5 RAJOITEONGELMAT

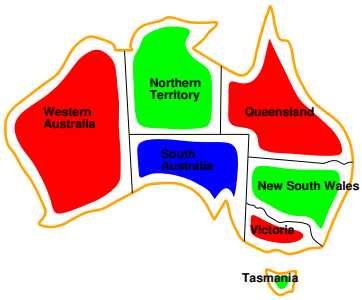
- Aiemmin etsintäongelmien tilat ovat olleet etsintäalgoritmin kannalta mustia laatikoita, joilla ei ole ollut sisäistä rakennetta. Niitä on käsitelty vain ongelmakohtaisilla rutiineilla: seuraajafunktiolla, heuristisella funktiolla ja maalitestillä.
- Nyt pyritään löytämään standardi esitysmuoto (representation) tiloille ja maalitestille.
- Päästään *rajoiteongelmiin* (constraint satisfaction problems, CSPs), joissa voidaan hyödyntää yleiskäyttöisiä heuristiikkoja.

RAJOITEONGELMIEN MUOTOILU

- Joukko muuttujia $X_1, \ldots, X_n$. Kullakin muuttujalla X_i on arvoalue D_i.
- Joukko *rajoitteita* $C_1, \ldots, C_m$. Jokainen rajoite määrittelee joillekin muuttujien osajoukolle sallitut arvoyhdistelmät.
- Ongelman tila on arvojen *sijoitus* (assignment) joillekin tai kaikille muuttujille $\{X_i = v_i, X_j = v_j, \ldots\}$.
- Sijoitusta, joka toteuttaa kaikki rajoitteet, kutsutaan *konsistentiksi* (consistent) tai *lailliseksi* (legal).
- *Täydellinen* (complete) sijoitus on sellainen, joissa kaikille muuttujille on sijoitettu arvot.
- Rajoiteongelman *ratkaisu* (solution) on täydellinen, konsistentti sijoitus.
- Joskus rajoiteongelmiin liittyy myös kohdefunktio, jota ratkaisun olisi maksimoitava.

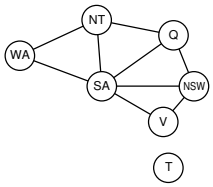
ESIMERKKI: KARTAN VÄRITYS

Muuttujat: $WA, NT.Q, NSW, V, SA, T$.
Arvoalueet: $D_i = \{red, green, blue\}$.
Rajoitteet: vierekkäiset alueet eri värisiä, siis esim. $WA \neq NT$ tai $(WA, NT) \in \{(red, green), (red, blue), (green, red), \ldots\}$



Nyt mahdollinen ratkaisu on täydellinen konsistentti sijoitus kuten esimerkiksi $\{WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = red\}$.

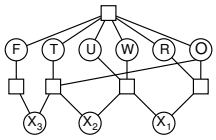
- Rajoiteongelman muotoilu standardin mukaisesti sallii seuraajafunktion ja maalitestin geneerisen toteutuksen.
- Samoin mahdollistuu yleisen ja tehokkaan heuristisen funktion käyttö, joka ei vaadi ongelmakohtaista tietämystä.
- Lisäksi *rajoiteverkon* (constraint graph) käyttö yksinkertaistaa ratkaisun etsintää, joissain tapauksissa eksponentiaalisesti.



- Jos muuttujat (lkm n) ovat diskreettejä ja niiden arvoalueet äärellisiä (koko d), niin mahdollisia täydellisiä sijoituksia on d^n. Esim. totuusarvoiset ongelmat, kuten 3SAT, joka on myös NP-täydellinen, kuuluvat näihin.
- Jos arvoalueet voivat olla äärettömiä, tarvitaan *rajoitekieltä* (constraint language) ilmaisemaan rajoitteet, esim. $\mathit{StartJob}_1 + 5 \leq \mathit{StartJob}_3$. Jos rajoitteet ovat lineaarisia, ongelmat ovat ratkeavia.
- Yleisiä ovat ongelmat, joissa arvoalueet ovat jatkuvia. Erikoistapauksissa (esim. lineaarinen ohjelmointi) ongelmia voidaan ratkaista tehokkaasti.
- Rajoitteet voivat olla *unaarisia*, *binaarisia* tai monen muuttujan välisiä, esim *AllDiff*. Jälkimmäiset voidaan ilmaista binaarisina.
- Rajoitteet ovat tavallisesti absoluuttisia, ”kovia rajoitteita”, mutta joskus käytetään myös ”pehmeitä rajoitteita”, jotka ilmaisevat minkälaisia ratkaisuita preferoidaan.

