Umělá inteligence I

Roman Barták, KTIML

roman.bartak@mff.cuni.cz
http://ktiml.mff.cuni.cz/~bartak

6

Na úvod

- Zatím pro nás byl model světa **černou skříňkou**, ke které přistupujeme pouze přes:
  - funkci následníka
  - test cílového stavu
  - heuristickou funkci (vzdálenost do cíle)
- **Dnes** si ukážeme
  - jednu z možných obecných reprezentací problémů – **problém splňování podmínek (CSP)**
    - má vnitřní strukturu, kterou lze využít při řešení
  - obecné metody pro **řešení CSP**
    - kombinace prohledávání a odvozovacích technik

Sudoku?

- **Logická hádanka,** jejímž cílem je doplnit chybějící čísla 1-9 do tabulky 9×9 tak, aby se čísla neopakovala v žádném řádku, sloupci ani v malých čtvercích 3×3.

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

Trochu historie

1979: poprvé publikováno v New Yorku

„Number Place" – „Umísti číslice"

1986: populární v Japonsku

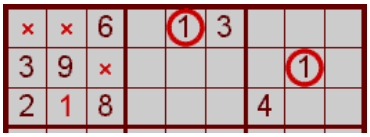
Sudoku – zkráceno z japonštiny "**Súdži** wa **doku**šin ni kagiru"

"Čísla musí být jedinečná"

2005: populární mezinárodně

Začínáme se Sudoku

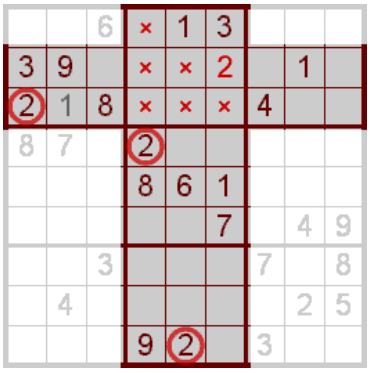
Jak poznáme, kam které číslo patří?



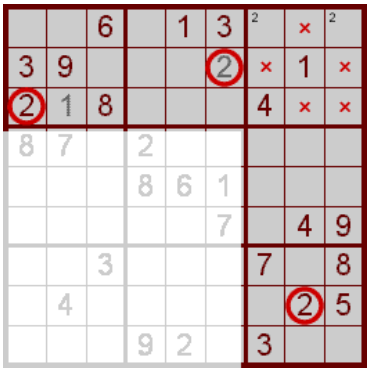
- Využijeme informace, že každé číslo je v řádku maximálně jedenkrát.

A co když to nestačí?

- Podíváme se do sloupců resp. zkombinujeme informaci ze sloupců a řádků.

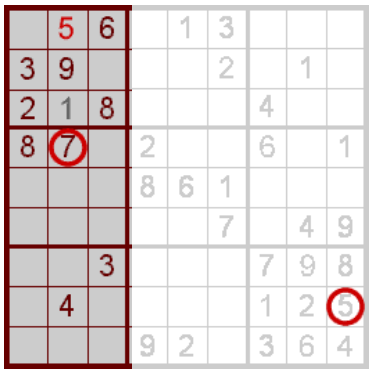


Sudoku o krok dál

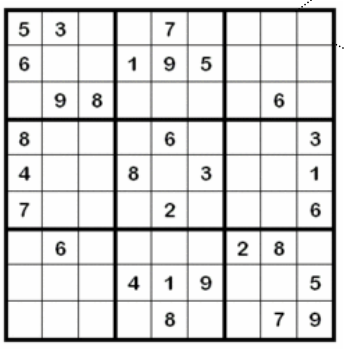


- Pokud řádky ani sloupce neposkytnout přesnou informaci, alespoň si můžeme poznamenat, kde dané číslo může ležet.

- Poloha čísla může být odvozena i z přítomnosti jiných čísel a z vlastnosti, že každé číslo se musí objevit alespoň jednou.



Sudoku obecně



Na každé políčko se podíváme jako na **proměnnou** nabývající hodnoty z **domény** {1,…,9}.

Mezi všemi dvojicemi proměnných v řádku, sloupci či malém čtverci je **podmínka** nerovnosti.

Hodnoty nesplňující podmínky **škrtáme**.

