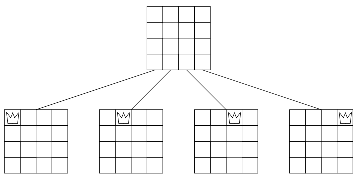
- Yksi puiden tärkeistä käyttötavoista on ongelmanratkaisussa tapahtuvan laskennan etenemisen kuvaaminen
- *kahdeksan kuningattaren ongelma:* miten voimme sijoittaa shakkilaudalle 8 kuningatarta sitten että ne eivät uhkaa toisiaan?
- kuningatar uhkaa samalla rivillä, sarakkeella sekä diagonaalilla olevia ruutuja



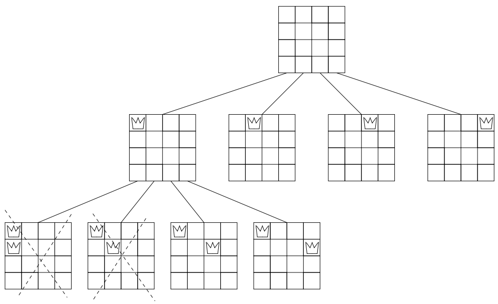
- yleistetty version ongelmasta: miten saamme sijoitettua n kuningatarta $n \times n$ -kokoiselle shakkilaudalle
- tarkastellaan konkreettisemmin tapausta missä $n = 4$
- selvästikin jokaisella rivillä täytyy olla tasan 1 kuningatar:



- etsitään oikea kuningatarasetelma systemaattisesti
  - aloitetaan tyhjältä laudalta
  - tämän jälkeen asetetaan kuningatar riville 1
  - neljä eri mahdollisuutta:

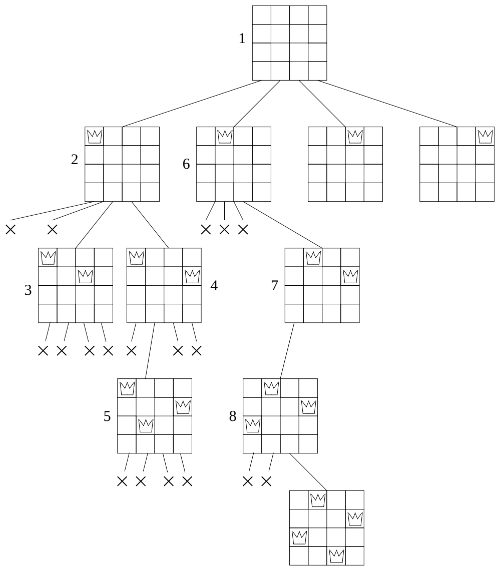


  - seuraavaksi tarkastellaan miten kuningattaret voidaan asettaa riville 2
  - aloitetaan vasemmanpuoleisesta 1 rivin valinnasta



  - huomaamme että olemme muodostamassa puuta mikä kuvaa erilaisia ratkaisumahdollisuuksia

  - kaksi vasemmanpuoleisinta yritystä ovat tuhoon tuomittuja, eikä enää kannata tutkia mitä niissä haaroissa tapahtuu
  - seuraavassa ratkaisun löytymiseen asti piirretty ratkaisupuu:



- kuvassa puun solmut on numeroitu *esijärjestyksessä*, ja kahdeksas solmu on siis ratkaisua vastaava pelitilanne

- kun laudan koko n kasvaa, tulee puusta varsin suuri
- huomionarvoista on kuitenkin se että koko puun ei tarvitse olla talletettuna muistiin
- itseasiassa riittää että muistissa on ainoastaan reitti juuresta parhaillaan tutkittavaan solmuun