ESIMERKKI

$$\begin{array}{r} \,T\;W\;O \\ +\;T\;W\;O \\ \hline F\;O\;U\;R \end{array}$$



Muuttujat: $F, T, U, W, R, O, X_1, X_2, X_3$

Arvoalueet: $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

Rajoitteet: *AllDiff*(F, T, U, W, R, O)
$O + O = R + 10 \cdot X_1$
$X_1 + W + W = U + 10 \cdot X_2$
$X_2 + T + T = O + 10 \cdot X_3$
$X_3 = F$

- Rajoiteongelmat voidaan muotoilla etsintäongelmina:

  Alkutila: Tyhjä sijoitus

  Seuraajafunktio: Arvon asettaminen muuttujalle, jolle ei vielä ole asetettu arvoa, edellyttäen että rajoitteita ei rikota.

  Maalitesti: Tämänhetkinen sijoitus on täydellinen

  Polkukustannus: Yleensä vakiokustannus.
- Nyt voidaan käyttää tavanomaisia sokeita etsintäalgoritmeja. Syvyyssuuntainen etsintä on suosittu, koska ratkaisut ovat täydellisiä sijoituksia ja puiden korkeus on siis rajattu n:ään.
- Huomataan kuitenkin, että erilaisia tekoja on alussa nd kappaletta, sillä jokaiselle n:lle muuttujalle voidaan valita jokin d:stä arvosta. Puun seuraavalla tasolla haarautumiskerroin on $(n-1)d$ ja niin edelleen, joten puuhun saadaan $n!d^n$ lehteä. Kuitenkin erilaisia täydellisiä sijoituksia on vain d^n kappaletta.

PERUUTTAVA ETSINTÄ RAJOITEONGELMISSA

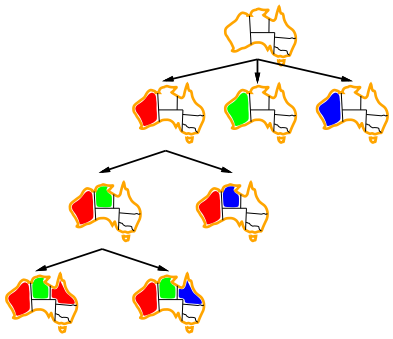
- Naiivi hakumenetelmä ei ota huomioon rajoiteongelmien *vaihdannaisuutta* (commutativity): sitä, että tekojen eli sijoitusten suoritusjärjestyksellä ei ole merkitystä. Kun tämä otetaan huomioon, lehtien määräksi saadaan d^n.
- Peruuttavassa (backtracking) syvyyssuuntaisessa haussa valitaan arvo yhdelle muuttujalle kerrallaan ja peruutetaan kun jollakin muuttujalla ei enää ole sallittuja arvoja.
- On huomattava, että koska rajoiteongelmat on esitetty standardimuodossa, algoritmi sopii sellaisenaan kaikkiin tilanteisiin eikä tarvita ongelmakohtaista alkutilaa, seuraajafunktiota eikä maalitestiä.

BACKTRACKING-ALGORITMI

function BACKTRACKING-SEARCH(*csp*) **returns** solution/failure
  return RECURSIVE-BACKTRACKING({ }, *csp*)

function RECURSIVE-BACKTRACKING(*assignment*, *csp*) **returns** soln/failure
  if *assignment* is complete **then return** *assignment*
  var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[*csp*], *assignment*, *csp*)
  for each *value* **in** ORDER-DOMAIN-VALUES(*var*, *assignment*, *csp*) **do**
    if *value* is consistent with *assignment* given CONSTRAINTS[*csp*] **then**
      add {*var* = *value*} to *assignment*
      result ← RECURSIVE-BACKTRACKING(*assignment*, *csp*)
      if *result* ≠ *failure* **then return** *result*
      remove {*var* = *value*} from *assignment*
  return *failure*

BACKTRACKING-ESIMERKKI



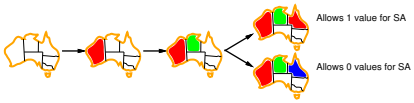
- Peruuttava syvyyssuuntainen haku on tavallisin sokea ratkaisumenetelmä rajoiteongelmille.
- Ratkaisee n:n kuningattaren ongelman, kun $n \approx 25$
- Vastaavasti kuin sokeita etsintämenetelmiä voitiin parantaa ohjaamalla hakua ongelmakohtaisen heuristisen funktion avulla, rajoiteongelmien sokeaa ratkaisua voidaan tehostaa yleisillä heuristiikoilla jotka kiinnittävät huomiota muuttujien ja arvojen käsittelyjärjestykseen, arvojen valinnan seurauksiin muiden muuttujien kannalta ja umpikujien välttämiseen arvojen valinnassa.

