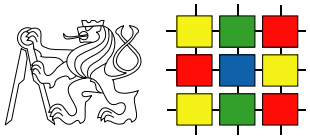
Alternativní algoritmy prohledávání stavového prostoru

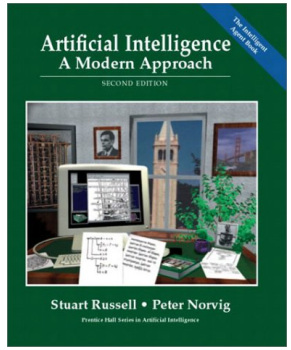
Michal Pěchouček

Department of Cybernetics
Czech Technical University in Prague



http://labe.felk.cvut.cz/~pechouc/kui/alter-noinfo.pdf

Použitá literatura pro umělou inteligenci



:: Artificial Intelligence: A Modern Approach (Second Edition) by Stuart Russell and Peter Norvig, 2002 Prentice Hall.

http://aima.cs.berkeley.edu/

Obsah

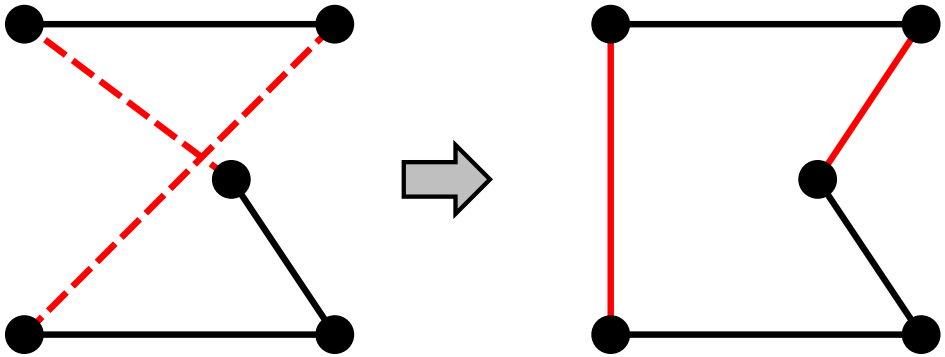
- Gradientní algoritmus (Hill-climbing)
- Simulované žíhání
- Genetické algoritmy
- Řešení problémů splňováním omezujících podmínek

Algoritmy založené na postupném zlepšování

- V řadě optimalizačních problémů není důležitá cesta, ale řešení
- V takovém případě je stavový prostor množinou **úplných** konfigurací
- Optimalizační úlohou je pak najít *optimální* konfiguraci, např.,
  - nejkratší cestu v obchodním cestujícím nebo,
  - nalezení konfigurace splňující všechna omezení, např. rozvrh
- V těchto případech lze použít algoritmy založené na **postupném zlepšování**; pracovat s jedním aktuálním stavem a snažit se ho postupně zlepšit
- Pracujeme s konstantním prostorem, vhodné pro *online* stejně tak jako *offline* prohledávání.

Příklad: Problém Obchodního Cestujícího

Začněme s libovolnou úplnou cestou, a provádějme párové výměny cest

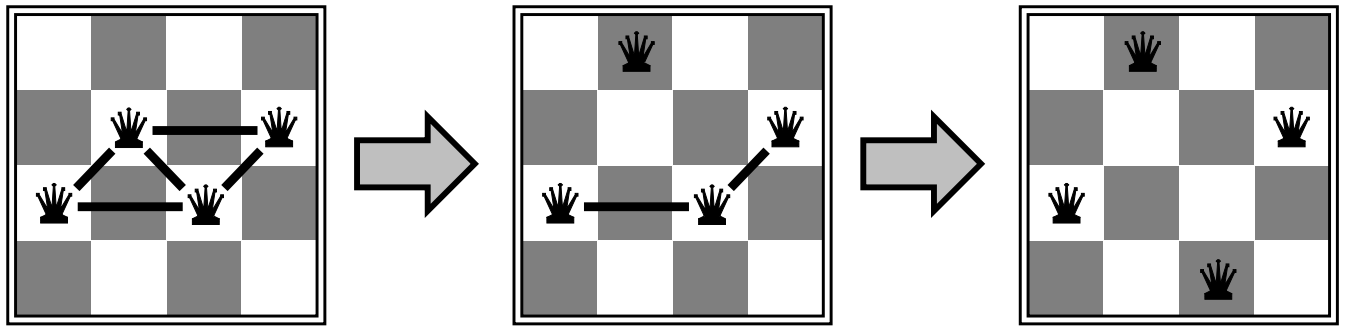


Varianty tohoto přístupu se dostanou velmi rychle do okolí 1% okolo optima i v úloze o tisících měst

Příklad: n-královen

Umístit n královen na šachovnici $n \times n$ tak aby se neohrožovaly

Začněte s libovolným uspořádáním a udělejte takový tah abyste redukovali počet konfliktů



Řeší problém n-královen *skoro* vždy téměř okamžitě
dokonce pro velmi velké n, t.j., $n = 10^6$

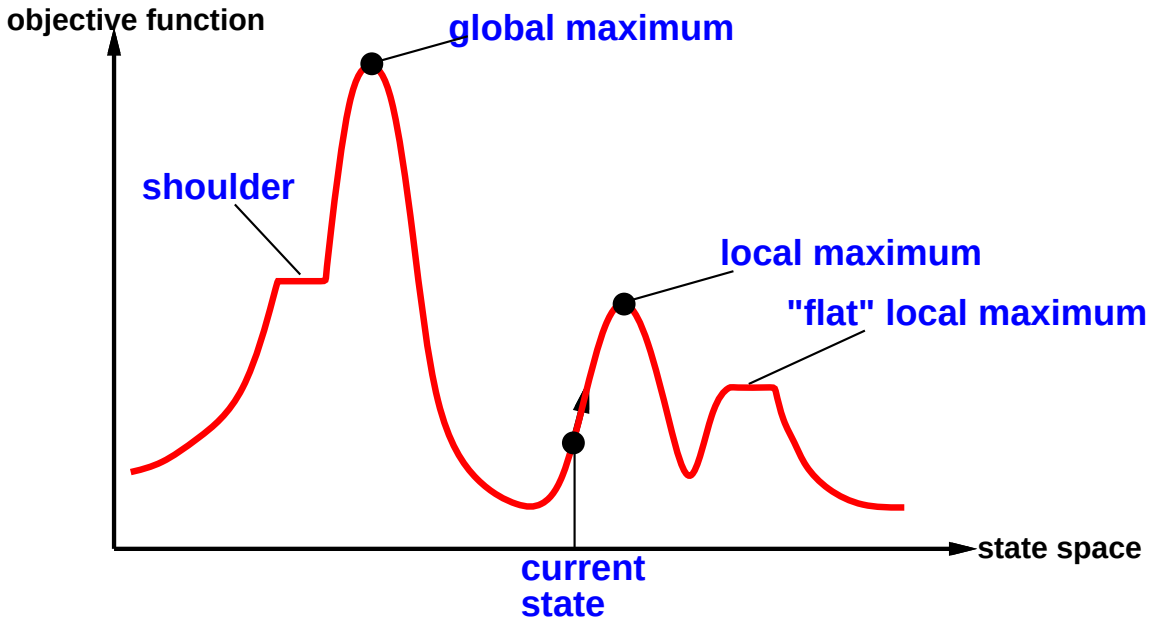
Gradietní Algoritmus (Hill-climbing)

"Like climbing Everest in thick fog"

```
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
```

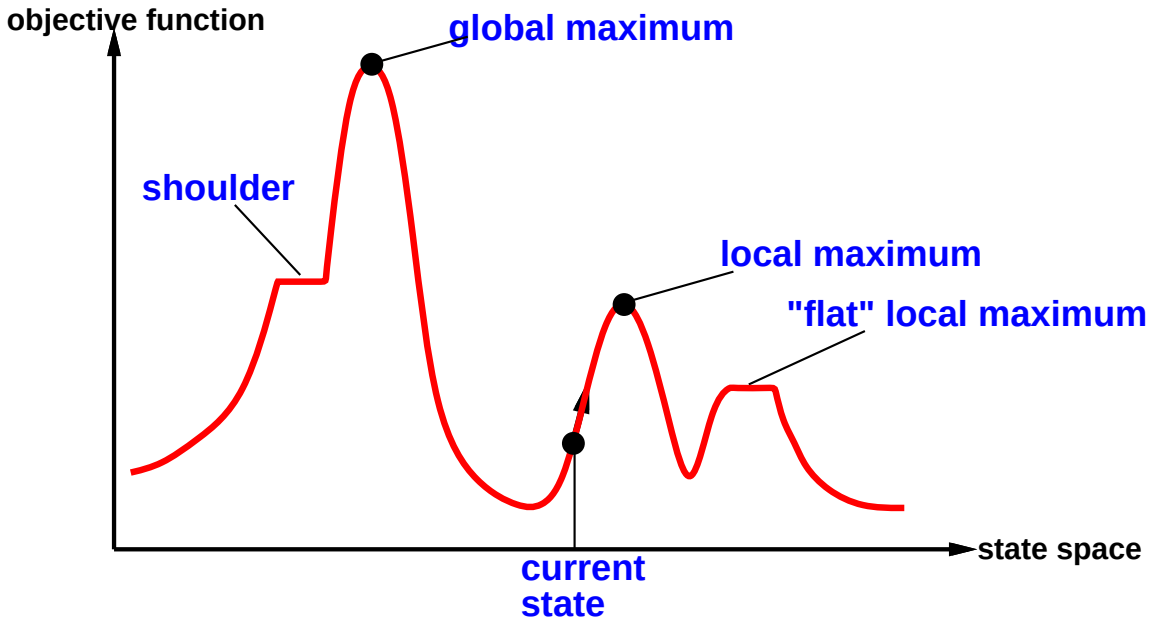
Gradietní Algoritmus (Hill-climbing)

Je důležité zohlednit **tvar stavového prostoru**



Gradietní Algoritmus (Hill-climbing)