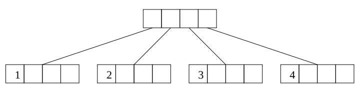
- voimme etsiä ratkaisun n:n kuningattaren ongelmaan suorittamalla ratkaisupuun läpikäynnin esijärjestyksessä ilman että ratkaisupuuta on missään vaiheessa olemassa!
- talletetaan pelitilanne $n \times n$ -taulukkoon:
  - oletetaan että pelilautaa esittää $n \times n$ -taulukko $table$
  - jos pelilaudan kohdassa (x,y) on kuningatar, on $table[x, y] = true$
  - muuten $table[x, y] = false$
- oletetaan että käytössä on funktio check(table)
  - funktio palauttaa true jos sen parametrina saama pelitilanne voidaan vielä täydentää ratkaisuksi tai on jo ratkaisu kuningatarongelmaan
  - muuten operaatio palauttaa false
- alkusi laitetaan $n \times n$ taulukon $table$ kaikkien ruutujen arvoksi false, ja kutsutaan $putqueen(table, 1)$

```
putqueen(table,row)
1  if check(table) = false then return false
2  if row = n+1 then
3      print(table)
4      return true
5  for x ←1 to n do
6      table2 ←table
7      table2[x,row] ←true
8      if putqueen(table2,row+1)=true then return true
9  return false
```

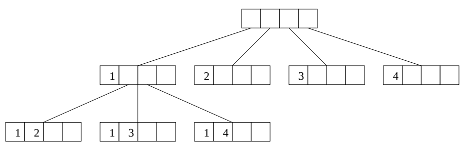
- operaation toiminta parametreilla $table, row$
  - operaatio tarkasta ensin (koodirivillä 1) edustaako $table$ pelilautaa mikä voi johtaa ratkaisuun tai on jo ratkaisu (koodirivi 2)
  - jos kyseessä on ratkaisu, tulostetaan pelilauta (rivit 3-4)
  - muussa tapauksessa tutkitaan kaikki tavat asettaa kuningatar riville row
  - luodaan uusi asetelma tauluun $table2$ ja rekursiivinen kutsu (koodirivi 8) tarkastaa johtaako tämä asetelma ratkaisuun
- algoritmi käy läpi puun mikä ei ole missään vaiheessa rakennettuna muistiin, tällaista puuta sanotaan *implisiittiseksi puuksi*
- jos puu olisi kokonaan muistissa, olisi sen koko valtava: $1 + n + n^2 + n^3 + \ldots + n^n$, koska nyt muistissa on korkeintaan puun korkeudellinen (eli n kpl) solmuja, on tilavaativuus kohtuullinen, eli $\mathcal{O}(n)$

- aikavaativuus sensijaan on suuri sillä vaikka kaikkia solmuja ei tarvisekkaan käydä läpi, kasvaa läpikäytävien solmujen määrä kuitenkin eksponentiaalisesti n:n suhteen
- käyttämästämme ongelmanratkaisutekniikasta käytetään nimitystä *branch-and-bound*, eli haaraudutaan ratkaisua etsittäessä yhä syvemmälle mutta rajataan etsintä niissä haaroissa mitkä eivät voi johtaa ratkaisuun
- samaa ratkaisustrategiaa voimme käyttää myös seuraavaan ongelmaan:

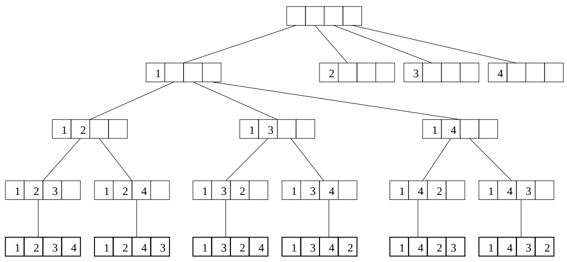
  miten generoida lukujen $1, 2, \ldots, n$ kaikki permutaatiot?
- tarkastellaan permutaatioita luvuille $1, 2, 3, 4$
  - permutaation ensimmäinen luku voi mikä tahansa yo. luvuista:



  - vasen haara jatkuisi siten että seuraava numero voi olla joku joukosta $2, 3, 4$, luku 1 on jo käytetty sillä se aloittaa permutaation



  - seuraavassa permutaatiopuu hieman pitemmälle piirrettynä



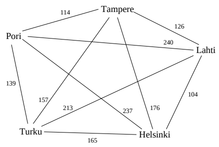
- valmiit permutaation löytyvät siis puun lehdistä, ja jos lehdet generoidaan esijärjestyksessä saadaan permutaation suuruusjärjestyksessä!
- algoritmi permutaatioiden generoimiseen
  - alustetaan n-paikkainen totuusarvoinen taulukko $used$ siten että jokaisen alkion arvo on false
  - $used$ taulukko kertoo mitkä luvuista on jo käytetty permutaatiossa
  - oletetaan että $table$ on n-paikkainen taulukko minkä alkiot ovat tyyppiä int (myös char riittää)
  - kutsutaan $generate(table, used, 1)$


```
generate(table,used,k)
1  if k=n+1 then print(table)
2  else for i ←1 to n do
3          if used[i]=false then
3              used2 ←used
4              used2[i] ←true
5              table2 ←table
6              table2[k] ←i
7              generate(table2, used2, k+1)
```

