A mesterséges intelligencia alapjai

kényszerkielégítési problémák

Áttekintés

- példák kényszerkielégítési problémákra
- problémamegoldás visszalépéses kereséssel
- heurisztikák
- problémák struktúrája és felbontása
- lokális keresés

Kényszerkielégítési probléma

- általános keresési probléma
  - az állapot egy *fekete doboz*,
  - az állapotot bármilyen adatstruktúra ábrázolhatja
  - csak az állapotátmenetek, heurisztika és célállapot legyen implementálva
- kényszerkielégítési probléma
  - az állapotot D_i tartományból származó X_i változókkal definiáljuk
  - a célteszt kényszerek halmaza, mely mindegyike a változók egy részhalmazát és megfelelő értékeket tartalmazzák

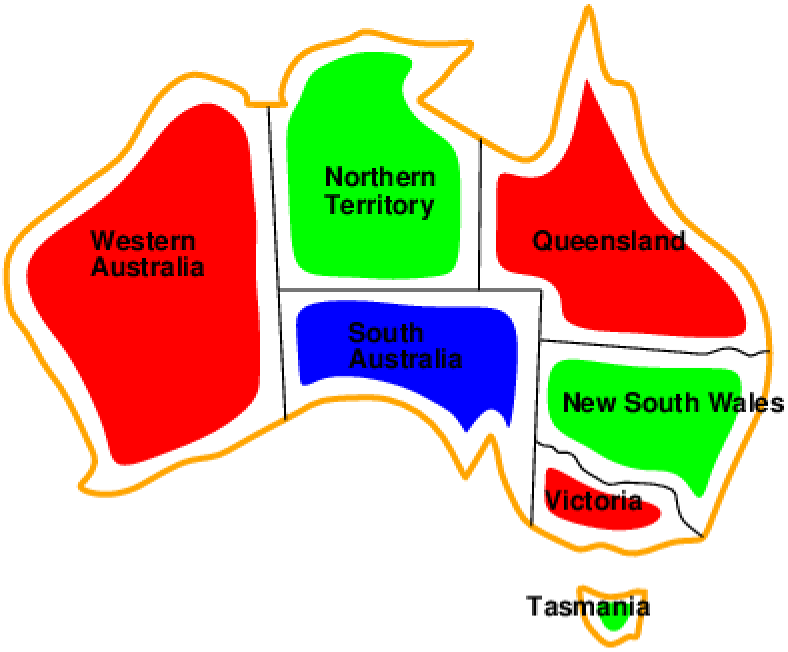
Példa: térképszínezés

- *változók*: WA, NT, Q, NSW, V, SA, T
- *tartományok*: D_i = {piros, zöld, kék}
- *kényszerek*: szomszédos tartomány nem lehet ugyanolyan színű
  - WA ≠ NT

Példa: térképszínezés

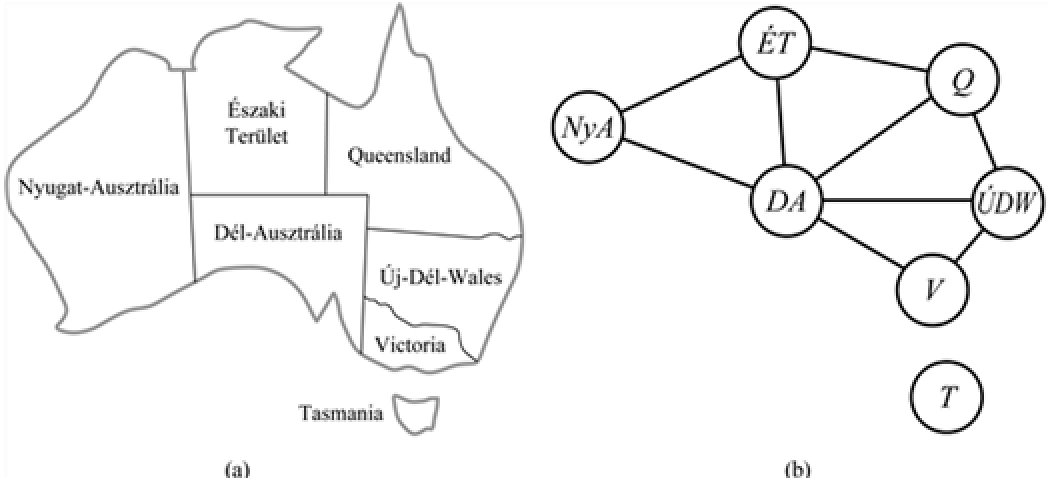
A megoldások teljesítik az összes kényszert, pl.

- WA = piros
- NT = zöld
- Q = piros
- NSW = zöld
- V = piros
- SA = zöld
- T = zöld



Kényszergráf

- **bináris kényszerkielégítési feladat**: minden kényszer maximum két változót tartalmaz
- **kényszergráf**: a csúcsok a változók, az élek a kényszereket jelölik
- a gráf szerkezetét felhasználva a keresés felgyorsítható, Tazmánia független részprobléma



Kényszerkielégítési feladatok fajtái

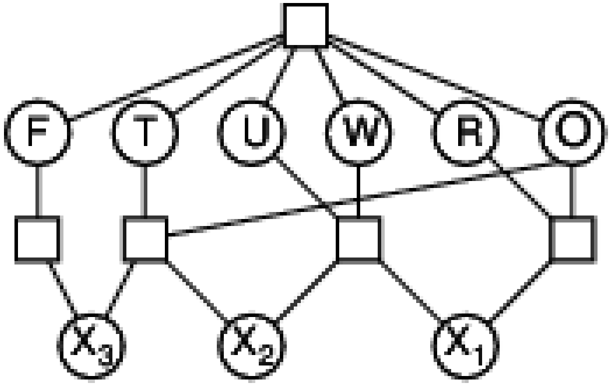
- Diszkrét változók
  - véges tartományok
    - ha ez d elemű, akkor $O(d^n)$ teljes értékadás
    - logikai kényszerkielégítési probléma (speciálisan SAT) ⇒ NP-teljes a megoldás
  - végtelen tartományok (egészek, sztringek)
    - feladatok ütemezése (változók jelölik a kezdő/záró napokat)
    - speciális kényszernyelv használatos ($Start1 + 5 \leq Start3$)
    - lineáris kényszereknél megoldható
    - nemlineáris esetben eldönthetetlen
- Valós változók
  - Hubble űrteleszkóp megfigyelések kezdete/vége
  - lineáris kényszerek esetén polinomiális időben megoldható (lineáris programozás)