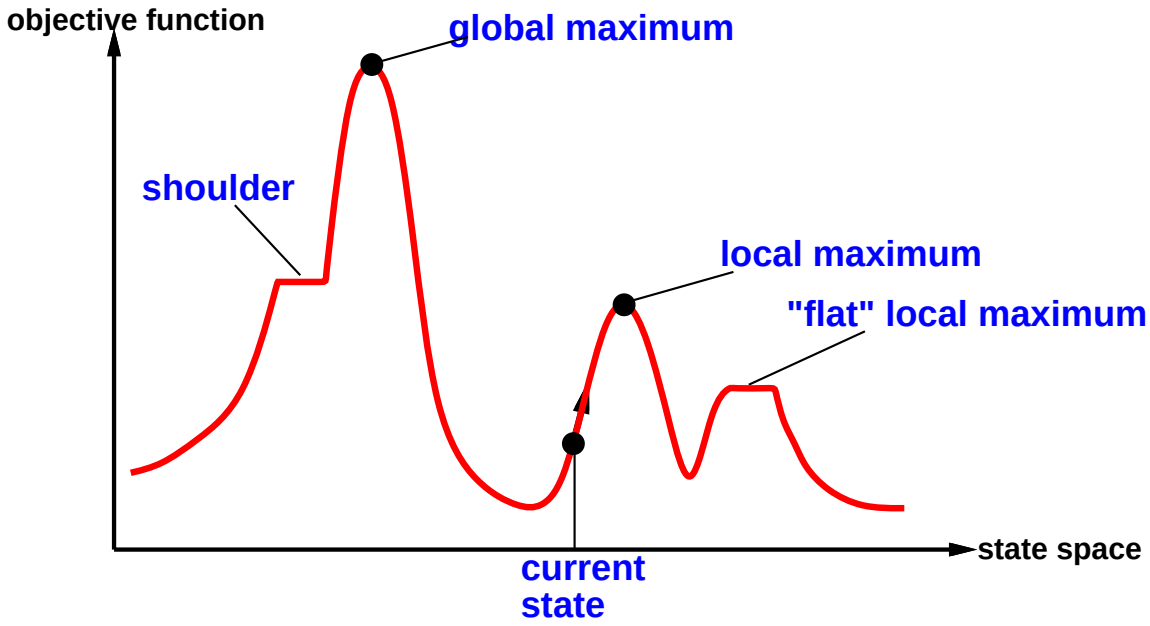
Je důležité zohlednit **tvar stavového prostoru**



- **Náhodně restartovaný gradientní algoritmus** (Random-restart hill climbing): překoná lokální extrémy tím že opakuje algoritmus s náhodnými počátečními stavy
  - pro pravděpodobnost nalezení optima p potřebujeme $1/p$ restartů
  - pro problém 8 královen, kde $p \approx 0.14$ potřebujeme 7 iterací

Gradietní Algoritmus (Hill-climbing)

Je důležité zohlednit **tvar stavového prostoru**



- **Náhodně restartovaný gradientní algoritmus** (Random-restart hill climbing): překoná lokální extrémy tím že opakuje algoritmus s náhodnými počátečními stavy
- **Náhodné úkroky** (Random sideways moves):
  (+) vyhne se ramenům (-) zacyklí se na rovných extrémech

Simulované žíhání

Klíčová myšlenka: vyhnout se lokálním extrémům tím že umožníme několik špatných tahů
ale s postupně se snižující velikostí a frekvenci

```
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)
```

Local beam search

Klíčová myšlenka: udržovat si k stavů místo jednoho a vybrat si k nejlepších následníků

- není to stejné jako k prohledávání, které běží paralelně
- Prohledávání, které nachází dobré stavy expanduje směrem k dalšímu slibnému prohledávání

Problem: často všech k prohledávání skončí na stejném lokálním extrému

Nápad: vyberme k následníků náhodně randomly, s jistou preferencí těch dobrých

Analogie s přirozeným výběrem v přírodě

Evoluční výpočty

Genetické algoritmy, **genetické programování** a další **evoluční strategie** jsou často řazeny (mylně) do kategorie metod konnekcionismu. Jedná se však o netradiční metody umělé inteligence. Souhrnně se řadí s neuronovými sítěmi do kategorie metod zvaných **softcomputing**.

Genetické algoritmy představuji speciální prohledávací technologii, která je založena na náhodné generaci většího množství stavů, jejich následné masivní evoluci.

- každý stav je reprezentován jako chromozóm (klasicky jako binární vektor, lze však i celočíselný vektor, matice, strom, ...)
- každý chromozóm je ohodnocen tzv. *fitness* funkcí – pouze kvalitní chromozómy mají šanci přežít do další generace
- v dlouhodobém horizontu stoupá průměrná hodnota fitness a nekvalitní chromozómy vymírají

Softcomputing (vsuvka)

- http://en.wikipedia.org/wiki/Soft_computing
- do Softcomputingu patří:
  - Neuronové sitě
  - Fuzzy systémy
  - Evoluční výpočtu
  - Swarm inteligence (Inteligence založená na chování hejna)
  - Pravděpodobnostní modely (např. Baysovské sítě) ??

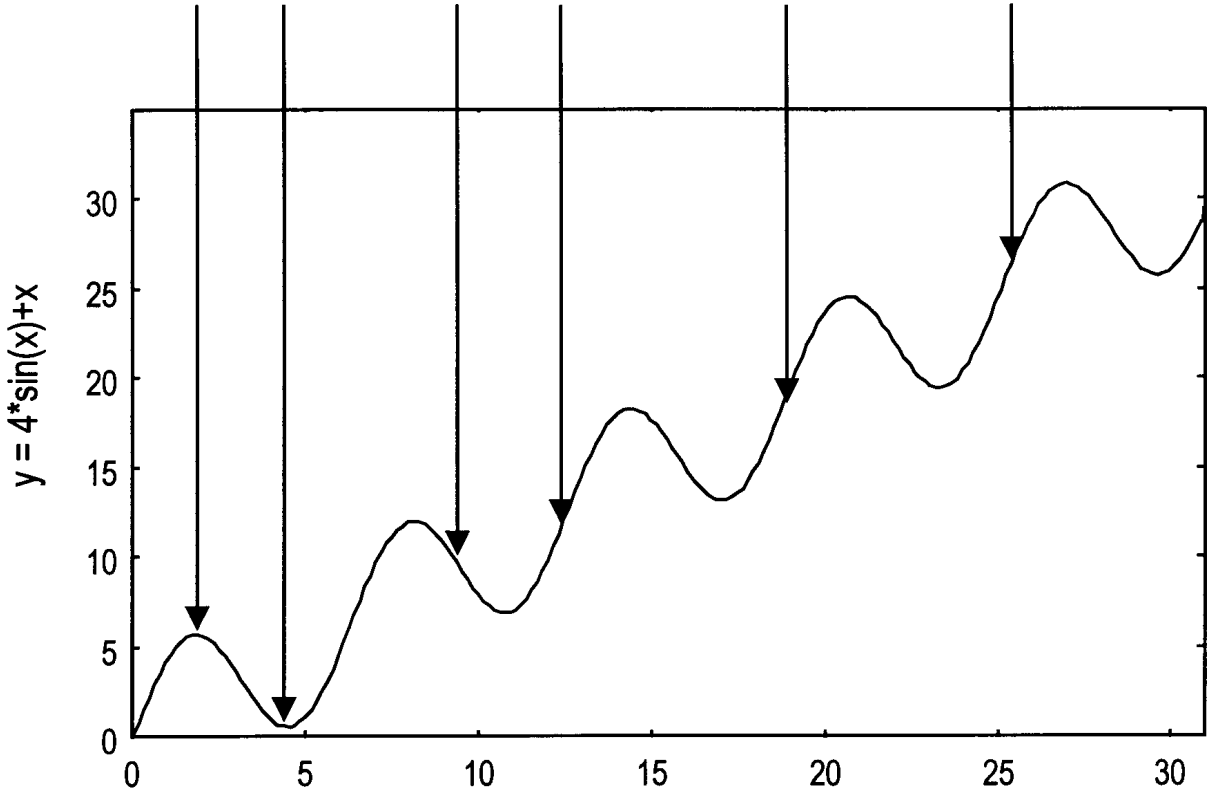
Evoluční výpočty

Genetické algoritmy, **genetické programování** a další **evoluční strategie** jsou často řazeny (mylně) do kategorie metod konnekcionismu. Jedná se však o netradiční metody umělé inteligence. Souhrnně se řadí s neuronovými sítěmi do kategorie metod zvaných **softcomputing**.

Genetické algoritmy představuji speciální prohledávací technologii, která je založena na náhodné generaci většího množství stavů, jejich následné masivní evoluci.

- každý stav je reprezentován jako chromozóm (klasicky jako binární vektor, lze však i celočíselný vektor, matice, strom, ...)
- každý chromozóm je ohodnocen tzv. *fitness* funkcí – pouze kvalitní chromozómy mají šanci přežít do další generace
- v dlouhodobém horizontu stoupá průměrná hodnota fitness a nekvalitní chromozómy vymírají