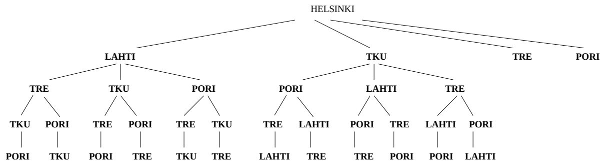
- operaation toimintaidea
  - rivillä 1 tarkistetaan onko permutaatio jo generoitu, jos on niin permutaatio tulostetaan
  - jos permutaatio ei ole vielä valmis, niin jatketaan permutaatiota kaikilla luvuilla jotka eivät vielä ole käytettyjä (rivit 2-3)
  - riveillä 3-6 käyttämätön luku lisätään permutaatioon (taulukkoon table2), merkataan luku käytetyksi (taulukkoon used2) ja rekursiivinen kutsu (rivillä 7) jatkaa kyseistä haaraa alaspäin
- erona kuningatar-ongelmaan siis tällä kertaa on se että puun generoimista ei lopeteta missään vaiheessa sillä haluamme tulostaa *kaikki* permutaatiot
- kyseessä on siis branch-and-bound ilman bound-mahdollisuutta
- edelleen tilavaativuus on varsin kohtuullinen, $\mathcal{O}(n)$, mutta aikavaativuus taas $\mathcal{O}(1 + n + n^2 + \ldots + n^n) = \mathcal{O}(n^{n+1})$

Kauppamatkustajan ongelma

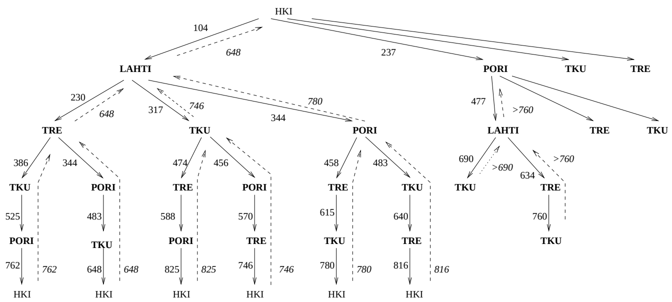
– Helsingissä asuvan kauppamatkustajan täytyy vierailla Lahdessa, Turussa, Porissa ja Tampereella



- kulujen minimointi bisneksessä on tärkeää: mikä on lyhin reitti joka alkaa Helsingistä ja päättyy Helsinkiin ja sisältää yhden vierailun kussakin kaupungissa?
- huomaamme että mahdolliset reitit ovat kaupunkien jonon Turku, Tampere, Pori, Lahti permutaatiot
- voimme siis käyttää ratkaisussa samaa periaatetta kuin permutaatioiden tulostuksessa:



- näin siis saamme systemaattisesti generoitua kaikki mahdolliset reitit
- reitin pituus kannattaa laskea heti generoinnin yhteydessä:



- palatessamme etsintäpuuta ylöspäin muistamme mikä oli parhaan kyseistä kautta kulkevan reitin pituus
- uutta reittiä etsittäessä ei kannata enää jatkaa jos tiedämme että kyseinen reitti tulee joka tapauksessa olemaan pitempi kuin paras aiemmin löydetty reitti

  esim. kuvassa Helsinki → Pori → Lahti → Turku
- koko etsintäpuun läpikäytyämme saamme tietoon lyhimmän reitin pituuden, samalla toki kannattaa merkitä muistiin minkä kaupunkien kautta reitti kulkee
- algoritmihahmotelma
  - oletetaan että kaupunkeja on n kappaletta, Helsinki on kaupunki numero 1
  - kaksiulotteinen taulukko $dist$ kertoo kaupunkien välimatkat, esim $dist[1, 3]$ sisältää Helsingin ja kaupungin numero 3 välimatkan