Kényszerek fajtái

- kemény kényszerek
  - unáris kényszer egy változót tartalmaz
    - SA ≠ zöld
  - bináris kényszer két változót tartalmaz
    - SA ≠ WA
  - magasabb rendű kényszer legalább három változót tartalmaz
    - pl. betűrejtvény: $S+M+X_3=O+10X_4$
- preferenciák – puha kényszerek: a piros jobb mint a zöld
  - rendszerint minden értékeléshez költség kapcsolódik → kényszert tartalmazó optimalizációs probléma

Példa: betűrejtvény

```
  T W O
+ T W O
-------
F O U R
```



- Változók: F, T, U, W, R, O, X_1, X_2, X_3
- Tartomány: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
- Kényszerek:
  - mind-különböző(F, T, U, W, R, O)
  - $O + O = R + 10\ X_1$
  - …

Valós életbeli kényszerkielégítési problémák

- hozzárendelési probléma
  - Melyik osztályt ki tanítsa?
- órarend/menetrend problémák
  - melyik óra mikor és hol legyen
- hardver konfiguráció
- táblázatkezelő
- szállítmányozás tervezés
- gyártásütemezés
- alaprajz tervezés

A valós világ problémái rendszerint valós értékű változókkal írhatóak le

Standard (inkrementális) keresési módszer

Az állapotokat a már értékkel rendelkező változók adják meg

- *kezdeti állapot*: egyik változónak sincs értéke
- *rákövetkező függvény*: rendeljünk egy olyan értéket valamely még értékkel nem rendelkező változóhoz, mely nem okoz konfliktust
    - ha nincs ilyen érték, akkor sikertelen a keresés
- *célteszt*: az értékelés teljes, minden változó kapott értéket

1. egységes minden kényszerkielégítési feladatra
2. n változó esetén minden megoldás n mélységben van ⇒ mélységi keresés
3. az út érdektelen
4. b = (n-L)d az L. szinten, így $n!d^n$ levele van a keresőfának

Visszalépéses keresés

- az értékek hozzárendelése a változókhoz kommutatív:
  - [WA=piros, NT=zöld] ugyanaz, mint [NT=zöld, WA=piros]
- minden szinten csak egy változónak kell értéket keresni
  - b = d, így a keresőfa d^n levelet tartalmaz
- a kényszerkielégítési feladatra alkalmazott mélységi keresést, ahol egyszerre egy változó kap értéket **visszalépéses** keresésnek nevezzük
- a visszalépéses keresés az alapvető nem informált módszere a kényszerkielégítési feladatoknak
- az n vezér problémáját n=25 esetre képes megoldani

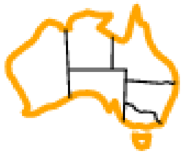
Visszalépéses keresés – pszeudokód

function Backtracking-Search(csp): megoldás vagy „sikertelen”
  return Recursive-Backtracking({ }, csp)
function Recursive-Backtracking(assignment,csp): megoldás vagy „sikertelen”
  if assignment <u>teljes</u> then return assignment
  var := Select-Unassigned-Variable(Variables[csp], assignment, csp)
  for each value in Order-Domain-Values(var, assignment, csp) do
    if value <u>konzisztens</u> a Constraints[csp] értékadással then
      assignment += {var = value}
      result := Recursive-Backtracking(assignment, csp)
      if result != <u>sikertelen</u> then return result
      assignment -= {var = value}
  return <u>sikertelen</u>

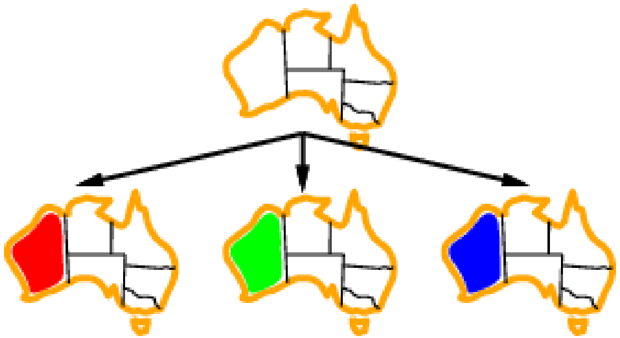
AIMA kód

```
def backtrack(assignment):
    if len(assignment) == len(csp.variables):
        return assignment
    var = select_unassigned_variable(assignment, csp)
    for value in order_domain_values(var, assignment, csp):
        if 0 == csp.nconflicts(var, value, assignment):
            csp.assign(var, value, assignment)
            removals = csp.suppose(var, value)
            if inference(csp, var, value, assignment, removals):
                result = backtrack(assignment)
                if result is not None:
                    return result
            csp.restore(removals)
    csp.unassign(var, assignment)
    return None
```