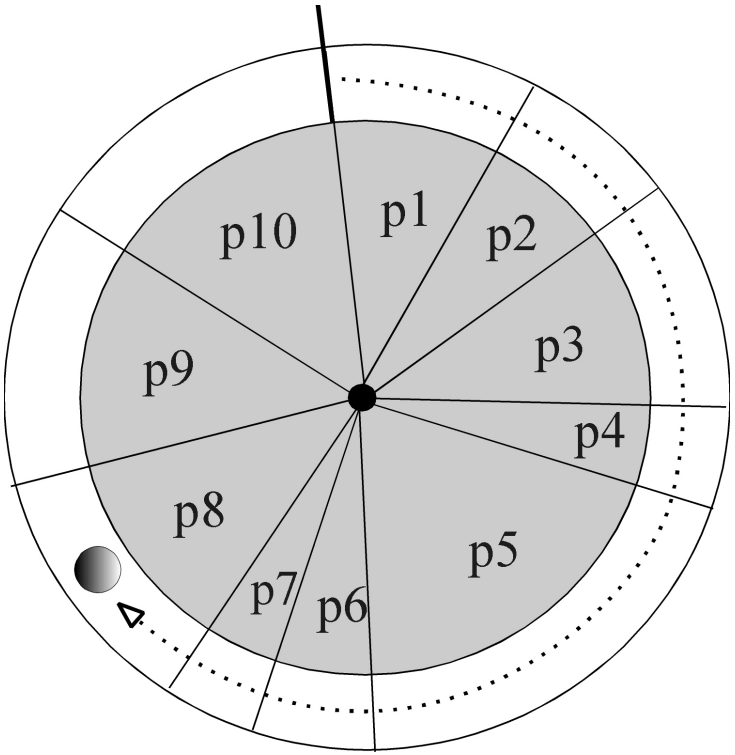
Evoluční výpočty



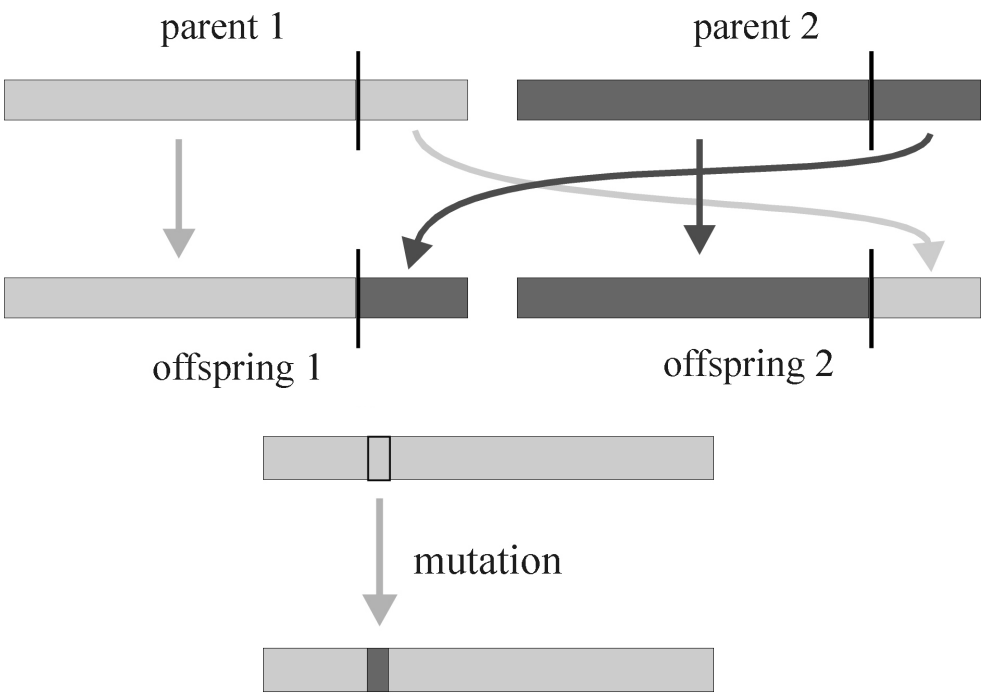
Standartní GA

```
t=0
Initialize P(t)
while (not termination-condition) do
    begin
        Evaluate P(t)
        t=t+1
        Select P(t) from P(t-1)
        Recombine
    end
```

Výběr (selection)

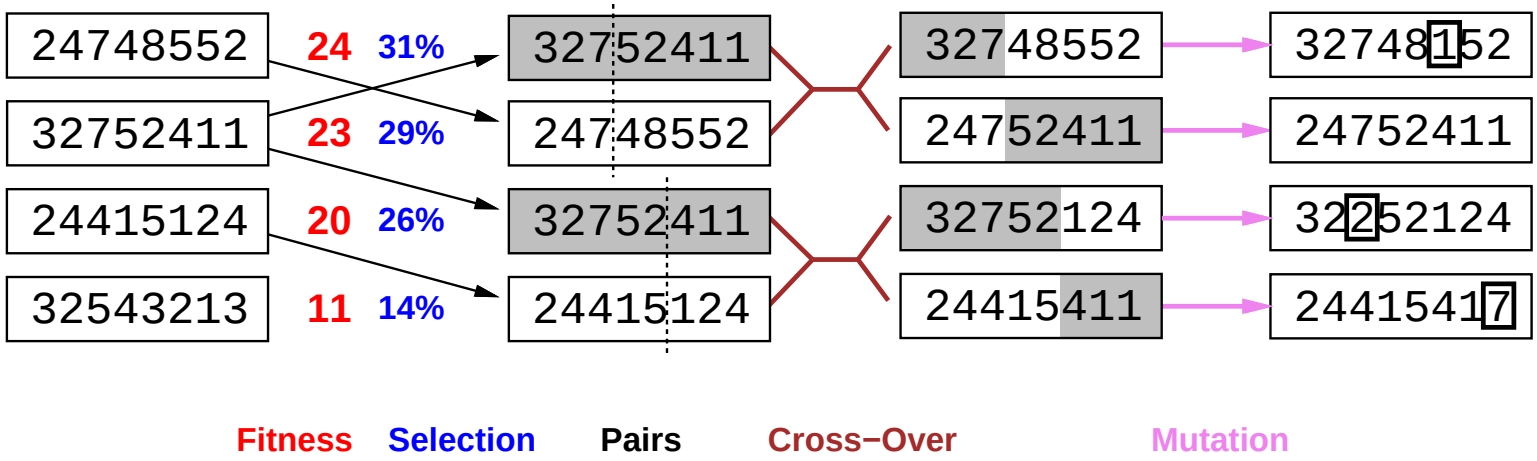


Mutace a křížení (recombination)



Genetické algoritmy

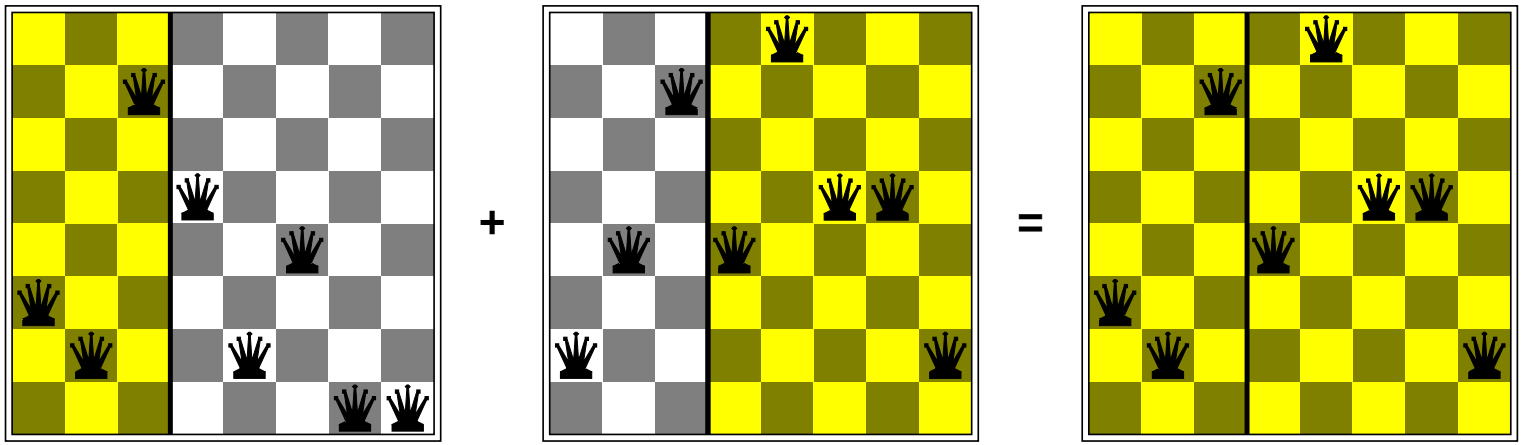
= stochastický local beam search + generování následníků z *dvojic* stavů



Genetické algoritmy

GAs požadují kódování stavů do řetězců (ale také matic, stromů či programů)

Křížení pomáhá v případě, že podřetězce obsahují smysluplné komponenty



Problém splnění omezujících podmínek

Constraint satisfaction problems (CSPs)

Mějme standardní prohledávací problém:

- **stav** je "black box" – libovolná datová struktura, která podporuje funkce `goal_test`, `eval`, `successor`

Problém splnění omezujících podmínek

Constraint satisfaction problems (CSPs)

Mějme standardní prohledávací problém:

- **stav** je "black box" – libovolná datová struktura, která podporuje funkce `goal_test`, `eval`, `successor`

V problému splnění omezujících podmínek je pak

- **stav** definován pomocí *proměnných* X_i nabývajících *hodnot* ze specifické *domény* D_i (oboru hodnot)
- **goal_test** je definován jako množina *omezení*, které specifikují možné kombinace hodnot pro podmnožiny proměnných

Problém splnění omezujících podmínek

Constraint satisfaction problems (CSPs)

Mějme standardní prohledávací problém:

- **stav** je "black box" – libovolná datová struktura, která podporuje funkce `goal_test`, `eval`, `successor`

V problému splnění omezujících podmínek je pak

- **stav** definován pomocí *proměnných* X_i nabývajících *hodnot* ze specifické *domény* D_i (oboru hodnot)
- **goal_test** je definován jako množina *omezení*, které specifikují možné kombinace hodnot pro podmnožiny proměnných

jedná se o jednoduchý příklad popisu problému pomocí *formálního jazyka*

Umožňuje použití *obecného* (*general-purpose*) algoritmu s většími výpočetními možnostmi než standardní prohledávací algoritmy

Příklad: SUDOKU

- http://www.websudoku.com/
- Každá Sudoku má jedno unikátní logické řešení, kterého lze dosáhnout logicky, bez hádání.
- Každý řádek, každý sloupec i každá buňka 3x3 musí obsahovat všech 9 číslic

				9		3		
	2		6		1	5	9	
3			5		8		2	4
	9					4	8	
6			2		9			1
	5	2					3	
9	7		1		5			3
	3	8	7		4		1	
		5		8				

Příklad: SUDOKU

- http://www.websudoku.com/
- Každá Sudoku má jedno unikátní logické řešení, kterého lze dosáhnout logicky, bez hádání.
- Každý řádek, každý sloupec i každá buňka 3x3 musí obsahovat všech 9 číslic

				9		3		
	2		6		1	5	9	
3			5		8		2	4
	9					4	8	
6			2		9			1
	5	2					3	
9	7		1		5			3
	3	8	7		4		1	
		5		8				