  - n-paikkainen totuusarvoinen taulukko $visited$ kertoo missä kaupungeissa on jo vierailtu tutkittavalla polulla
  - alustetaan $visited[i]$ = false jokaiselle i:lle
  - kutsutaan $gen(\infty, visited, 1, 1)$

```
gen(best,length,visited,cur,k)
1  if k=n−1 then return length+dist[cur,1]
2  mybest ←∞
3  for i ←2 to n do
4      if visited[i]=false then
5          vis2 ←visited
6          vis2[i] ←true
7          if length+dist[cur,i]< best then
8              newp = gen(best,length+dist[cur,i],vis2,i,k+1)
9              if newp<mybest then mybest ←newp
10             if newp<best then best ←newp
11 return mybest
```

- operaation toimintaidea
  - parametrit:

    best parhaan jo löydetyn reitin pituus

    length kuinka pitkä reitin tähän asti tutkittu osa on

    visited missä kaupungeissa on jo käyty

    cur nykyisen kaupungin numero

    k kuinka monessa kaupungissa on jo käyty
  - rivillä 1 huomataan jos koko reitti on jo generoitu

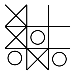
    palautetaan tässä tapauksessa reitin pituus (tähän asti käydyn osan pituus + matka Helsinkiin)

  - jos reitti ei ole vielä valmis, niin jatketaan reittiä kaikilla kaupungeilla joissa ei vielä ole käyty (rivit 3-4)
  - rekursiivinen kutsu rivillä 8 tutkii mikä on kaupungilla i jatkuvan reitin pituus
  - uutta reittiä ei kuitenkaan tutkita jos se on jo tässä vaiheessa toivottoman pitkä
  - lopulta palautetaan lyhin reitti löytyi
- jälleen kyseessä branch-and-bound strategian sovellus, tilavaativuus kohtuullinen $\mathcal{O}(n)$ mutta aikaa algoritmi vie eksponentiaalisesti tutkittavien kaupunkien määrään nähden

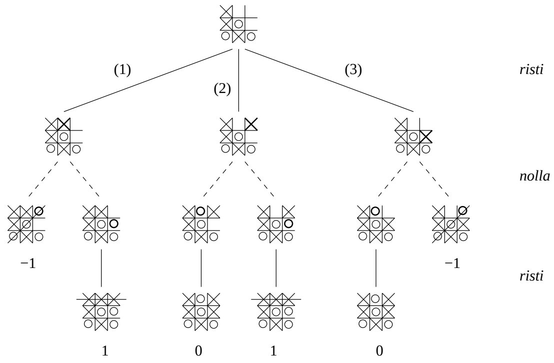
- huom: reitti Helsinki → Lahti → Tampere → Pori → Turku → Helsinki on saman pituinen myös päinvastaiseen suuntaan kuljettuna
- sama pätee jokaiselle reitille, algoritmimme siis oikeastaan tutkii jokaisen erilaisen reitin kahteen kertaan
- vaikka optimoisimme algoritmia siten että tämä epäkohta poistuisi, pysyy aikavaativuus silti eksponentiaalisena
- kauppamatkustajan ongelmalle ei tiedetä eksponentiaalisessa ajassa toimivaa parempaa ratkaisualgoritmia
- kyseessä on ns. NP-täydellinen ongelma, aiheesta enemmän Laskennan teoria -kurssilla

3.6 Pelipuu

- tietokone pelaa risti-nollaa ihmistä vastaan
- on ristin vuoro, mitä tietokoneen kannattaa tehdä seuraavassa tilanteessa?



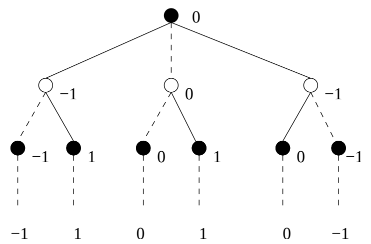
- tietokone rakentaa päätöksensä tueksi *pelipuun*



- tehtävä siis valinta kolmen mahdollisen siirron suhteen
- pelipuuhun on kirjattu auki myös kaikki mahdolliset nollaa-pelaavan siirrot, eli miten nolla voisi vastata kunkin ristin siirron jälkeen