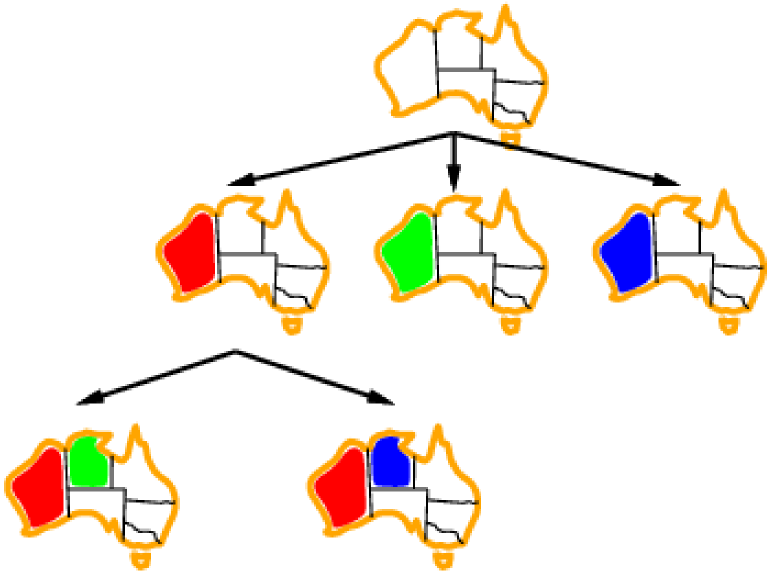
Példa visszalépéses keresésre



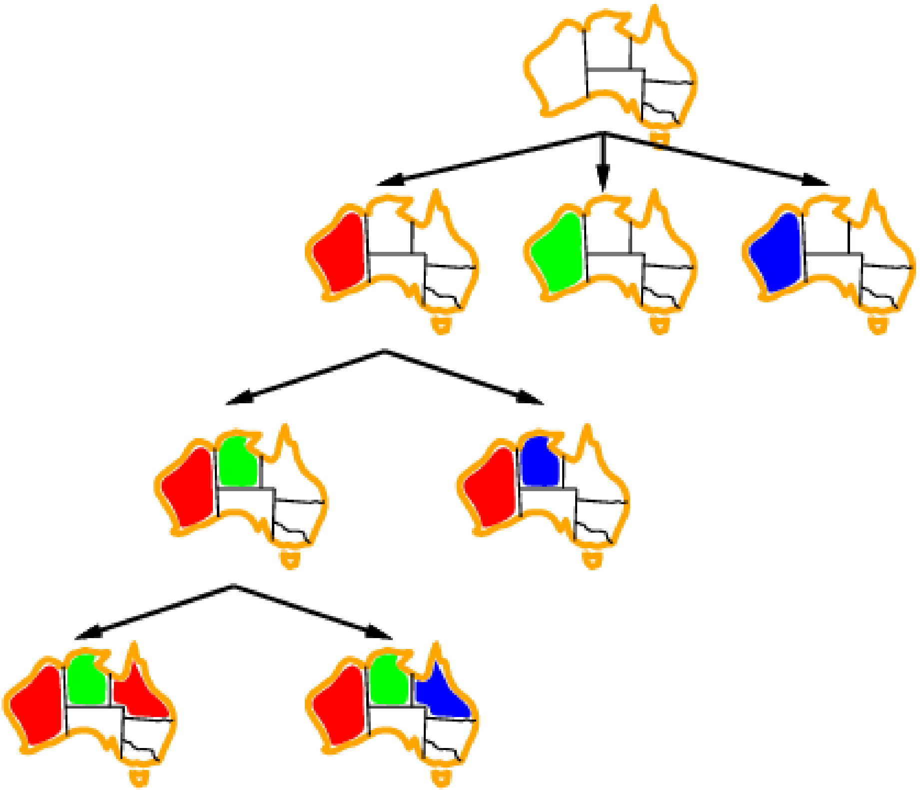
Példa visszalépéses keresésre



Példa visszalépéses keresésre



Példa visszalépéses keresésre



Visszalépéses keresés hatékonyságának növelése

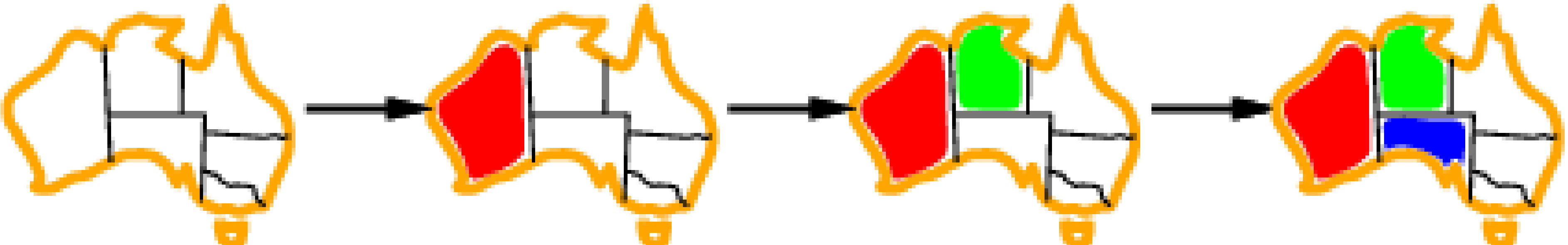
Az általános módszert jelentősen felgyorsíthatjuk:

- Mely legyen következő változó, mely értéket kap?
- Az értékeket milyen sorrendben próbáljuk ki?
- Korán fel tudjuk ismerni az elkerülhetetlen hibákat?
- Felhasználhatjuk a feladat szerkezetét?

Legkevesebb fennmaradó érték (MRV)

Válasszuk azt a változót, melynek a legkevesebb „megengedett” értékkel rendelkezik. (Legkorlátozottabb változó)

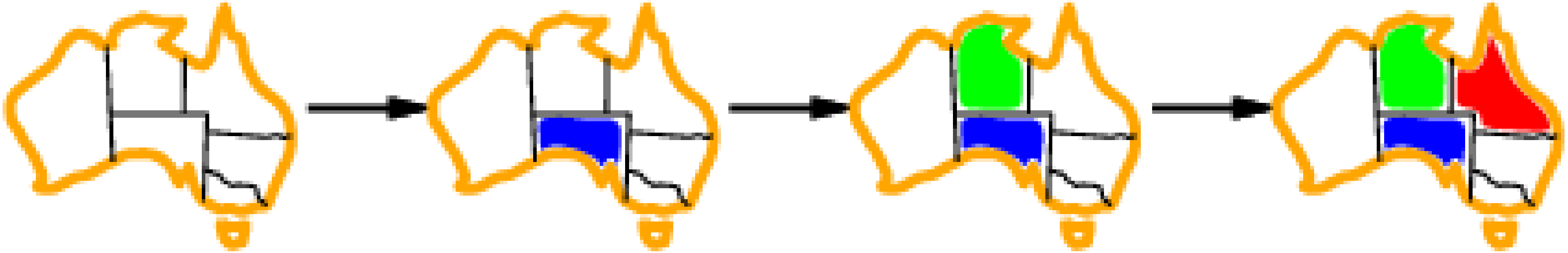
- gyorsan rátalál a hibákra, megnyesi a keresési fát



Fokszám heurisztika

holtverseny esetén dönt a legkevesebb fennmaradó érték heurisztika változóiról

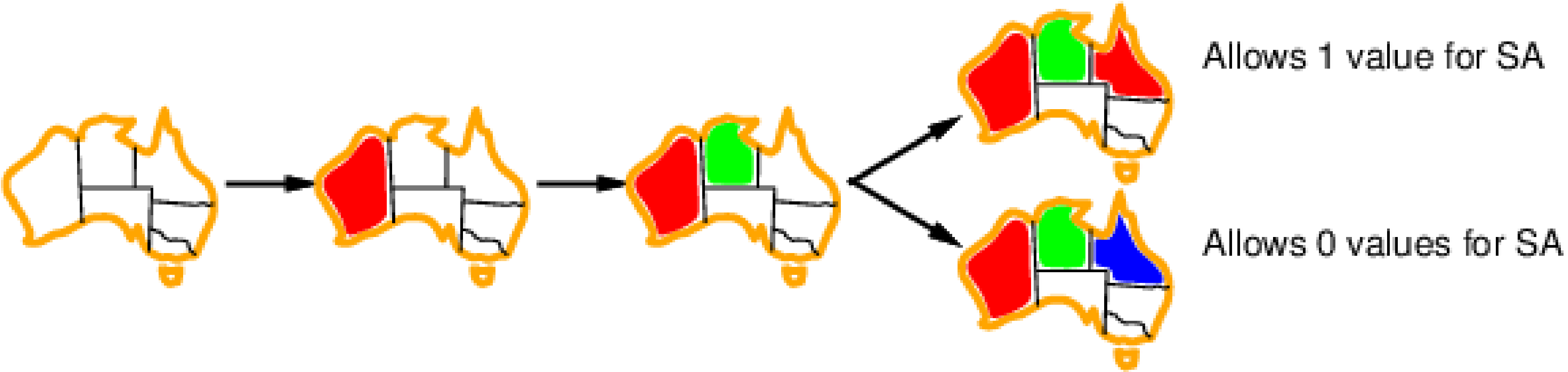
- válasszuk azt a változót, amely legtöbbször szerepel a hozzárendeletlen változókra vonatkozó kényszerekben



Legkevésbé korlátozó érték

Milyen sorrendben vizsgáljuk meg az értékeket?