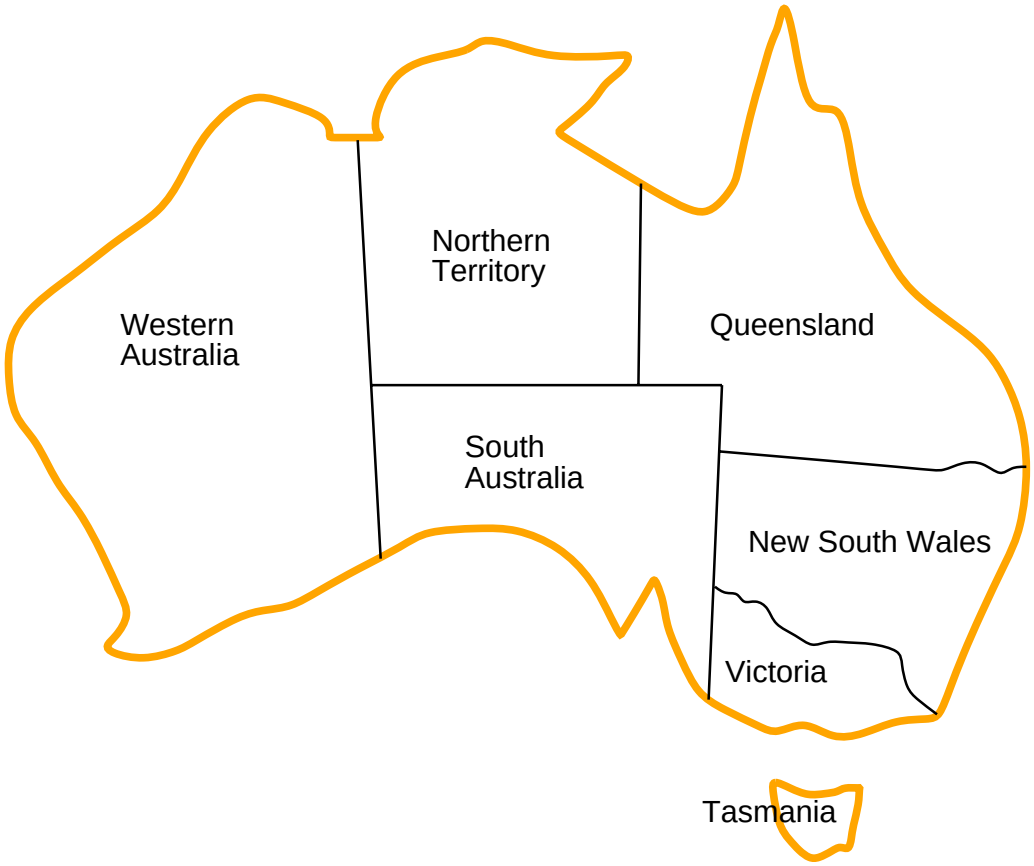
Proměnné $X_{1,1}, ...X_{9,9}$

Domény $D_i = \{1, .., 9\}$

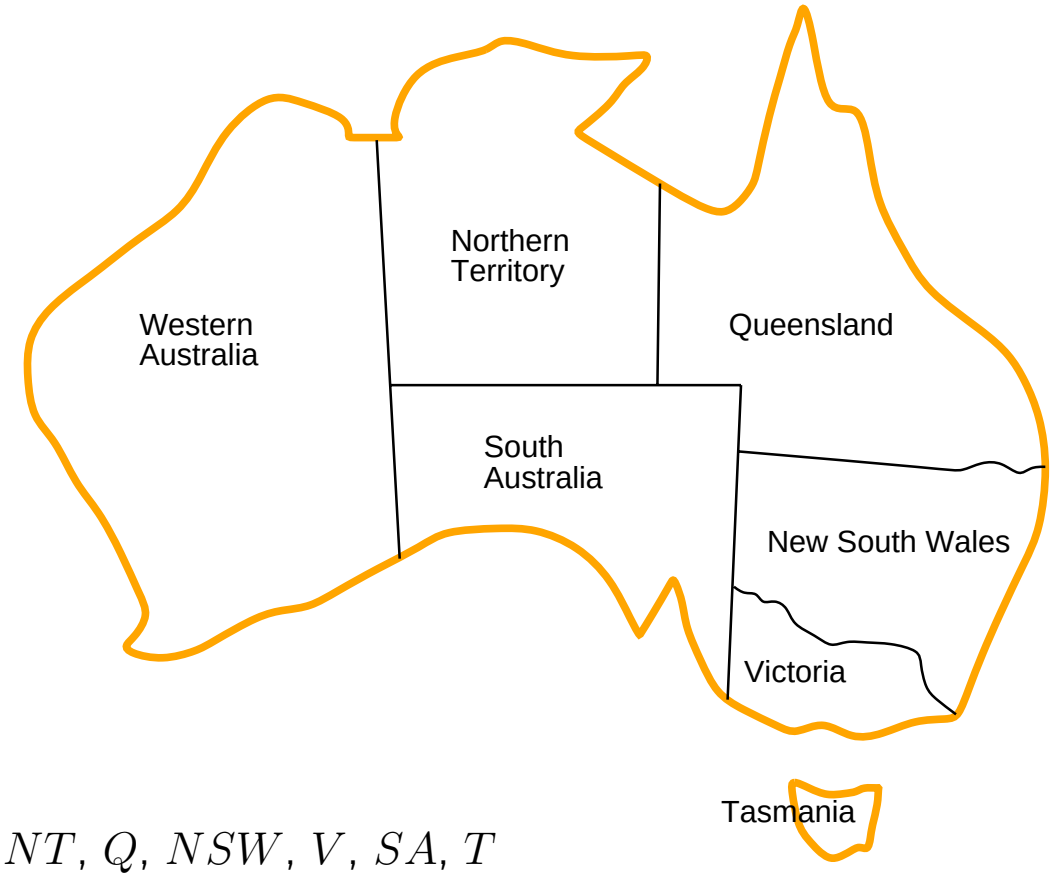
Omezení: $X_{1,1} \neq X_{1,2}, ..., X_{1,1} \neq X_{1,9}, ...$ ve sloupcích, řádcích a blocích

Řešení: úplné přiřazení hodnot všem proměnným za splnění omezení

Příklad: Obarvení mapy



Příklad: Obarvení mapy



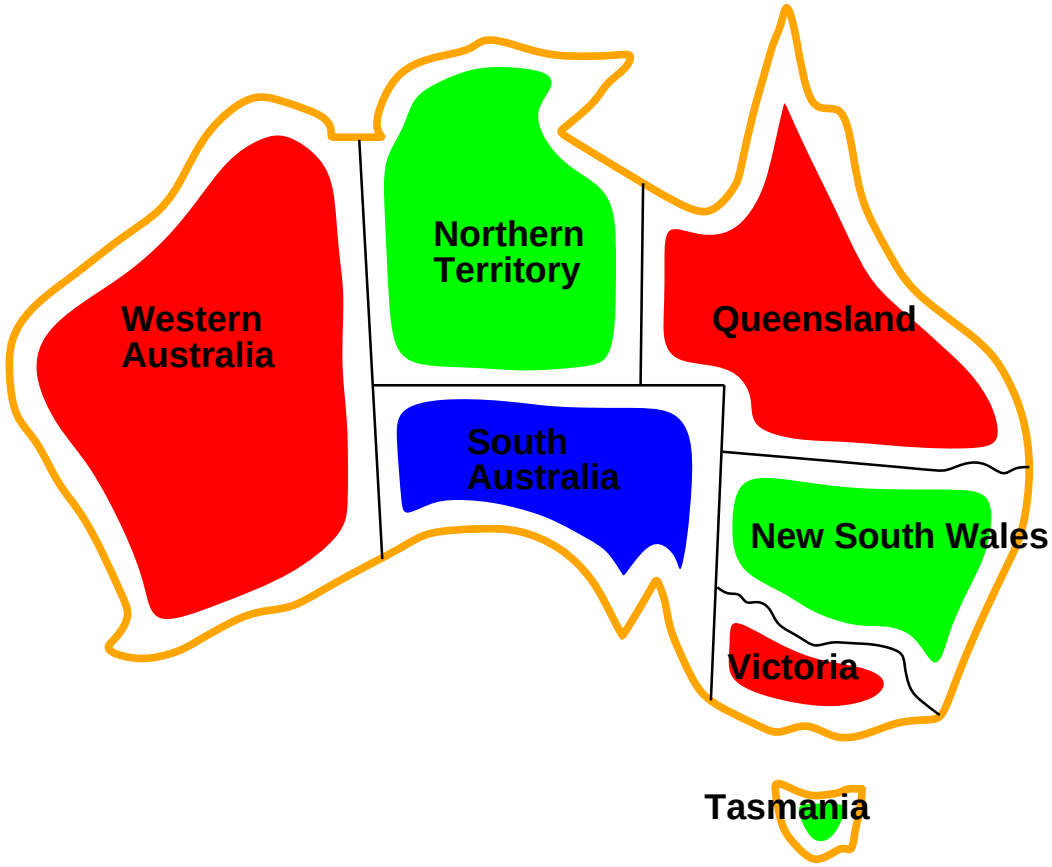
Proměnné WA, NT, Q, NSW, V, SA, T

Domény $D_i = \{red, green, blue\}$

Omezení: sousední oblasti se musejí odlišovat barvami e.g., $WA \neq NT$ (pokud to reprezentace

umožňuje), nebo $(WA, NT) \in \{(red, green), (red, blue), (green, red), (green, blue), \ldots\}$

Příklad: Obarvení mapy

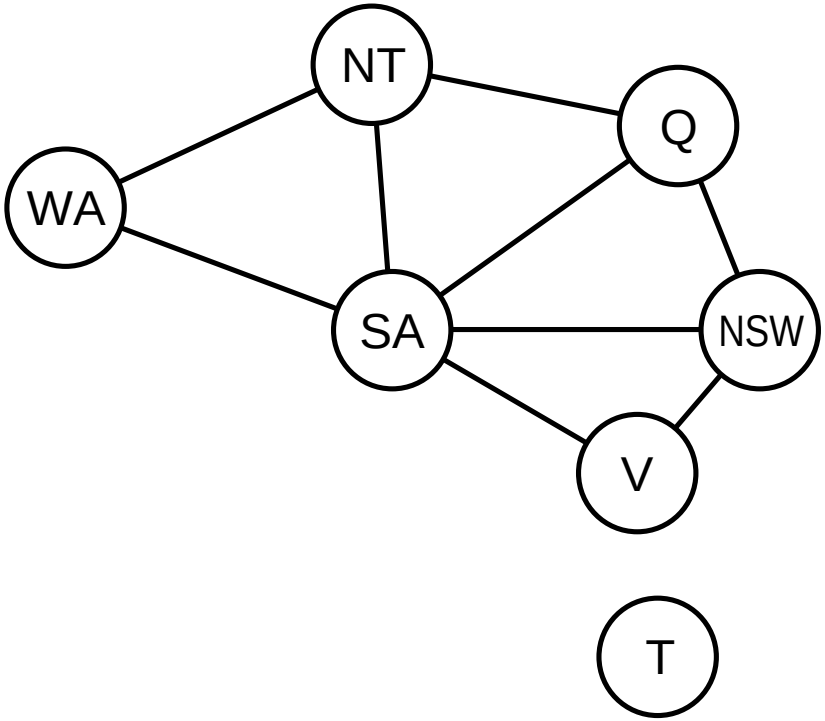


Řešení je přiřazení barev, které splňují daná omezení, např.,
$\{WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = green\}$

Graf Omezení

Binární CSP: každé omezení specifikuje vztah mezi maximálně dvěmi proměnnými

Graf Omezení: uzly jsou proměnné, hrany jsou omezení



Obecný CSP algoritmus používá grafovou strukturu za účelem urychlení procesu prohledávání. např. umožní detekovat, že Tasmania je nezávislý subproblem!