MUUTTUJIEN JA ARVOJEN JÄRJESTYS

- vähiten mahdollisia arvoja -heuristiikka (MRV, minimum remaining values)

- MRV + asteheuristiikka (degree heuristic)

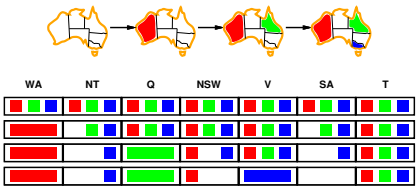
- vähiten rajoittava arvo -heuristiikka (least constraining value)



INFORMAATION VÄLITTÄMINEN RAJOITTEISSA

- Edellisiä heuristiikoita yhdistämällä voidaan ratkaista jo tuhannen kuningattaren ongelma.
- Äsken muuttujaan liittyviä rajoitteita tarkasteltiin vain kun muuttuja valittiin sijoitukseen, mutta etsintää voidaan tehostaa huomattavasti rajoittamalla mahdollisia arvoja jo aikaisemmin.
- *Eteenpäintarkistus* (forward checking): aina kun asetetaan arvo muuttujalle, poistetaan jäljelläolevien arvoalueista sellaiset arvot, jotka eivät sovi yhteen asetetun arvon kanssa. Jos joltakin muuttujalta loppuvat sallitut arvot, on peruutettava.
- Menetelmä sopii hyvin yhteen vähiten mahdollisia arvoja -heuristiikan kanssa. (MRV)

ESIMERKKI: ETEENPÄINTARKISTUS



- Muuttujalla SA ei ole enää sallittuja arvoja, joten sijoitus johtaa epäkonsistenssiin.

RAJOITTEIDEN LEVITYS

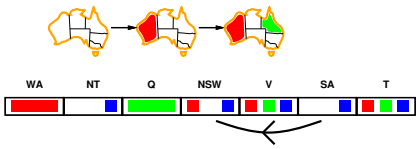
- Eteenpäintarkistus ei kuitenkaan huomaa kaikkia mahdollisia ongelmatilanteita, mikä näkyy kuvan kolmannella rivillä: WA=red, Q=green joten sekä NT:n että SA:n ainoa mahdollinen arvo on blue, mutta se ei käy, koska solmut ovat naapureita. Algoritmi kuitenkin jatkaa etsintää ja joutuu jossain vaiheessa peruuttamaan.
- *Rajoitteiden levitys* (constraint propagation) tarkoittaa arvojen asettamisen seurauksien propagointia muihin muuttujiin.
- *Kaarikonsistenssi* (arc consistency) tarkoittaa että *suunnattu* kaari A:sta B:hen on konsistentti, joss kaikille mahdollisille A:n arvoille löytyy myös jokin sopiva B:n arvo.
- Saadaan kaarikonsistenssialgoritmi, joka tarkistaa arvon kiinnittämisen jälkeen seuraukset muiden muuttujien arvojoukoille.

KAARIKONSISTENSSIALGORITMI

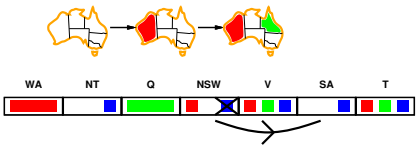
function AC-3(*csp*) **returns** the CSP, possibly with reduced domains
  inputs: *csp*, a binary CSP with variables $\{X_1, X_2, \ldots, X_n\}$
  local variables: *queue*, a queue of arcs, initially all the arcs in *csp*

  while *queue* is not empty **do**
    $(X_i, X_j) \leftarrow$ REMOVE-FIRST(*queue*)
    if REMOVE-INCONSISTENT-VALUES(X_i, X_j) **then**
      for each X_k **in** NEIGHBORS[X_i] **do**
        add (X_k, X_i) to *queue*