Takto zadaný problém se nazývá **problém splňování omezujících podmínek.**

Škrtání hodnot – **filtrace domén** – se provádí tak dlouho, dokud se nějaká doména mění.

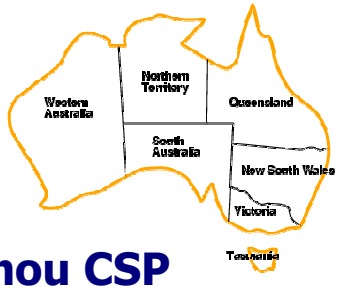
CSP

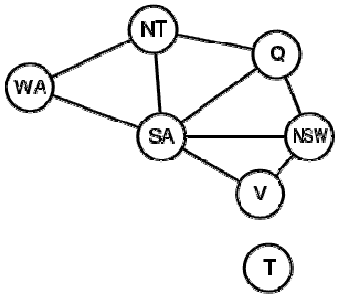
Problém splňování podmínek se skládá z:

- konečné množiny **proměnných**
  - popisují „atributy" řešení, o kterých je potřeba rozhodnout například pozici dámy na šachovnici
- **domén** – konečné množiny hodnot pro proměnné
  - popisují možnosti, které máme při rozhodování např. čísla řádků pro umístění dámy
  - někdy se definuje jedna superdoména společná pro všechny proměnné a individuální domény se v ní vytyčí přes unární podmínky
- konečné množiny **podmínek**
  - podmínka je libovolná relace nad množinou proměnných například poziceA ≠ poziceB
  - může být definována extenzionálně (jako množina kompatibilních n-tic) nebo intenzionálně (formulí)

Příklad CSP

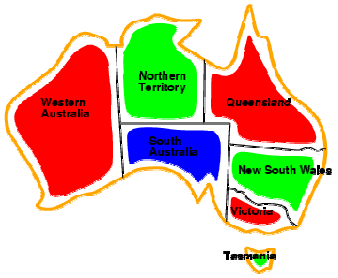
- Najděte obarvení států (červená, modrá, zelená) takové, že sousední státy nemají stejnou barvu.





- **Reprezentace formou CSP**
  - proměnné: {WA, NT, Q, NSW, V, SA, T}
  - superdoména: {č, m, z}
  - podmínky: WA ≠ NT atd.
- Lze také popsat formou **sítě podmínek** (vrcholy=proměnné, hrany=podmínky)

- **Řešení problému**
  WA = č, NT = z, Q = č, NSW = z, V = č, SA = m, T = z



Základní pojmy

Stav odpovídá přiřazení hodnot do (některých) proměnných.

Konzistentní stav (přiřazení) neporušuje žádnou z podmínek.

V **úplném stavu** (přiřazení) mají všechny proměnné přiřazenu hodnotu.

Cílem je najít úplný konzistentní stav (přiřazení).

Někdy se k definici problému přidává **objektivní funkce**, tj. funkce definovaná na (některých) proměnných. Potom je **optimálním cílovým stavem** úplný konzistentní stav (přiřazení), který maximalizuje/minimalizuje hodnotu objektivní funkce.

Jak řešit CSP?

- Zatím jsme se naučili **prohledávací algoritmy**, tak je zkusíme i pro CSP.
  - **počáteční stav**: prázdné přiřazení
  - **možné akce**: přiřazení hodnoty do volné proměnné takové, že neporuší žádnou podmínku
  - **cíl**: úplné konzistentní přiřazení

Poznámky:

- řešící postup je stejný pro všechny CSP
- řešení je vždy v hloubce n, kde n je počet proměnných
  - můžeme bez problémů použít DFS (bez kontroly cyklů)
- pořadí akcí nemá žádný vliv na řešení (problém je tzv. **komutativní**)
  - ⟨WA=č, NT=z⟩ je stejné jako ⟨NT=z, WA=č⟩
- pro řešení CSP lze používat i jiná typy větvení než enumeraci (například půlení domén)