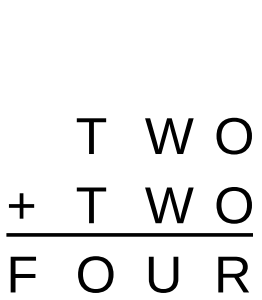
Variace CSPs

- **Diskrétní proměnné:**
  - z konečné domény velikosti $d \implies O(d^n)$ pro všechna přiřazení
    - např., Boolean CSPs, incl. Boolean satisfiability (NP-complete)
  - z nekonečné domény (integers, strings, ...)
    - např., rozvrhování, proměnné pro každý start/end den pro každou úlohu
    - potřeba zavedení **jazyka omezení**, např., $StartJob_1 + 5 \leq StartJob_3$
    - **lineární** omezení jsou řešitelná, **nelineární** omezení jsou nerozhodnutelná
- **Spojité proměnné**
  - např.: start/end čas pro práci Hubblova Teleskopu
  - lineární omezení jsou řešitelná v polynomiáním čase pomocí LP metod

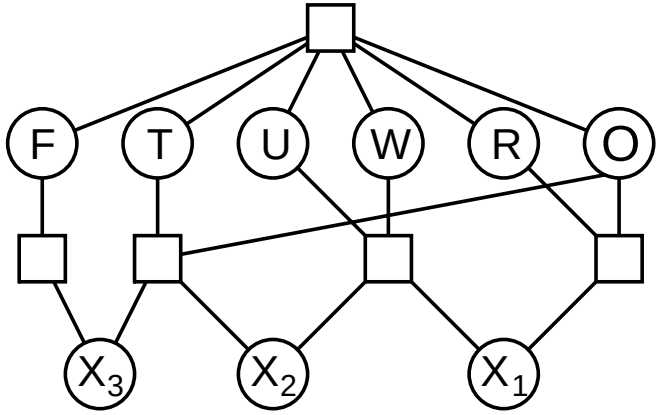
Variace CSPs

- **Unární** omezení pracuje výhradně s jednou proměnnou,
  e.g., $SA \neq green$
- **Binární** omezení pracuje s dvojicí proměnných,
  např., $SA \neq WA$
- **Omezení vyššího řádu** zahrnuje 3 a více proměnných,
  např., cryptaritmetika, sudoku, omezení relace mezi sloupci
- **Preference** (soft constraints), např., red je lepší než $green$
  často reprezentované pomocí ceny za každé přiřazení hodnoty proměnné
  $\rightarrow$ vede na problémy optimalizace za omezujících podmínek
  (constrained optimization problems)

Příklad: Cryptarithmetic



(a)



(b)

Proměnné: $F\ T\ U\ W\ R\ O\ X_1\ X_2\ X_3$

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

Omezení

$\texttt{alldiff}(F, T, U, W, R, O)$

$O + O = R + 10 \cdot X_1$, etc.

Problémy CSP v reálném světě

- Přiřazovací (assignment) problémy
  např., kdo učí jaký předmět
- Rozvrhovací (timetabling) problémy
  např., jaký předmět se učí kdy a kde?
- Rozvrhovací (Factory scheduling) problém
  jak narozvrhovat vhodné operace na vhodné stroje ve vhodný čas
  při zachování vzájemné provázanosti strojů i operací
- Konfigurační problém
  problém sestavení hardwaru, problém konfigurace sítě
- Transportní problém
  problém rozvrhování průběžných logistických operací
- Problém prostorového uspořádání (Floorplanning)

Mnohé reálné problémy obsahují proměnné s oborem reálných čísel

Specifikace prohledávacího algoritmu pro CSP

Začněme s nejjednoduším algoritmem a postupně ho zlepšíme

Stavy jsou definovány pomocí dosud přiřazených hodnot proměnným

- **Počáteční stav**: prázdné přiřazení, $\emptyset$
- **Operátor expanze**: přiřadí hodnotu jedné nepřiřazené proměnné
  tak, aby nedošlo ke konfliktu se současným přiřazením.
  $\Longrightarrow$ vrátí FAIL když nelze expandovat
- **Goal test**: aktuální přiřazení je úplné

Specifikace prohledávacího algoritmu pro CSP

Začněme s nejjednoduším algoritmem a postupně ho zlepšíme

Stavy jsou definovány pomocí dosud přiřazených hodnot proměnným

- **Počáteční stav**: prázdné přiřazení, $\emptyset$
- **Operátor expanze**: přiřadí hodnotu jedné nepřiřazené proměnné
  tak, aby nedošlo ke konfliktu se současným přiřazením.
  $\Longrightarrow$ vrátí FAIL když nelze expandovat
- **Goal test**: aktuální přiřazení je úplné

Vlastnosti:
– tento algoritmus funguje pro všechny CSP
– každé řešení se nachází v hloubce n s n proměnnými
  $\Longrightarrow$ lze použít depth-first search
– cesta není důležitá
– faktor větvení v hloubce l je $b = (n - l)d$, tedy $n!d^n$ uzlů!!!!

Backtracking search (zpětné prohledávání)

- Backtracking search – prohledávání založené na zpětném vyhledávání, nebo na inteligentním navracení se

Přiřazení proměnných je **komutativní**, t.j.,

$$[\text{nejprve } WA = red \text{ pak } NT = green]$$

$$\text{je stejné jako}$$

$$[\text{nejprve } NT = green \text{ pak } WA = red]$$

Je třeba přiřadit právě jednu proměnnou v každém uzlu

$\Longrightarrow$ $b = d$ a existuje d^n uzlů kde n je velikost domény

Prohledávání do hloubky pro CSPs s přiřazováním jedné proměnné se nazývá **backtracking search**

Backtracking search je základní neinformovaný algoritmus pro řešení CSPs

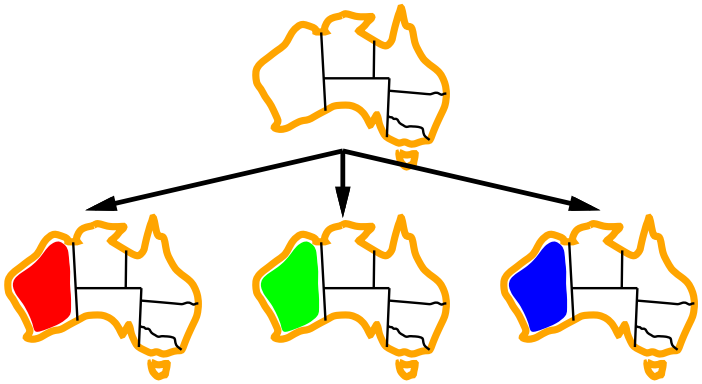
Může řešit problém n-královen s přibližně $n \approx 25$

Zpětné prohledávání

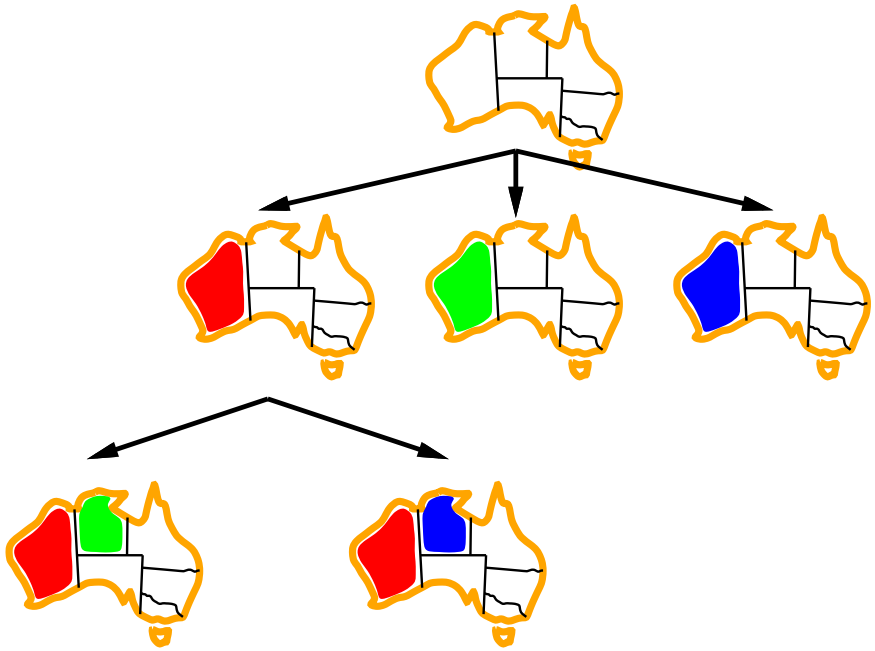
```
function BACKTRACKING-SEARCH(csp) returns solution/failure
    return RECURSIVE-BACKTRACKING({ }, csp)

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
```