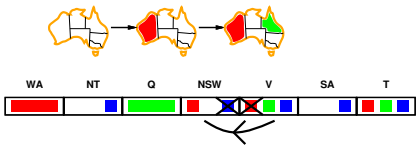
function REMOVE-INCONSISTENT-VALUES(X_i, X_j) **returns** true iff succeeds
  removed $\leftarrow$ *false*
  for each *x* **in** DOMAIN[X_i] **do**
    if no value y in DOMAIN[X_j] allows (*x*,*y*) to satisfy $X_i \leftrightarrow X_j$
      then delete *x* from DOMAIN[X_i]; *removed* $\leftarrow$ *true*
  return *removed*



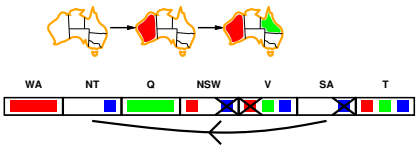
- Esim. kuvassa kaari SA → NSW on on konsistentti, koska SA:n arvolle B löytyy sopiva NSW:n arvo R.



- Esim. kuvassa kaari SA → NSW on on konsistentti, koska SA:n arvolle B löytyy sopiva NSW:n arvo R.
- Sen sijaan kaari NSW → SA ei ole joten poistetaan arvo B.



- Esim. kuvassa kaari SA → NSW on on konsistentti, koska SA:n arvolle B löytyy sopiva NSW:n arvo R.
- Sen sijaan kaari NSW → SA ei ole joten poistetaan arvo B.
- Jos muuttujan arvo poistetaan, on tarkistettava naapurit.



- Esim. kuvassa kaari SA → NSW on on konsistentti, koska SA:n arvolle B löytyy sopiva NSW:n arvo R.
- Sen sijaan kaari NSW → SA ei ole joten poistetaan arvo B.
- Jos muuttujan arvo poistetaan, on tarkistettava naapurit.
- Konsistenssin tarkastus solmujen NT ja SA välillä poistaa toisesta arvon B, jolloin solmulle ei jää mitään arvoja jäljelle eikä etsintää enää kannata jatkaa vaan osataan peruuttaa ajoissa. ⇒ Epäkonsistentit tilanteet löydetään nopeammin kuin eteenpäintarkistuksella.

- Kaarikonsistenssi täytyy tarkistaa toistuvasti, koska arvojen poistaminen voi synnyttää uusia ristiriitamahdollisuuksia.
- Algoritmin AC-3 aikavaatimus on luokkaa $O(n^2d^3)$ ja vaikka sen käyttäminen on paljon kalliimpaa kuin pelkkä eteenpäintarkistus, se yleensä kannattaa.
- Konsistenssista on myös vahvempia muotoja:
  - k-konsistenssi
  - vahva k-konsistenssi.
- Vahvempien konsistenssitarkistusten tekeminen vie aikaa, mutta pienentää haarautumiskerrointa.
- On periaatteessa mahdollista laskea etukäteen sellainen konsistenssitaso että konsistenssitarkistuksia tekemällä varsinainen haku ei joudu peruuttamaan, mutta sen laskeminen on usein epäkäytännöllistä. Sopivan tason löytäminen vaatii yleensä empiiristä kokeilua.

ERITYISTEN RAJOITTEIDEN KÄSITTELY

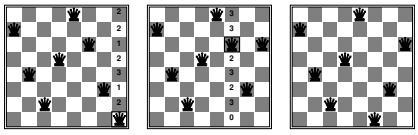
- Jotkin usein toistuvat erityisrajoitteet voidaan käsitellä nopeammin erityisesti niitä varten suunnitelluilla algoritmeilla.
- Esimerkiksi AllDiff: kaikilla muuttujilla on oltava eri arvot. Jos erilaisia arvoja on vähemmän kuin muuttujia, tämä ei voi toteutua. Siis saadaan algoritmi:
  1. Poista muuttuja jolla on vain yksi arvo ja poista sen arvo muiden muuttujien mahdollisten arvojen joukosta.
  2. Toista niin kauan kun on muuttujia joilla on vain yksi mahdollinen arvo. Jos milloin tahansa käy niin että jollain muuttujalla ei ole arvoja jäljellä tai arvoja on vähemmän kuin muuttujia, tilanne on inkonsistentti.
- Kirjassa muitakin esimerkkejä: *resurssirajoitteet* (resource constraint), *rajojen säätäminen* (bounds propagation)