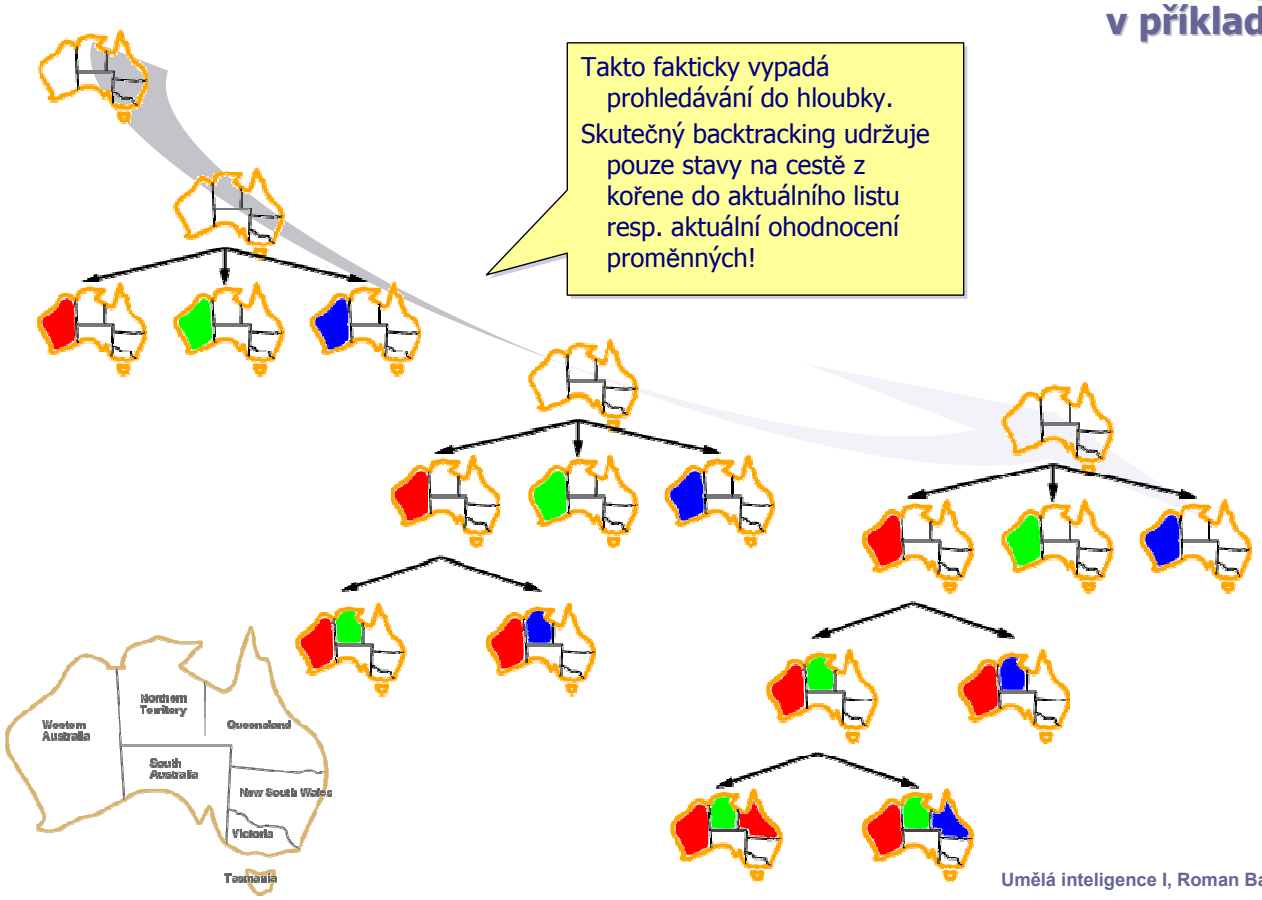
Backtracking
rekurzivní algoritmus

- Základní neinformovaný algoritmus pro řešení CSP
  - postupně přiřazujeme hodnoty do volných proměnných
  - pokud žádná hodnota přiřadit nelze, potom u poslední přiřazené proměnné zkusíme jinou hodnotu