Zpětné prohledávání

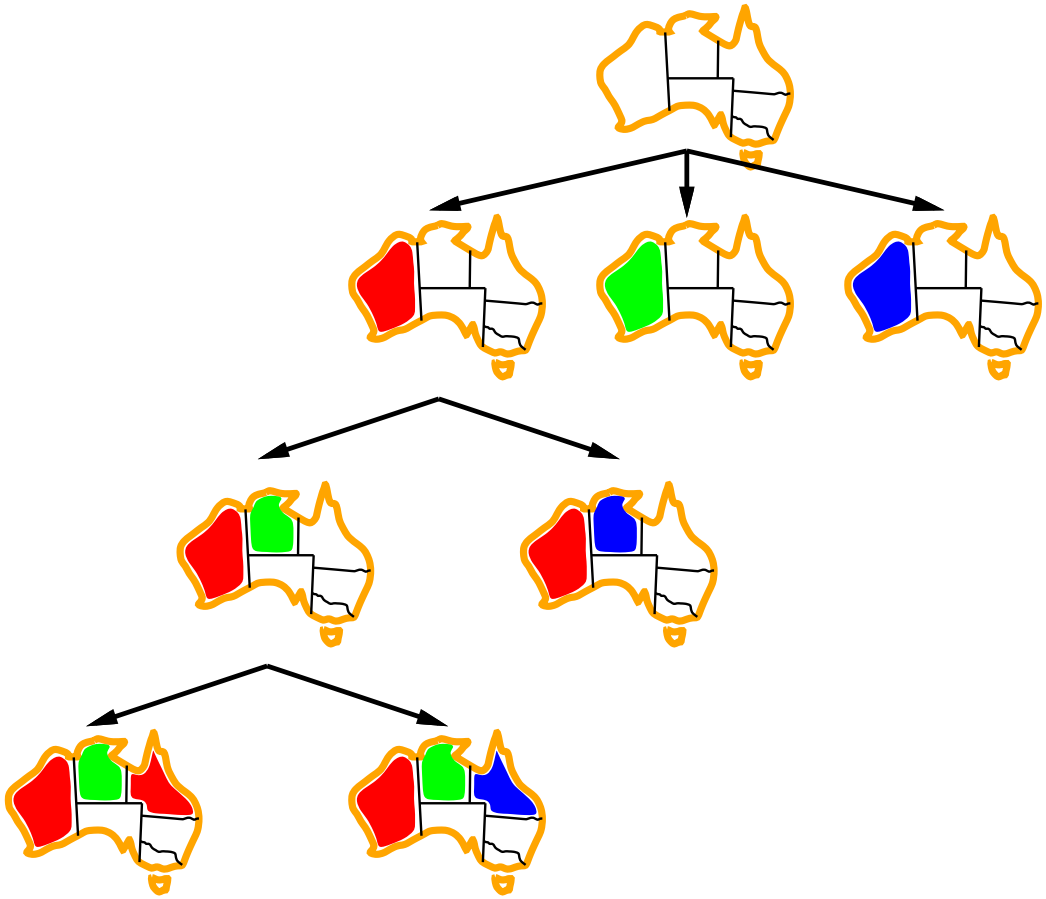
Zpětné prohledávání



Zpětné prohledávání



Zpětné prohledávání



Zlepšení efektivity zpětného prohledávání

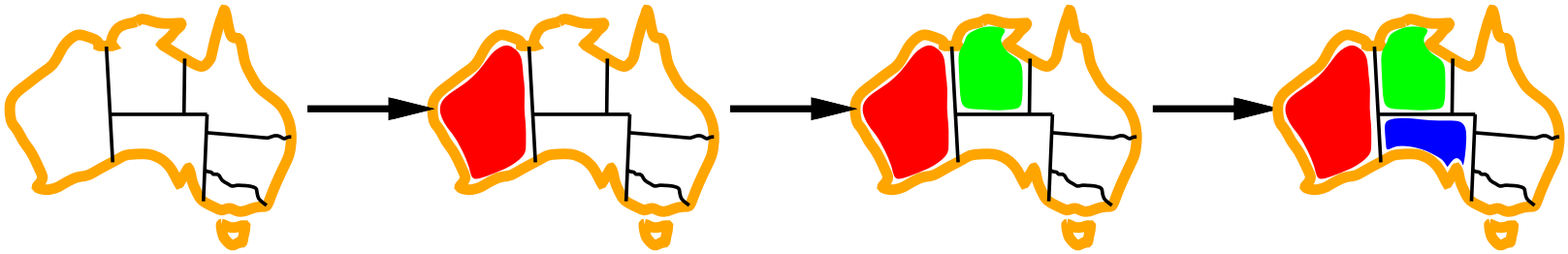
Obecné metody lze zlepšit, bude-li se algoritmus schopen správně rozhodnout:

1. kterou proměnnou přiřadit v příštím kole?
2. v jakém pořadí bychom měli zkoušet hodnoty?
3. můžeme dopředu předvídat nevyhnutelné selhání přiřazení (failures)?
4. lze využít předem známá struktura problému?

Minimální zbývající hodnota (MRV)

Minimální zbývající hodnota (**Minimum remaining values**):

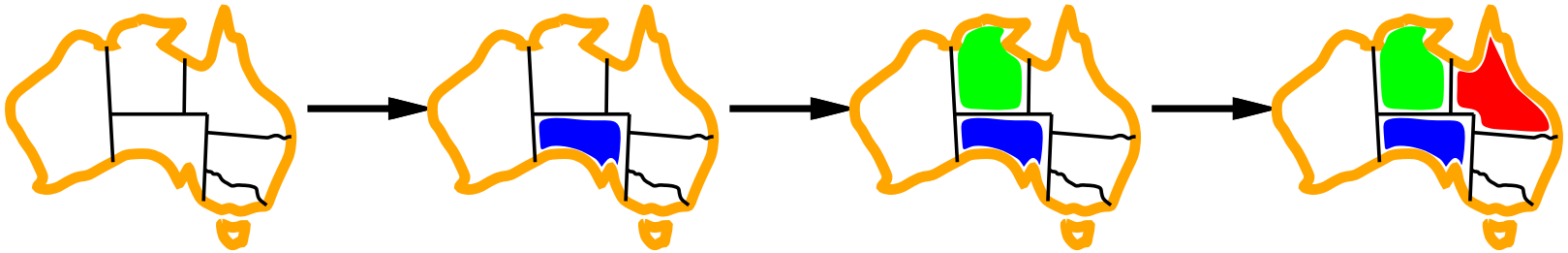
- vyber proměnnou s nejmenším množstvím přípustných hodnot,
- nazývána rovněž *most constrained variable* nebo *fail-first* heuristika.



Heuristika podle stupně omezení

Alternativa k MRP je heuristika podle stupně omezení (**degree heuristic**:)

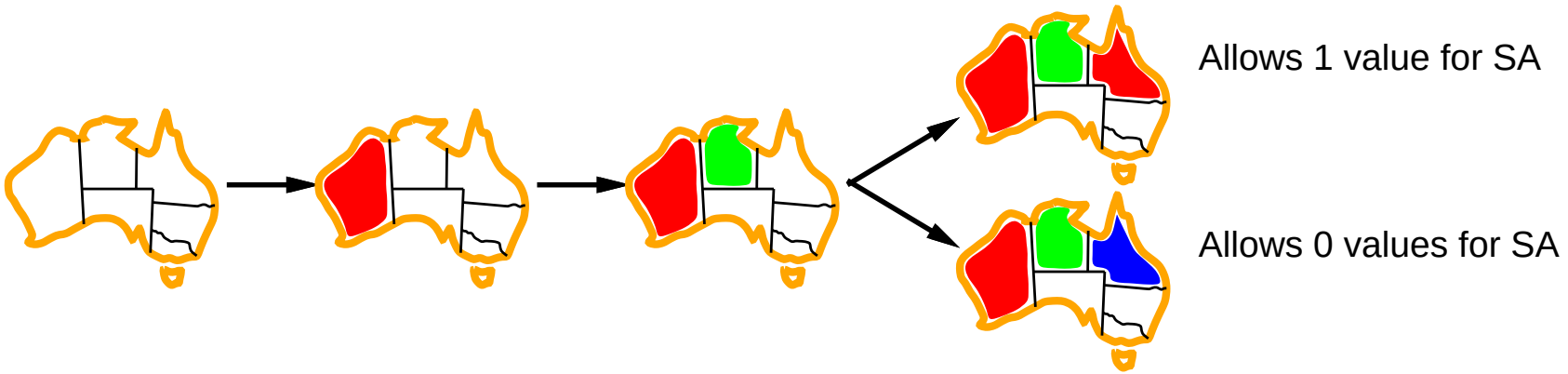
1. vyber proměnnou s nejvíce omezeními na zbývajících proměnných.
2. Omezuje faktor větvení na budoucích proměnných.



Nejméně omezující hodnota

pro danou proměnnou vyber hodnotu, která ji nejméně omezuje (**least-constraining value**):

- tato hodnota vyřadí nejméně hodnot pro ostatní proměnné

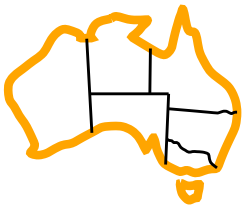


Za kombinace těchto heuristik lze řešit až 1000 královen

Dopředná kontrola (Forward checking)

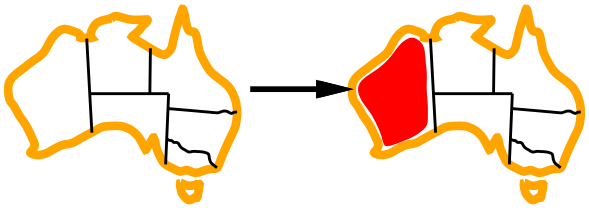
Hlavní myšlenka: Pamatovat si přípustné zbývající hodnoty pro nepřiřazené proměnné

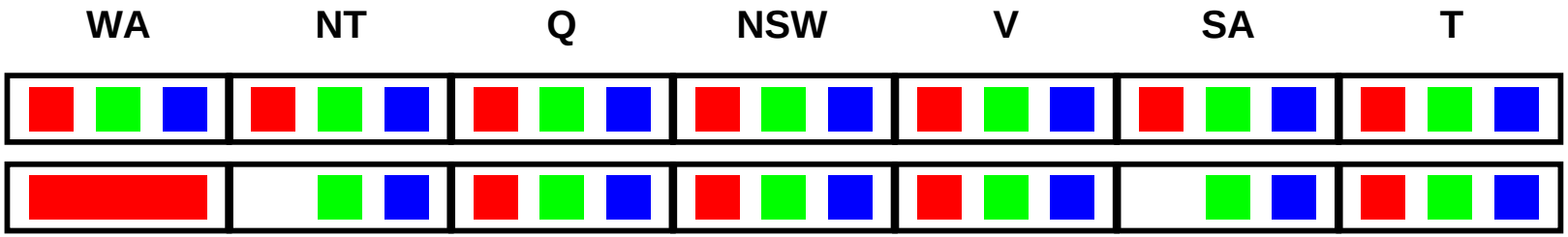
Ukončit prohledávání, když už žádná proměnná nemá žádnou legální hodnotu.



Dopředná kontrola (Forward checking)

Hlavní myšlenka: Pamatovat si přípustné zbývající hodnoty pro nepřiřazené proměnné
Ukončit prohledávání, když už žádná proměnná nemá žádnou legální hodnotu.