ÄLYKÄS PERUUTTAMINEN

- Kronologinen peruuttaminen: viimeksi asetettu arvo peruutetaan
- Järkevämpää on palata johonkin niistä muuttujista, jotka aiheuttivat konfliktin. Jos muuttujalle X ei enää löydy konsistenttia arvoa, niin X:n *konfliktijoukkoon* (conflict set) kuuluvat kaikki aiemmin sijoitetut muuttujat joilla on rajoitteita X:n kanssa.
- *Paluuhyppymenetelmä* (backjumping) palaa viimeiseksi asetettuun konfliktijoukon muuttujaan.
- Paluuhyppymenetelmä ei löydä muita kuin sellaisia konflikteja jotka löytyvät myös eteenpäintarkistuksella (forward checking), joten niiden käyttäminen samanaikaisesti ei kannata.
- *Konfliktiohjattu paluuhyppymenetelmä* (conflict-directed backjumping) pyrkii tarkemman konfliktijoukon määrittelyn avulla löytämään ongelmakohdat nopeammin.

PAIKALLINEN ETSINTÄ RAJOITEONGELMISSA

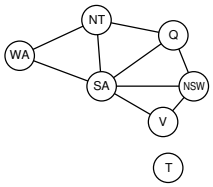
- Konfliktien minimointi -heuristiikka (min-conflicts heuristic): valitaan muuttujalle konflikteja minimoiva arvo



- (Kuvassa valinta olisi hyvin voinut mennä toisinkin.)
- Hämmästyttävän nopea suurillakin ongelmilla: Jopa miljoonan kuningattaren ongelma tulee ratkaistuksi keskimäärin 50 askeleessa.
- Paikallisen etsinnän etuna on myös sen käytön mahdollisuus online-tilanteissa: rajoitteet voivat muuttua mutta hakua voidaan jatkaa palaamatta alkuun.

ONGELMIEN RAKENNE

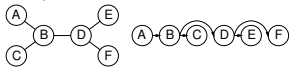
- Joskus (harvoin) on mahdollista jakaa koko ongelma toisistaan riippumattomiin osaongelmiin.
- Rajoiteverkkoa käyttämällä sellaiset löytyvät helposti *yhtenäisistä komponenteista* (connected components).
- Osaongelmien sijoitukset voidaan suoraan yhdistää koko ongelman sijoitukseksi.



- Tavallisesti osaongelmat eivät ole toisistaan riippumattomia.
- Mutta jos ovat, voidaan saada huomattava säästö:
  - Oletetaan, että koko ongelmassa on n totuusarvoista muuttujaa, mutta se voidaan jakaa c:n muuttujan osaongelmiksi, joita siis tulee n/c kappaletta.
  - Nyt kunkin osaongelman vaativuus on $O(d^c)$ joten koko ongelman vaativuus on enää $O(n/c \cdot d^c)$, joka on n:n suhteen lineaarinen, toisin kuin aiempi $O(d^n)$.
  - Esimerkiksi jos $n = 80, d = 2, c = 20$ ja solmuja voidaan käsitellä 10 miljoonaa sekunnissa, niin
    - 2^{80} solmua voidaan käsitellä neljässä miljardissa vuodessa ,
    - $4 \cdot 2^{20}$ solmua voidaan käsitellä 0,4 sekunnissa.

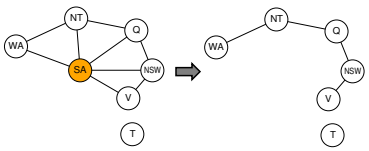
PUUN MUOTOISET RAJOITEVERKOT

- Jos rajoiteverkko on puu, se voidaan ratkaista lineaarisessa ajassa $O(nd^2)$ muuttujien lukumäärän n suhteen. Algoritmi:
  1. Valitse jokin muuttuja puun juureksi, järjestä kaikki lapset järjestykseen vanhempiensa jälkeen ja nimeä järjestyksen perusteella $X_1, \ldots, X_n$:ksi:



  2. Kaikille j n:stä 2:een, sovella kaarikonsistenssia kaarelle (X_i, X_j), missä X_i on X_j:n vanhempi ja poista tarvittaessa arvoja X_i:n (vanhemman) arvoalueesta
  3. Kaikille j 1:stä n:ään, sijoita X_j:lle jokin arvo, joka on konsistentti sen vanhemmalle asetetun arvon kanssa.
- Koska 2:n jälkeen verkko on kaarikonsistentti, vaihe 3 ei vaadi peruutusta. Käänteinen järjestys kohdassa 2 varmistaa, ettei arvojen poistaminen uhkaa jo asetettujen kaarten konsistenssia.