- válasszuk azt az értéket, mely a legkevesebb értéket zárja ki a hozzárendeletlen változóknál

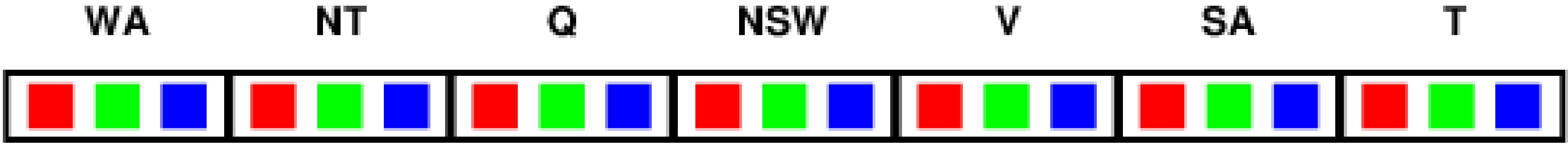


ezeket a heurisztikákat használva 1000 vezér problémáját képesek vagyunk megoldani

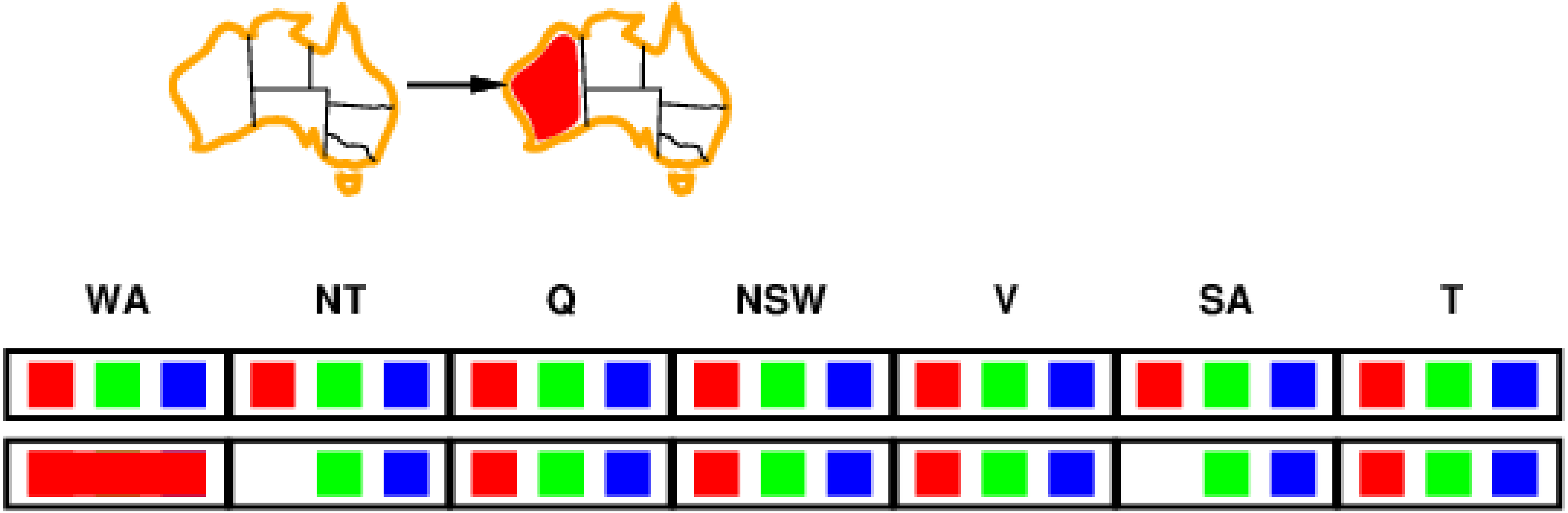
Előrenéző ellenőrzés

Ötlet: tartsuk nyilván a hozzárendeletlen változók lehetséges értékeit

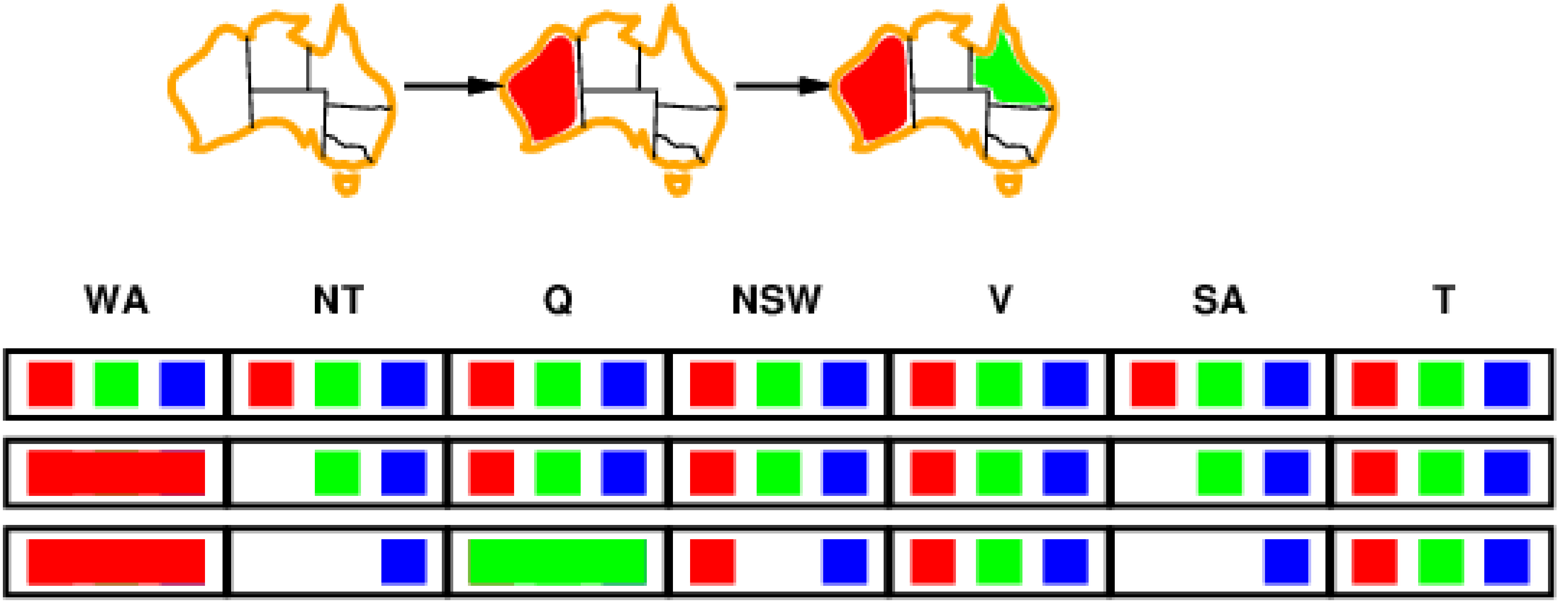
- állítsuk le a keresést, ha valamely változónak már nem maradt értéke