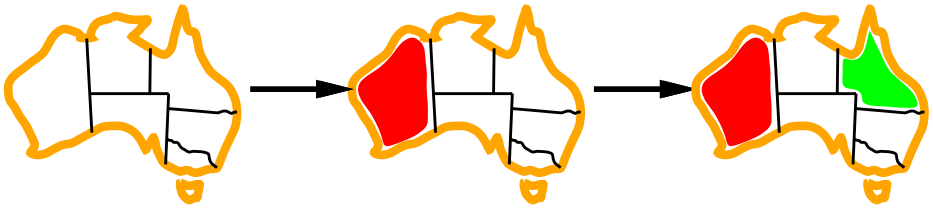


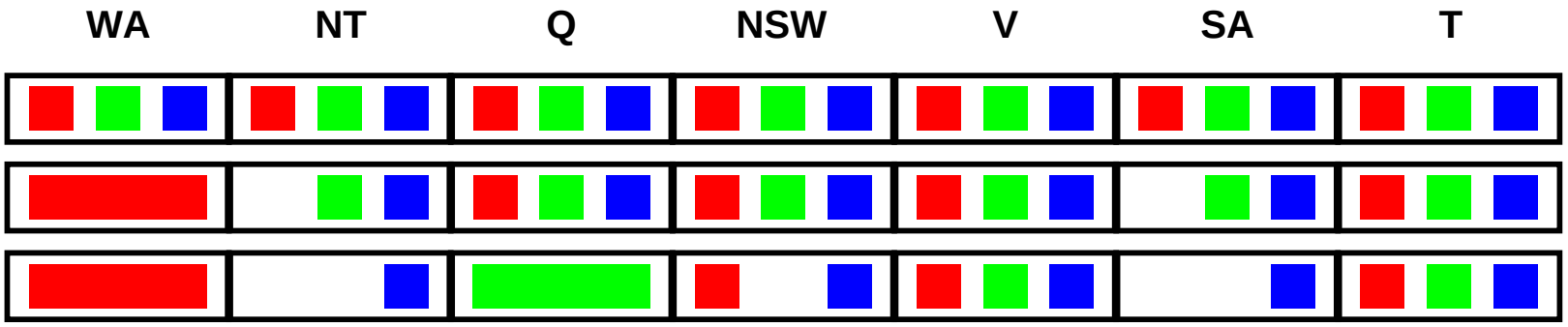


Dopředná kontrola (Forward checking)

Hlavní myšlenka: Pamatovat si přípustné zbývající hodnoty pro nepřiřazené proměnné

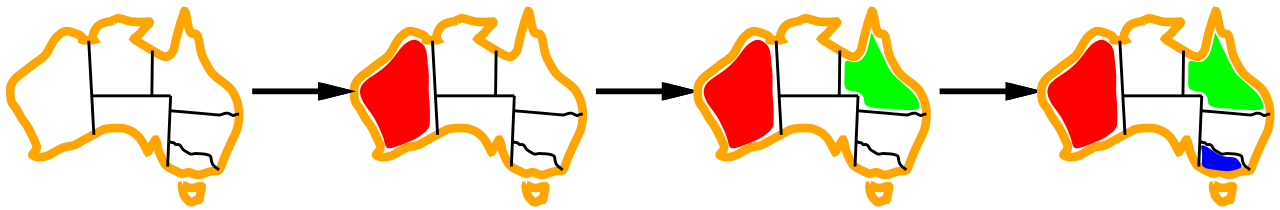
Ukončit prohledávání, když už žádná proměnná nemá žádnou legální hodnotu.

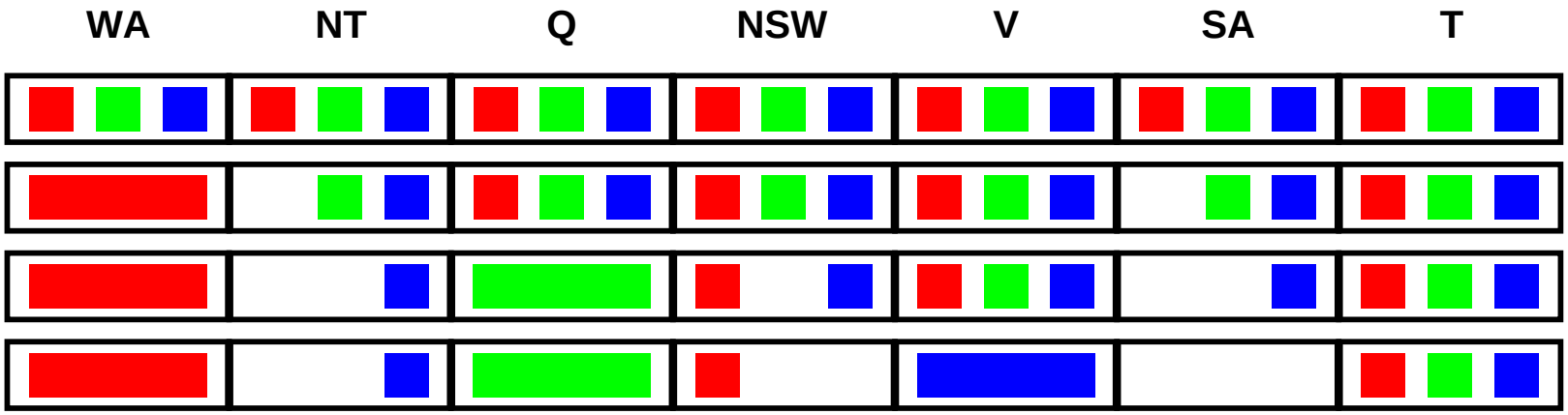




Dopředná kontrola (Forward checking)

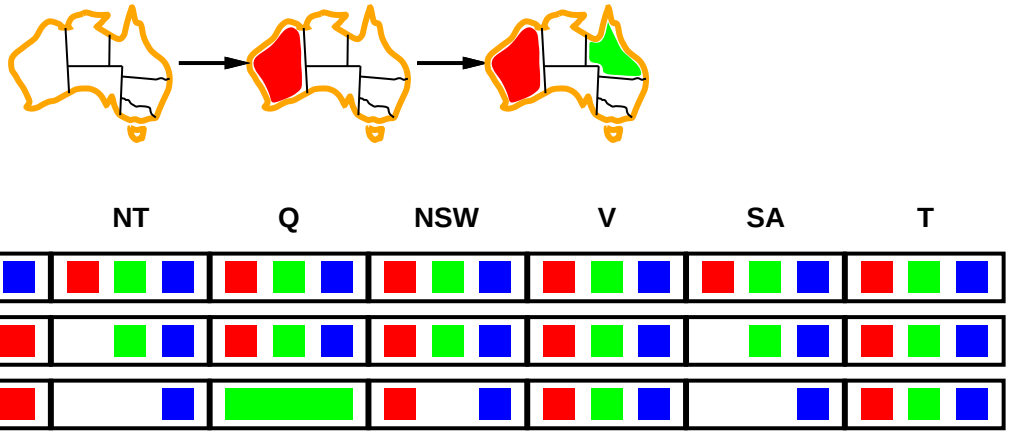
Hlavní myšlenka: Pamatovat si přípustné zbývající hodnoty pro nepřiřazené proměnné
Ukončit prohledávání, když už žádná proměnná nemá žádnou legální hodnotu.





Šíření omezení (Constrain propagation)

Dopředná kontrola šíří informace od přiřazených k nepřiřazeným proměnným, ale nedetekuje všechna budoucí selhání:



NT and SA nemohou být zároveň modré!

Šíření omezení opakovaně vynucuje omezení lokálně

Porovnání

- mapa USA: barvení mapy USA pomocí 4 barev
- n-královen: řešení 2 až 50-královen
- zebra: logická hádanka (viz následující slide)
- random1, random 2: umělé problémy

problém	BCK	BCK+MRV	FCH	FCH+MRV
mapa USA	($>$ 1.000K)	($>$ 1.000K)	2K	60
n-královen	($>$ 40.000K)	13.500K	($>$ 40.000K)	817K
zebra	3.859K	1K	35K	0.5K
random1	4.15K	3K	26K	2K
random2	942K	27K	77K	15K

měříme medián počtu kontroly konzistence (přes 5 testů), výraz v závorce znamená, že po tolika operacích nebylo nalezeno řešení.

Porovnání

ZEBRA: Consider the following logic puzzle: In five houses, each with a different color, live 5 persons of different nationalities, each of whom prefer a different brand of cigarette, a different drink, and a different pet. Given the following facts, the question to answer is "Where does the zebra live, and in which house do they drink water?"

The Englishman lives in the red house.

The Spaniard owns the dog.

The Norwegian lives in the first house on the left.

Kools are smoked in the yellow house.

The man who smokes Chesterfields lives in the house next to the man with the fox.

The Norwegian lives next to the blue house.

The Winston smoker owns snails.

The Lucky Strike smoker drinks orange juice.

The Ukrainian drinks tea.

The Japanese smokes Parliaments.

Kools are smoked in the house next to the house where the horse is kept.

Coffee is drunk in the green house.

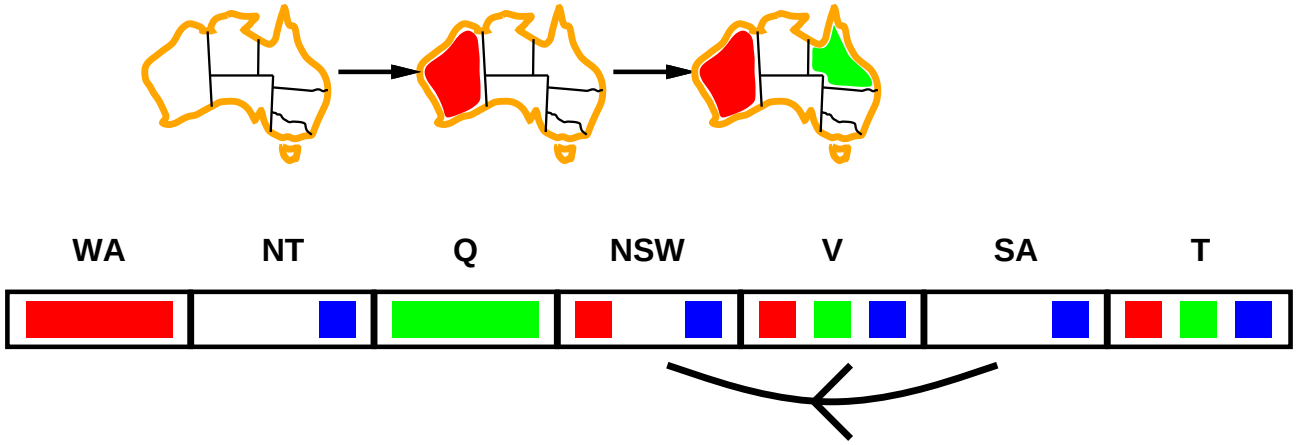
The Green house is immediately to the right (your right) of the ivory house.

Milk is drunk in the middle house.