- ja edelleen, miten peli voisi jatkua kahden siirtovuoron jälkeen
- lopputilanteita vastaaviin pelipuun lehtisolmuihin on merkattu tilanteen arvo ristin kannalta, voitto 1, tasapeli 0 ja tappio -1
- siis minkä siirron tietokone tekee?
  - valinta (1) johtaa lopulta joko nollan tai ristin voittoon
  - valinta (2) johtaa joko tasapeliin tai ristin voittoon
  - valinta (3) johtaa joko tasapeliin tai nollan voittoon

  eli järkevintä valita siirto (2), voittomahdollisuus jää mutta on varmaa ettei ainakaan hävitä
- strategiana on valita parhaan arvon (voitto 1, tasapeli 0, tappio -1) tuottava haara siten että *oletetaan että valkoinen pelaa mahdollisimman hyvin*
- seuraavassa pelipuu piirrettynä hiukan abstraktimmin



- ristin vuoroa vastaavat solmut ovat mustia ja nollan vuoroa vastaavat valkoisia

- pelipuu evaluoida lähtien lehdistä edeten juureen
  - mustat solmut ovat *max*-solmuja, ne saavat arvokseen lapsen jolla suurin arvo
  - valkoiset solmut ovat *min*-solmuja, saaden arvokseen lapsen jolla pienin arvo
- ristin siirtoa vastaa se lapsi minkä arvon juuren max-solmu perii
- kuten edellisissä esimerkissämme, ei nytkään ole tarvetta luoda pelipuuta eksplisiittisesti muistiin, riittää generoida yksi polku kerrallaan
- seuraavassa rekursiiviset operaatiot suorittavat pelipuun evaluoinnin
- aluksi kutsutaan operaatiota risti parametrina meneillään olevaa pelitilannetta vastaava solmu

risti(v)
1 **if** v:llä ei lapsia tai peli jo ohi **then**
2   **if** ristillä kolmen suora **then return** 1
3   **if** nollalla kolmen suora **then return** -1
4   **return** 0
5 mybest $\leftarrow -\infty$
6 **for** kaikilla v:n lapsilla w **do**
7   newval $\leftarrow$nolla(w)
8   **if** newval $>$ mybest **then** mybest $\leftarrow$newval
9 **return** mybest

nolla(v)
1 **if** v:llä ei lapsia tai peli jo ohi **then**
2   **if** ristillä kolmen suora **then return** 1
3   **if** nollalla kolmen suora **then return** -1
4   **return** 0
5 myworst $\leftarrow \infty$
6 **for** kaikilla v:n lapsilla w **do**
7   newval $\leftarrow$risti(w)
8   **if** newval $<$ myworst **then** myworst $\leftarrow$newval
9 **return** myworst

- rivillä 1 siis huomaamme jos siirtoja ei enää ole ja palautamme pelitilannetta vastaavan arvon (rivit 2-4)
- jos peli jatkuu vielä käymme läpi kaikki mahdolliset siirrot (rivi 6) ja evaluoimme miten peli etenee tämän siirron seurauksena (rivi 7)
- risti-operaatio palauttaa parhaan lapsensa arvon ja nolla palauttaa huonoimman lapsensa arvon
- esitetty pelipuun evaluointimenetelmä kulkee kirjallisuudessa nimellä *min-max-algoritmi*
- risti-nollassa pelipuut ovat vielä kohtuullisen kokoisia, eli siirron valinta vie kohtuullisen ajan (huom: jotkut puun haarat ovat symmetrisiä eikä "samanlaisista" tarvitse tutkia kuin yksi vaihtoehto)
- esim. Shakissa tilanne onkin aivan toinen, pelipuut ovat niin

suuria että niiden läpikäynti kokonaisuudessaan on mahdotonta

- tällöin paras mitä voidaan tehdä on generoida pelitilanteita tiettyyn syvyyteen asti
- jos pelipuuta ei voida rakentaa valmiisiin tilanteisiin (voitto, häviö, tasapeli) asti, ongelmaksi nouseekin se mikä on pelipuun lehtisolmuissa olevien pelitilanteiden arvo
- tähän on toki mahdollista kehitellä erilaisia arviointitapoja (jäljellä olevat omat/vastustajan nappulat, asetelma laudalla, ym ...)