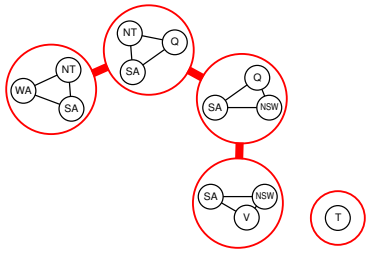
- Nyt kun puiden käsittelyyn on olemassa tehokas algoritmi, voidaan yleisempiä rajoiteverkkoja yrittää redusoida puiksi. Tähän on kaksi menetelmää: solmujen poistaminen ja solmujen yhdistäminen.
- Ensimmäisestä on esimerkki kuvassa: Syklin aiheuttava solmu poistetaan käsittelystä kiinnittämällä sen arvo ja poistamalla muista valitun kanssa yhteensopimattomat arvot, jolloin jäljelle jäävä verkko on puumuotoinen. Jos ratkaisua ei löydy, täytyy kokeilla uudestaan jollain toisella alkuarvolla.



- Saadaan algoritmi:
  1. Valitse muuttujien joukosta osajoukko S, jonka poistaminen muuttaa rajoiteverkon puuksi. S on "syklienleikkausjoukko" (cycle cutset)
  2. Kaikille mahdollisille S:n muuttujien sijoituksille
     (a) poista muiden muuttujien arvojoukoista S:n arvojen kanssa yhteensopimattomat
     (b) jos jäljelle jääneellä puumuotoisella rajoiteongelmalla on ratkaisu, palauta se yhdessä S:n sijoituksen kanssa.
- Algoritmin aikavaatimus on luokkaa $O(d^c \cdot (n-c)d^2)$ kun c on joukon S koko.
- Jos c on pieni, niin säästöä syntyy, mutta c voi olla jopa $(n-2)$, jolloin ollaan taas eksponentiaalisessa aikavaativuudessa. Pienimmän syklejä aiheuttavan joukon löytäminen on NP-kova ongelma.

- Toinen tapa on "puuhajotusmenetelmä" (tree decomposition), jossa rajoiteverkko jaetaan joukoksi yhdistettyjä osaongelmia.
- Seuraavia periaatteita täytyy noudattaa:
  - Jokaisen muuttujan täytyy olla ainakin yhdessä osaongelmassa.
  - Jos kahdella muuttujalla on jokin yhteinen rajoite, niiden täytyy esiintyä yhdessä (tuon rajoitteen kanssa) ainakin yhdessä osaongelmassa.
  - Jos muuttuja esiintyy kahdessa puun osaongelmassa, sen täytyy esiintyä jokaisessa noiden kahden ongelman välisellä polulla esiintyvässä osaongelmassa

- Sitten ratkaistaan erikseen kukin osaongelma (jos ratkaisua johonkin ei löydy niin tiedetään ettei sitä löydy koko ongelmallekaan) minkä jälkeen kaikkien osaongelmien kaikki ratkaisut ovat uusien "megamuuttujien" mahdollisia arvoja ja voidaan jälleen käyttää puumuotoisten rajoiteongelmien ratkaisualgoritmia.



- Yhdestä rajoiteongelmasta voidaan muodostaa erilaisia jakoja osaongelmiksi. Tällöin tavoitteena on tehdä osaongelmista mahdollisimman pieniä, jotta ne voitaisiin ratkaista nopeasti (ne eivät yleensä ole puumuotoisia).
- Jaon *puunleveydeksi* (tree width) kutsutaan lukua joka on yhtä pienempi kuin suurimman osaongelman koko.
- Jos puunleveys on w ja jako osaongelmiksi on annettu, voidaan koko ongelma ratkaista ajassa $O(nd^{(w+1)})$, joten rajoiteongelmat, joiden puunleveys on rajoitettu, ovat ratkaistavissa polynomisessa ajassa.
- Sellaisen jaon löytäminen, jolla on mahdollisimman pieni puunleveys, on NP-kovaa, mutta käytännöllisiä heuristisia menetelmiä on olemassa.