Konzistence hran

Nejjednoduší forma šíření umožňuje konzisetnci hran

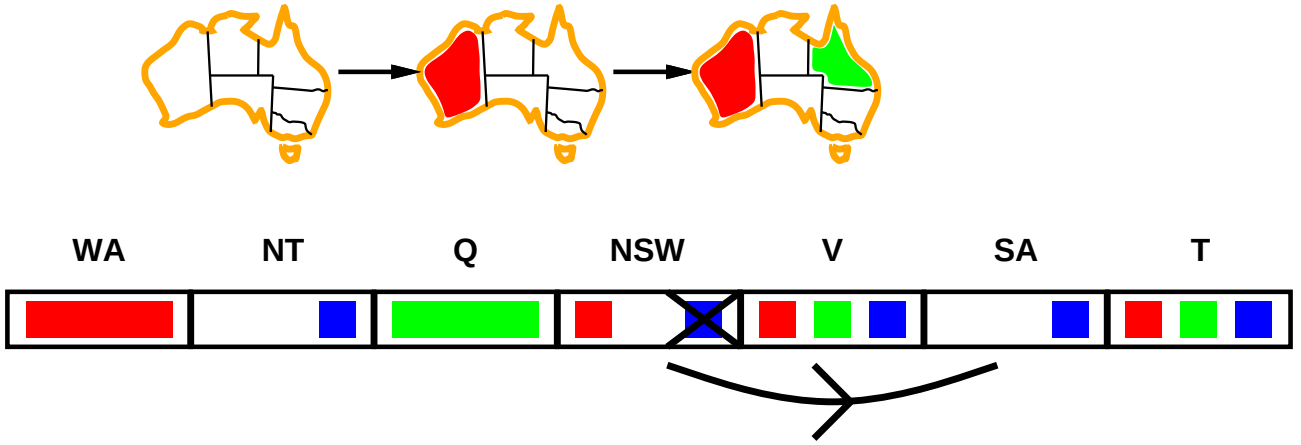
$X \rightarrow Y$ je konzistentní právě když
pro *každou* hodnotu x z X existuješ *nějaké* přípustné y



Konzistence hran

Nejjednoduší forma šíření umožňuje konzisetnci hran

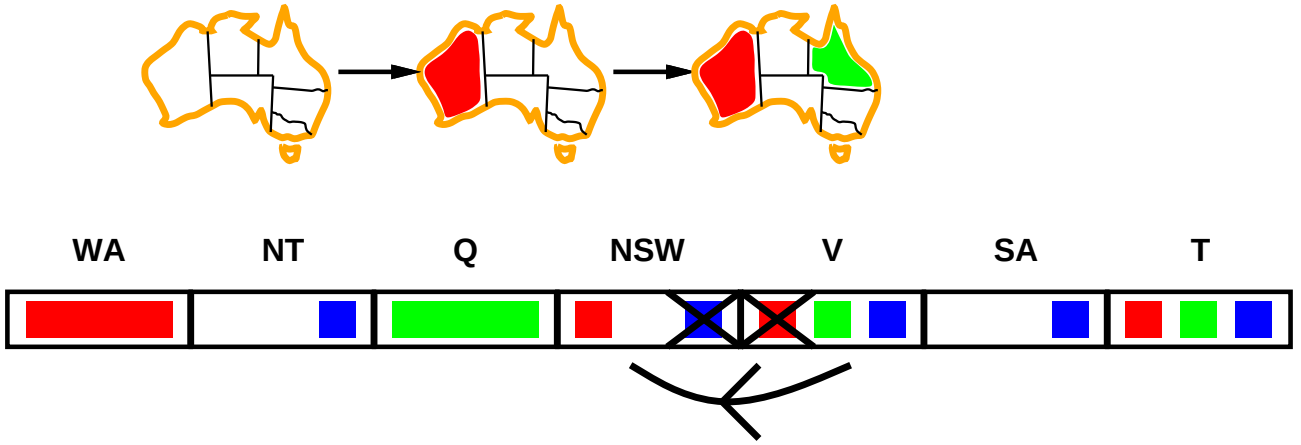
$X \rightarrow Y$ je konzistentní právě když
pro *každou* hodnotu x z X existuje *nějaké* přípustné y



Konzistence hran

Nejjednoduší forma šíření umožňuje konzisetnci hran

$X \rightarrow Y$ je konzistentní právě když
pro *každou* hodnotu x z X existuje *nějaké* přípustné y

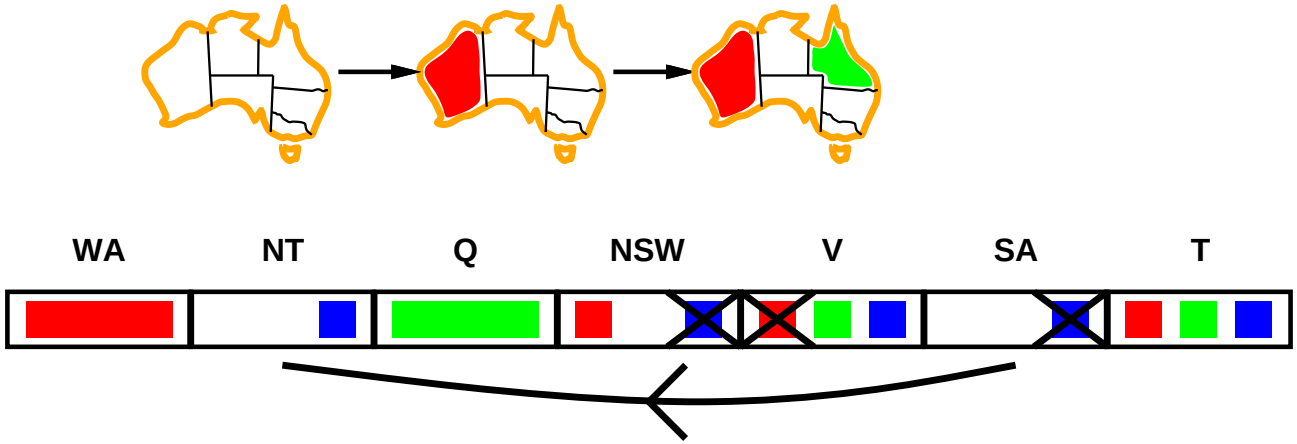


V případě, že X ztratí hodnotu, souseda X je třeba znovu zkontrolovat

Konzistence hran

Nejjednoduší forma šíření umožňuje konzisetnci hran

$X \rightarrow Y$ je konzistentní právě když
pro *každou* hodnotu x z X existuješ *nějaké* přípusné y



V případě, že X ztratí hodnotu, souseda X je třeba znovu zkontrolovat

Konzistence hran detekuje selhání dříve než dopředná kontrola

Může běžet jako preprocessor or nebo po každém přiřazení

Algoritmus konzistence hran

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 the constraint $X_i \leftrightarrow X_j$
      then delete x from DOMAIN$[X_i]$; *removed* $\leftarrow$ *true*
  return *removed*

$O(n^2d^3)$, lze redukovat do $O(n^2d^2)$ (ale detekce *všech* je NP-hard)

Použití algoritmů lokálního prohledávání pro CSP

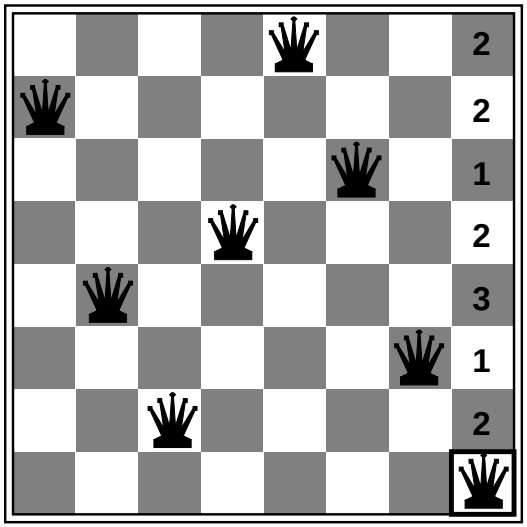
- Gradientní metody pracují zpravidla s **množinou úplných konfigurací**, t.j., všechny proměnné jsou přiřazeny
- Aplikace na problém CSPs:
  - umožnit práci se stavy s nesplněnými omezeními
  - zavést operátory *znovupřiřazení* (změny) hodnot proměnných
- **Výběr proměnných:** náhodně vybrat konfliktní proměnou
- **Výběr hodnoty:** například pomocí **min-conflicts** heuristiky:
  - vyber takovou hodnotu, která bude porušovat minimální počet omezení
    t.j., gradientní prohledávání s $h(n) =$ počet nesplněných omezení

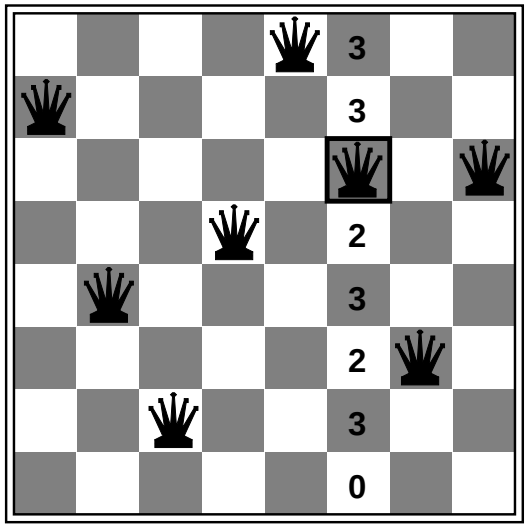
Algoritmus `min-conflict`

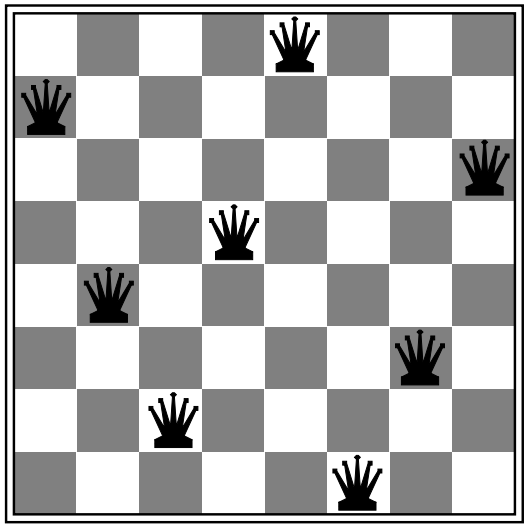
function MIN-CONFLICTS(csp, max_steps) **returns** a solution or failure
  inputs: csp, a constraint satisfaction problem
    max_steps, the number of steps allowed before giving up

  $current \leftarrow$ an initial complete assignment for csp
  for i = 1 to max_steps **do**
    if $current$ is a solution for csp **then return** $current$
    $var \leftarrow$ a randomly chosen, conflicted variable from VARIABLES[csp]
    $value \leftarrow$ the value v for var that minimizes CONFLICTS(var, v, $current$, csp)
    set $var = value$ in $current$
  return $failure$

Příklad: 8-královen







Porovnání

- mapa USA: barvení mapy USA pomocí 4 barev
- n-královen: řešení 2 až 50-královen
- zebra: logická hádanka (viz následující slide)
- random1, random 2: umělé problémy

problém	BCK	BCK+MRV	FCH	FCH+MRV	MIN-CON
mapa USA	($>$ 1.000K)	($>$ 1.000K)	2K	60	64
n-královen	($>$ 40.000K)	13.500K	($>$ 40.000K)	817K	4K
zebra	3.859K	1K	35K	0.5K	2K
random1	4.15K	3K	26K	2K	
random2	942K	27K	77K	15K	

měříme medián počtu kontroly konzistence (přes 5 testů), výraz v závorce znamená, že po tolika operacích nebylo nalezeno řešení.