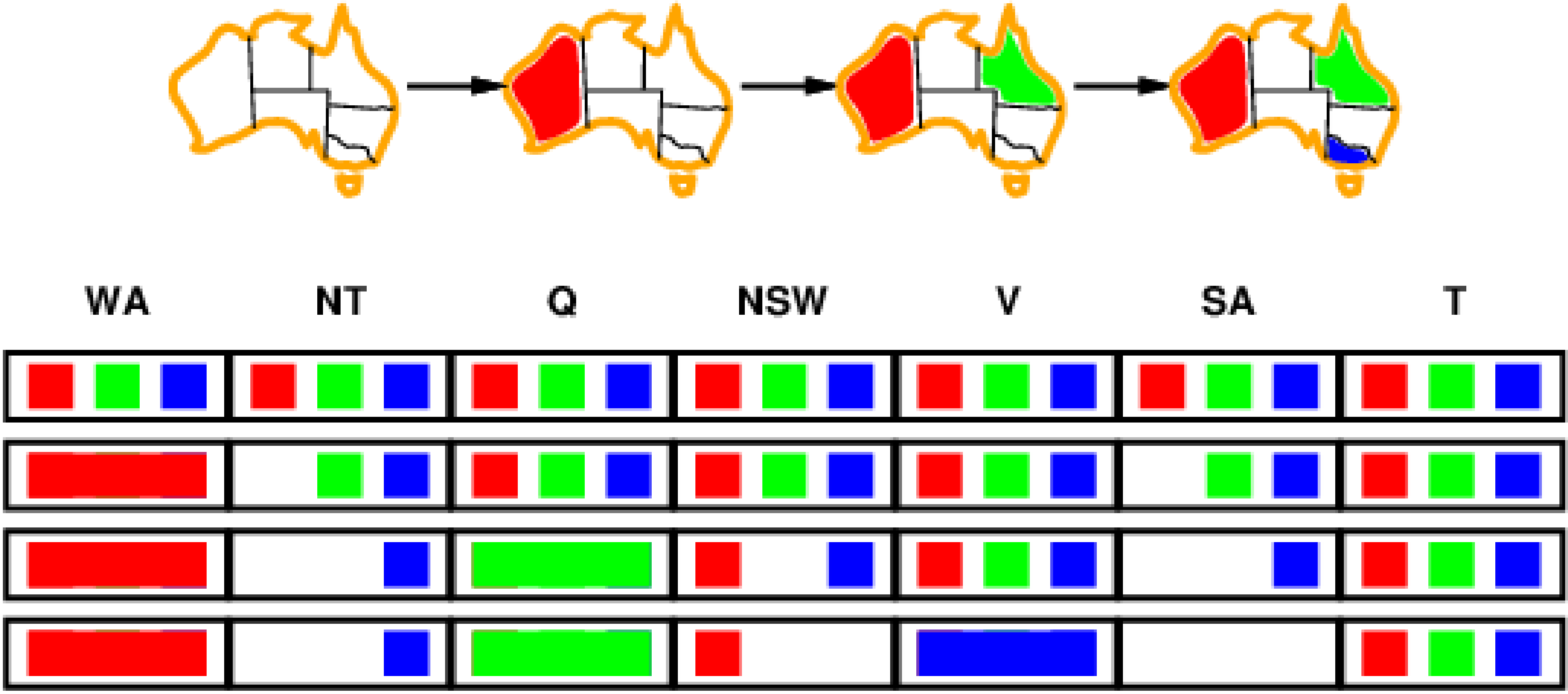
Előrenéző ellenőrzés



Előrenéző ellenőrzés



Előrenéző ellenőrzés

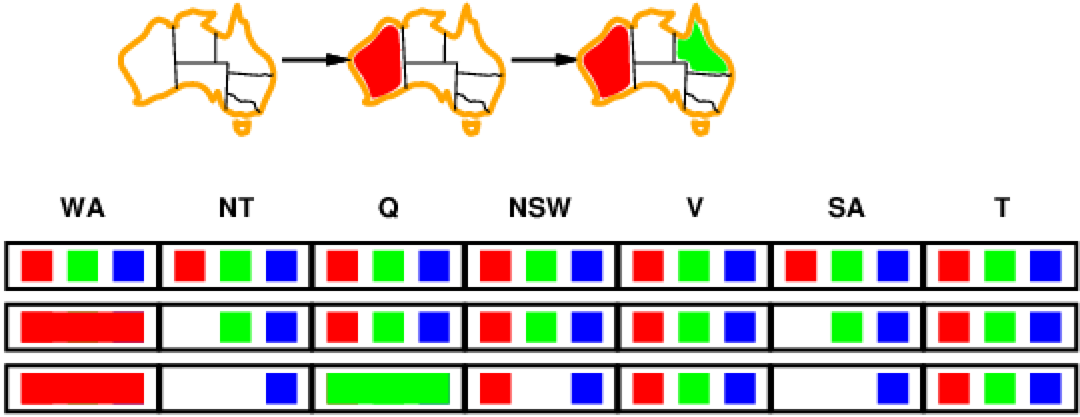


Kényszerek terjesztése

Az előrenéző ellenőrzés sok inkonzisztenciát észrevesz, de nem mindent. A harmadik lépésben már kikövetkeztethető a zsákutca, de az előrenéző ellenőrzés csak egy lépést vizsgál előre.

Mind NT, mind SA csak kék lehet.