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

function RECURSIVE-BACKTRACKING( assignment,csp) returns a solution, or
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 according to Constraints[csp] then
         add { var = value } to assignment
         result ← RECURSIVE-BACKTRACKING(assignment, csp)
         if result ≠ failue then return result
         remove { var = value } from assignment
   return failure
```

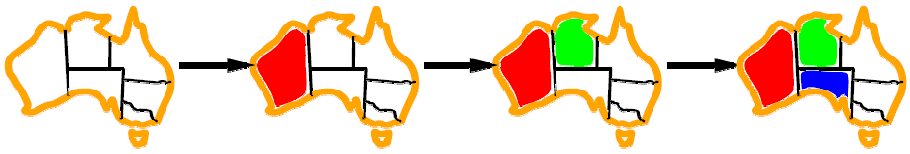

Backtracking
v příkladě



Jak ovlivnit efektivitu prohledávání?

- **volbou „správné" hodnoty**
  - zpravidla problémově závislé
- **volbou proměnné pro ohodnocení**
  - i když nakonec musíme přiřadit hodnoty do všech proměnných, může pořadí proměnných ovlivnit velikost prohledávaného prostoru
  - úspěšné problémově nezávislé heuristiky
- **včasnou detekcí slepých větví**
  - odvozením dodatečné informace
- využitím struktury problému
  - některé problémy (např. CSP se stromovým grafem podmínek) lze vyřešit bez navracení

- **Nejvíce omezená** proměnná
  - tj. proměnná s nejmenším počtem možných akcí
  - tj. proměnná s nejmenší doménou
  - tzv. **dom heuristika**

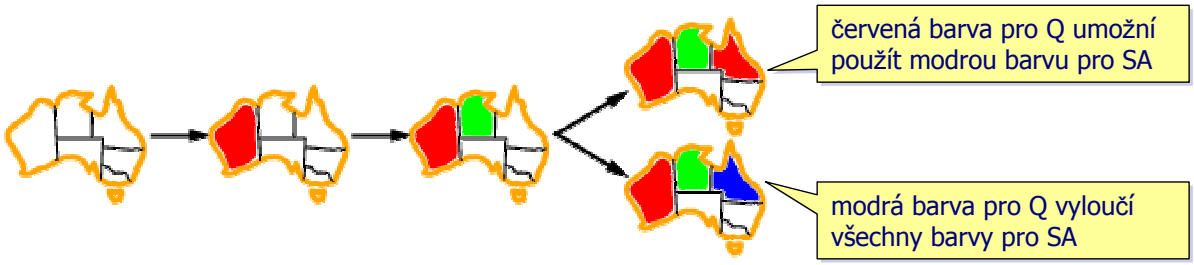


- Proměnná **s nejvíce podmínkami** s volnými proměnnými
  - tzv. **deg heuristika**
  - používá se pro dodatečný výběr pokud dom heuristika neurčí jedinou proměnnou (**dom+deg heuristika**)

V obou případech se jedná o instanci **pravidla prvního neúspěchu** (fail-first), které doporučuje vybrat proměnnou, jejíž ohodnocení nejspíše povede k neúspěchu.

Backtracking
výběr hodnoty

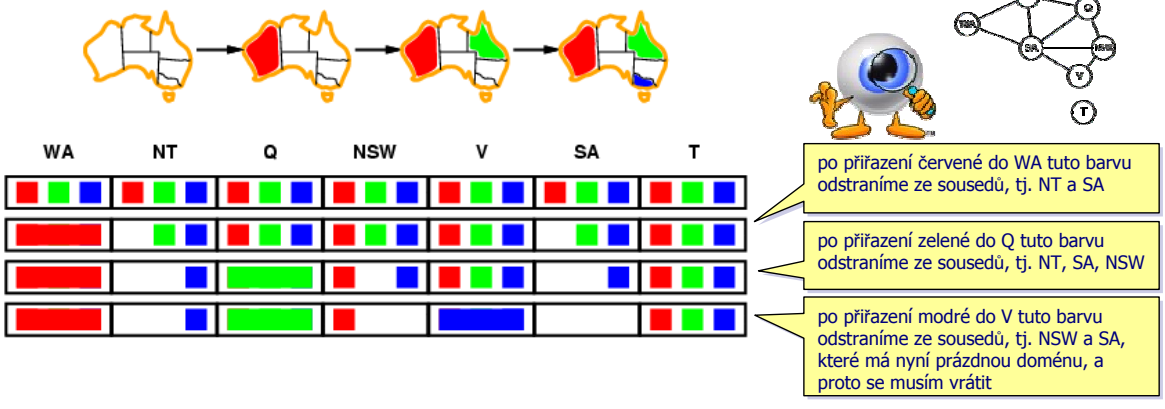
- Pří výběru hodnoty naopak preferujeme hodnotu, která nejspíše patří do řešení – **pravidlo prvního úspěchu** (succeed-first).
- Jak ji ale obecně poznáme?
  - například hodnota, která **nejméně omezí zbývající volné proměnné** (nechá v problému největší flexibilitu)