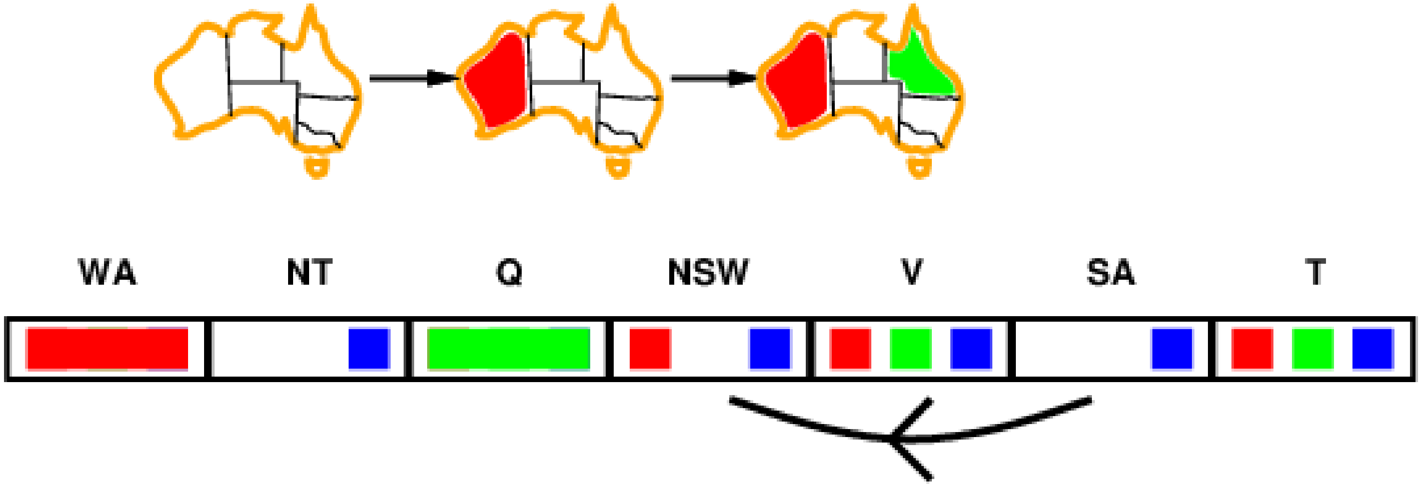
A kényszerek terjesztése lokálisan érvényesíti a kényszereket



Élkonzisztencia

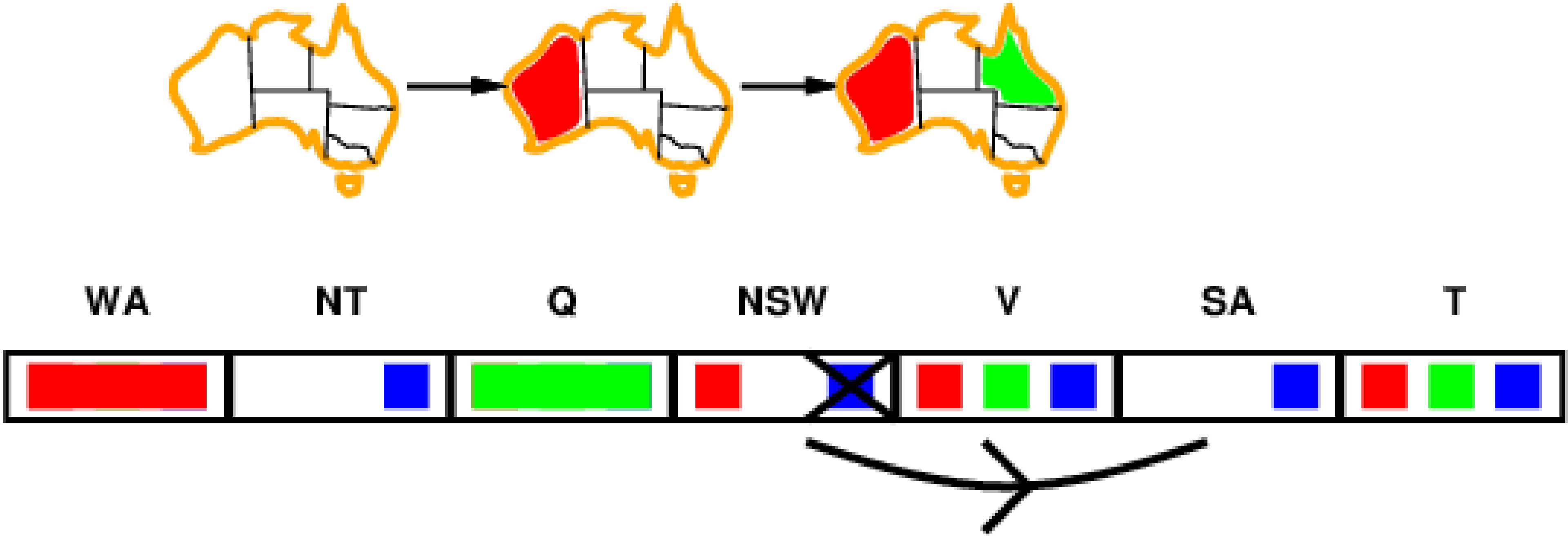
A kényszerek terjesztésének legegyszerűbb formája az, ha minden élt konzisztenssé teszünk

az X → Y él konzisztens, ha X minden x értékéhez van y ϵ Y megengedett érték

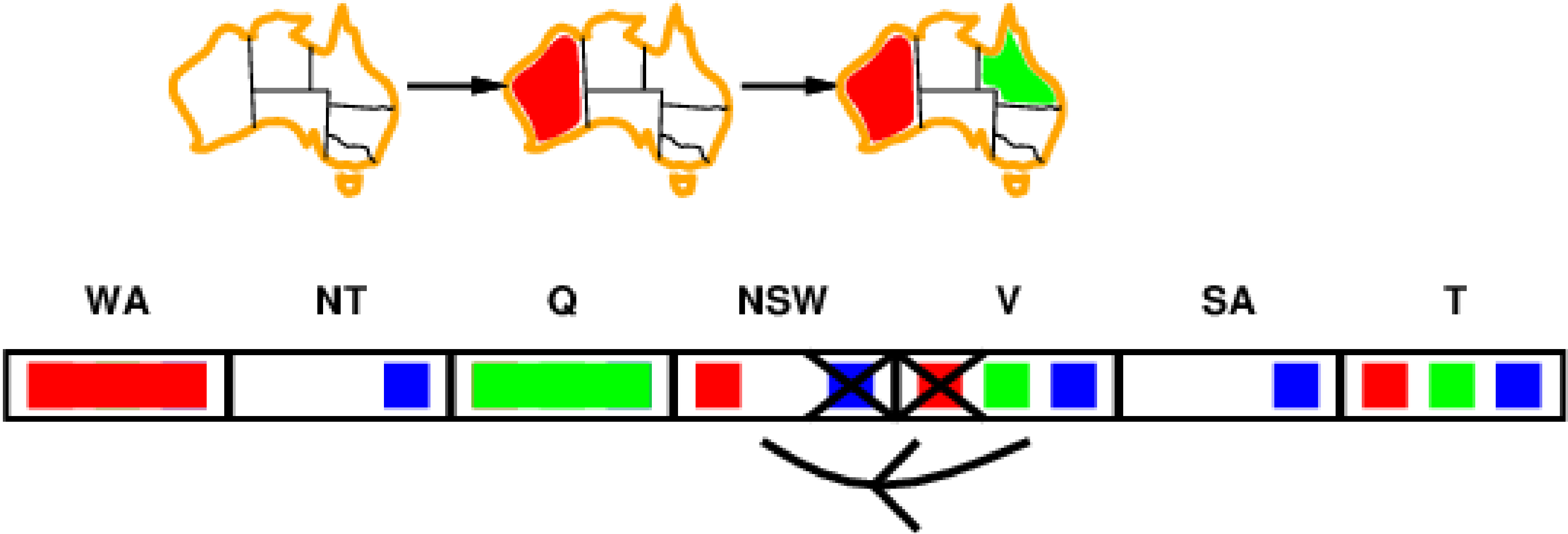


Élkonzisztencia

ha az X változó egy értékét töröljük, akkor X szomszédait újra kell vizsgálni

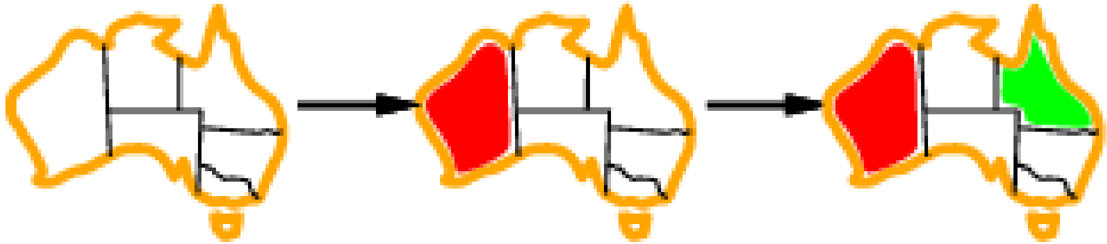


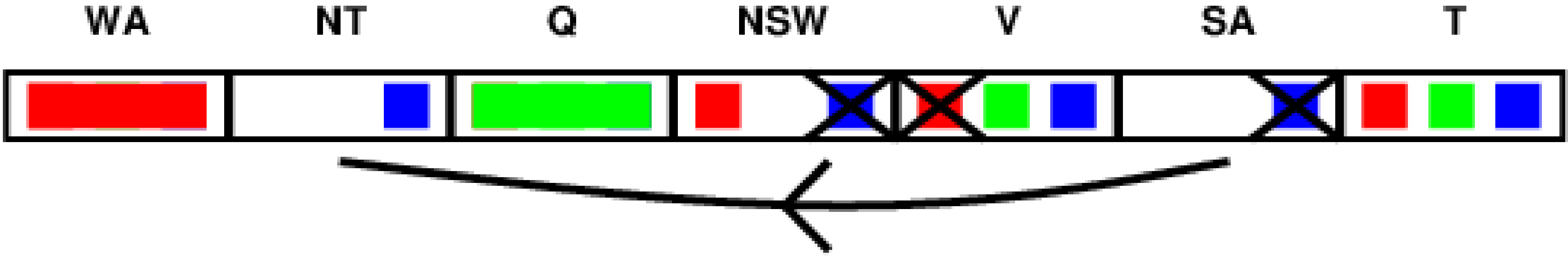
Élkonzisztencia



Élkonzisztencia

- az élkonzisztencia gyorsabban felfedezi a hibákat, mint az előrenéző ellenőrzés
- minden értékadás után érdemes lefuttatni





Élkonzisztencia

function AC-3(csp): kényszer kielégítési probléma, esetleg redukált tartományokkal

// csp változói: X_1, … X_n

queue: élek sora, kezdetben az összes élt tartalmazza

while queue is not empty do

(X_i, X_j) := Remove-First(queue)

if Remove-Inconsistent-Values(X_i, X_j) then

for each X_k in Neighbors[X_i] do

queue += (X_k, X_i)

Élkonzisztencia

function Remove-Inconsistent-Values(X_i, X_j)*: igaz (ha sikeres)/hamis*

removed := false

for each x in Domain[X_i] *do*

if nincs y ϵ *Domain*[X_j] *melyre (x,y) teljesíti az* $X_i \leftrightarrow X_j$ *kényszert*

then Domain[X_i] *-= x*

removed := true

return removed

- $O(n^2d^3)$ bonyolultság (csökkenthető $O(n^2d^2)$-re AC4-gyel)
- detektálás NP-nehéz (spec. tartalmazza a 3SAT feladatot)

Probléma szerkezete

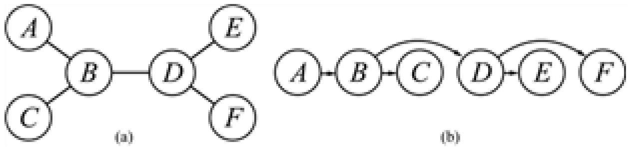
- Tazmánia és a szárazföld színezése független részprobléma
  - kényszergráf összefüggő komponensei
- tegyük fel, hogy minden részproblémának c változója van (az n-ből)
- legrosszabb esetben $n/c \cdot d^c$, ami lineáris n-ben
- ha n=80, d=2, c=20
  - 2^{80} = 4 milliárd év (10 millió csúcs/s)
  - $4 \cdot 2^{20}$ = 0,4 s (10 millió csúcs/s)

Faszerkezetű kényszerkielégítési probléma

Tétel: *ha a kényszergráf nem tartalmaz hurkot, a probléma $O(nd^2)$ időben megoldható.*

- általános esetben ez $O(d^n)$
- hasonlóan igaz logikai és valószínűségi következtetésekre is.