  - „přesnější" hodnotu lze najít například zjištěním počtu řešení relaxovaného problému, kde je daná hodnota použita
  - výpočtově náročné, proto se často používají problémově závislé heuristiky

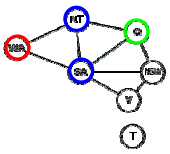
Kontrola dopředu

- **Nemohli bychom „uhodnout" dopředu, že daná cesta nevede do cíle?**
  - po přiřazení hodnoty můžeme dopředu zkontrolovat podmínky, které vedou z aktuální proměnné do volných proměnných – **kontrola dopředu**
  - zkontrolovat = vyřadit hodnoty, které podmínku nesplňují



Propagace podmínek

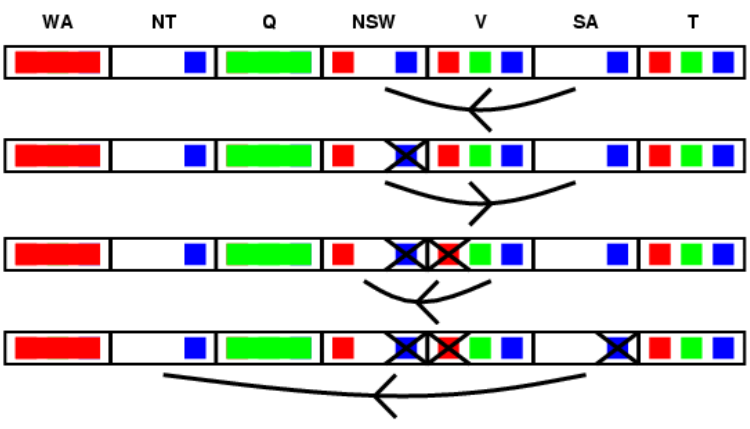
- **Nemohli bychom z podmínek odvodit ještě více informací?**



  - pokud bychom se při kontrole dopředu podívali i na podmínky mezi dosud neohodnocenými proměnnými, zjistíme, že sousedé NT a SA nemohou být najednou modří, což jsou jediné hodnoty v jejich doménách
  - protože se přiřazená hodnota propaguje přes všechny podmínky do dalších proměnných, hovoříme o **propagaci podmínek** nebo také o **pohledu dopředu** (look ahead)
  - realizuje se metodou udržování **konzistence podmínek**

Hranová konzistence

- každá podmínka odfiltrujte z domén „svých" proměnných hodnoty, které ji nesplňují
- často se realizuje směrově odstraněním hodnot z domény proměnné X, které nemají podporu v doméně proměnné Y z (binární) podmínky (X,Y) a naopak



  - filtraci je třeba zavolat kdykoliv se změní doména proměnné Y
  - filtrace se tak opakuje, dokud se zmenšují domény až do dosažení pevného bodu nebo vyprázdnění nějaké domény

Algoritmus AC-3

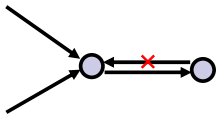
```
function AC-3( csp) returns the CSP, possibly with reduced domains
  inputs: csp, a binary CSP with variables {X1, X2, ..., Xn}
  local variables: queue, a queue of arcs, initially all the arcs in csp

  while queue is not empty do
    (Xi, Xj) ← REMOVE-FIRST(queue)
    if RM-INCONSISTENT-VALUES(Xi, Xj) then
      for each Xk in NEIGHBORS[Xi] do
        add (Xk, Xi) to queue

function RM-INCONSISTENT-VALUES( Xi, Xj) returns true iff remove a value
  removed ← false
  for each x in DOMAIN[Xi] do
    if no value y in DOMAIN[Xj] allows (x,y) to satisfy constraint(Xi, Xj)
      then delete x from DOMAIN[Xi]; removed ← true
  return removed
```

Algoritmus lze volat inkrementálně v rámci prohledávání - pokud se instanciovala proměnná X_i, potom se fronta inicializuje všemi hranami (podmínkami), které do ní vedou.

Pokud se změnila doména proměnné X_i, potom je potřeba zkontrolovat všechny hrany (podmínky), které do ní vedou s výjimkou X_j.

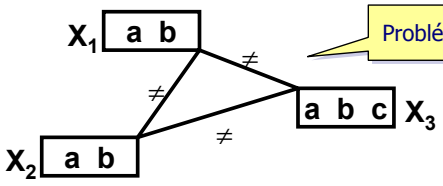


Filtrace domény proměnné X_i odstraní hodnoty, které nemají kompatibilní hodnotu v proměnné X_j a zároveň oznámí, zda se něco smazalo.
Pokud známe sémantiku podmínky, nemusíme kontrolovat hodnotu po hodnotě (např. u X<Y)

Časová složitost algoritmu $O(ed^3)$, kde e je počet podmínek a d velikost domény, není optimální, protože se opakovaně testují stejné dvojice hodnot. Pamatováním si výsledků testů lze udělat optimálně v $O(ed^2)$, AC-4, AC-3.1, AC-2001

Silnější konzistence

- Obecně lze definovat **k-konzistenci**, jako zajištění, že pro konzistentní hodnoty (k-1) různých proměnných lze najít konzistentní hodnotu libovolné další proměnné
  - hranová konzistence (AC) = 2-konzistence
  - konzistence po cestě (PC) = 3-konzistence



Problém je AC ale není PC.

- Je-li problém i-konzistentní ∀i=1,..,n (n je počet proměnných), potom umíme řešit bez navracení.
  - v DFS vždy najdeme hodnotu další proměnné, která je kompatibilní s i předchůdci
- Bohužel, složitost k-konzistence je exponenciální v k.

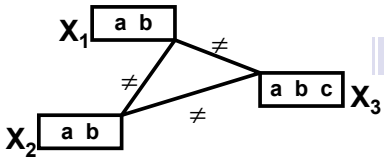
Globální podmínky

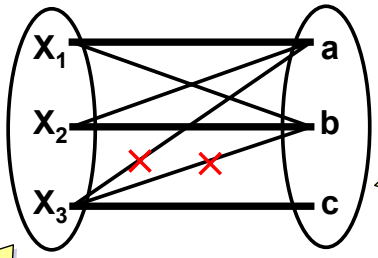
- Místo silnějších typů konzistencí se v praxi spíše používají tzv. **globální podmínky**, které v sobě sdružují několik jednoduchých podmínek a speciálním filtračním algoritmem zajišťují pro tuto množinu podmínek globální konzistenci.

Příklad:

podmínka **all_different({X_1,…, X_k})**

- nahrazuje binární nerovnosti $X_1 \neq X_2$, $X_1 \neq X_3$, …, $X_{k-1} \neq X_k$
- **all_different({X_1,…, X_k})** = {(d_1,…, d_k) | $\forall i \ d_i \in D_i$ & $\forall i \neq j \ d_i \neq d_j$}
- filtrační algoritmus založen na párování v bipartitních grafech





1. najde **maximální párování**
2. **odstraní hrany**, které nejsou součástí žádného maximálního párování
3. **odstraní** příslušné **hodnoty**

Bipartitní graf
- na jedné straně proměnné, na druhé straně hodnoty
- Hrana spojuje hodnotu s proměnnou, pokud je v její doméně

Finální poznámky

- **Deklarativní přístup** k řešení problémů
  - vytvoříme **model** (proměnné, domény, podmínky)
  - použijeme **obecný řešič** podmínek
- Možná rozšíření
  - **optimalizační problémy**
    - řešeno metodou větví a mezí (branch-and-bound)
  - **měkké podmínky**
    - podmínka určuje pouze preferenci, kterou je dobré respektovat
    - Řeší se podobně jako optimalizační problémy
- Jiné řešící techniky
  - **lokální prohledávání** (na cestě nezáleží)
  - celočíselné programování

Oblasti aplikací

Bioinformatika

- sekvencování DNA
- hledání 3D struktury proteinů

Plánování

- autonomní plánování operací kosmických lodí (Deep Space 1)

Rozvrhování výroby

- úspory po aplikaci CSP: denně US$ 0.2-1 milión

Pro zájemce

- **Systémy pro řešení podmínek**
  - SICStus Prolog (v našich laboratořích)
  - ECLiPSe (Open Source, http://eclipse.crosscoreop.com/)
  - GECODE (Open Source C++, http://www.gecode.org/)
  - ...
- Přednáška **Programování s omezujícími podmínkami**
  - **LS 2/0 Zk**
  - http://kti.mff.cuni.cz/~bartak/podminky/