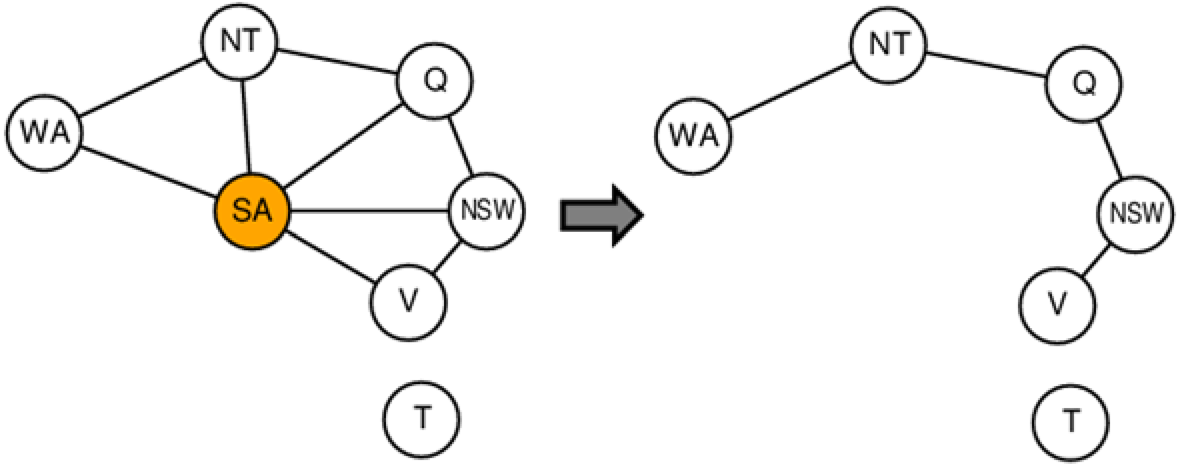
Algoritmus



1. készítsük el a gráf topologikus elrendezését
2. visszafele haladva alkalmazzuk a *RemoveInconsistent(Parent(X_j),X_j)*-t
3. előre haladva a X_j-nek adjunk értéket a szülője értékével összhangban

Közel fastruktúrájú kényszerkielégítési feladatok

- Készítsünk belőle fastruktúrájú problémát egyes csúcsok értékelésével (és a szomszédjaik tartományának szűkítésével)!
- Ha az értékelt csúcsok száma c, a teljes futási idő $O(d^c (n-c)d^2)$, ami kis c esetén nagyon gyors



Lokális keresések kényszerkielégítési problémákra

A hegymászás, szimulált hűtés, *teljes* állapottal dolgozik, amikor minden változó értékelt.

- alkalmazási lehetőségek
  - hozzárendeletlen változókat tartalmazó kényszerek engedélyezése
  - változók értékét átíró operátorok
- változó kiválasztása
  - konfliktusos változók közül véletlenszerűen
- értékek választása (minimális konfliktusok)
  - a legkevesebb kényszert megszegő érték
  - hegymászó keresés, ahol a heurisztika a megszegett kényszerek száma

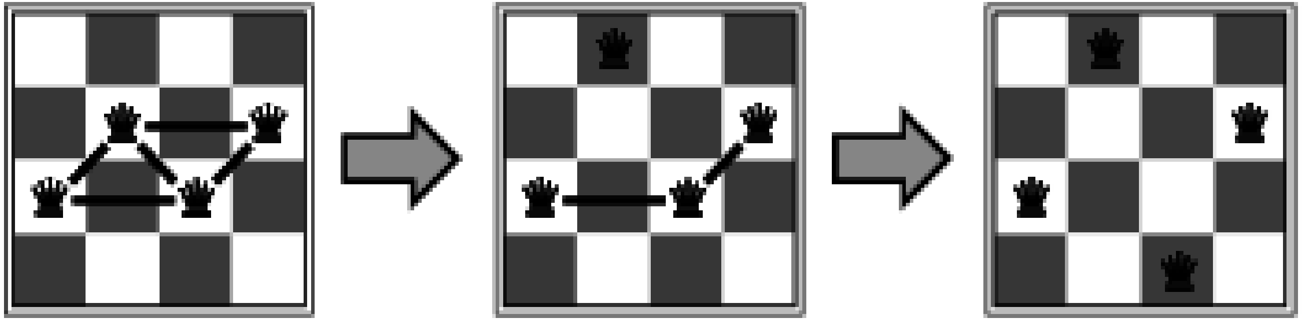
Példa: 4 vezér

Állapotok: 4 vezér 4 oszlopban (256 állapot)

Operátor: vezér mozgatása az oszlopában

Célteszt: nincsenek ütésben

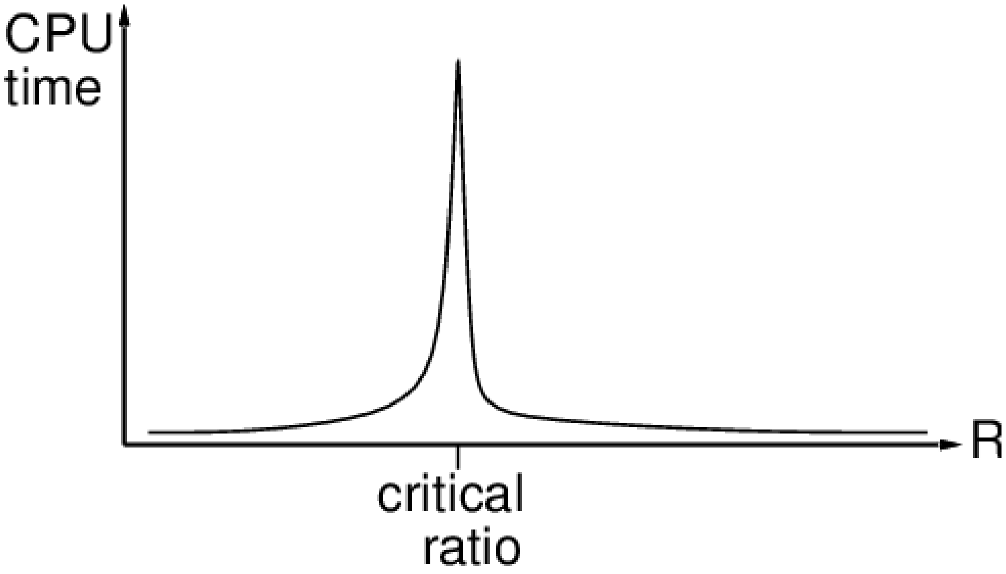
Értékelőfv: h(n) = ütések száma



h = 5 h = 2 h = 0

Minimális konfliktusok teljesítménye

- Tetszőleges kiinduló állapotból közel konstans idő alatt megold 10^7 méretű n-vezér problémát.
- Hasonló igaz véletlen módon generált kényszerkielégítési problémákra is, kivéve egy szűk régiót.
- R = # kényszerek/ # változók



Összefoglalás

- a kényszerkielégítési problémák speciális szerkezetűek
  - adott változók halmazával definiáljuk az állapotokat
  - a céltesztet a változók értékeit tartalmazó kényszerek együttese adja meg
- visszalépéses keresés = mélységi keresés egyszerre egy változó értékadásával
- a változók és az értékek megfelelő kiválasztása sokat gyorsít
- az előrenéző ellenőrzés segít kivédeni a későbbi kudarcot
- a kényszerek terjesztése (élkonzisztencia) tovább gyorsít a keresésen
- lehetséges a probléma szerkezetének vizsgálata, felhasználása
- faszerkezetű feladatok lineáris időben megoldhatóak
- a min-konfliktusok módszere hatékony a gyakorlatban