

Dekompozice problému, AND/OR grafy, problémy s omezujícími podmínkami

Aleš Horák

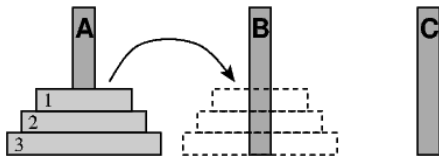
E-mail: hales@fi.muni.cz
<http://nlp.fi.muni.cz/uui/>

Obsah:

- Dekompozice a AND/OR grafy
- Prohledávání AND/OR grafů
- Problémy s omezujícími podmínkami
- CLP – Constraint Logic Programming

Příklad – Hanojské věže

- máme tři tyče: **A**, **B** a **C**.
- na tyči **A** je (podle velikosti) n kotoučů.
- úkol: přeskládat z **A** pomocí **C** na tyč **B** (zaps. $n(\mathbf{A}, \mathbf{B}, \mathbf{C})$) bez porušení uspořádání

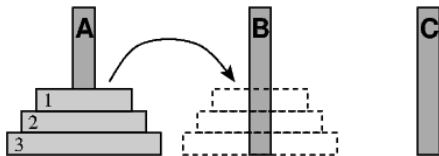


bit.ly/uuihanoi1 (Tower3 je zde cíl)

Slido

Příklad – Hanojské věže

- máme tři tyče: **A**, **B** a **C**.
- na tyči **A** je (podle velikosti) n kotoučů.
- úkol: přeskládat z **A** pomocí **C** na tyč **B** (zaps. $n(\mathbf{A}, \mathbf{B}, \mathbf{C})$) bez porušení uspořádání

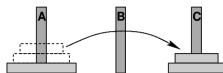


bit.ly/uuihanoi1 (Tower3 je zde cíl)

Slido

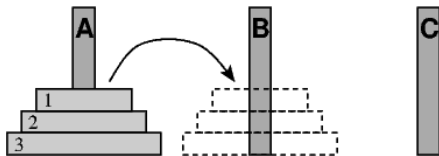
Můžeme rozložit na fáze:

1. přeskládat $n - 1$ kotoučů z **A** pomocí **B** na **C**.



Příklad – Hanojské věže

- máme tři tyče: **A**, **B** a **C**.
- na tyči **A** je (podle velikosti) n kotoučů.
- úkol: přeskládat z **A** pomocí **C** na tyč **B** (zaps. $n(\mathbf{A}, \mathbf{B}, \mathbf{C})$) bez porušení uspořádání

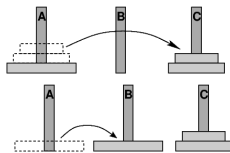


bit.ly/uuihanoi1 (Tower3 je zde cíl)

Slido

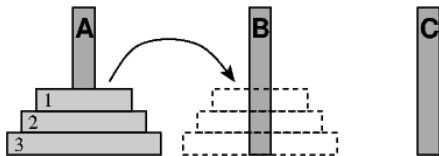
Můžeme rozložit na fáze:

1. přeskládat $n - 1$ kotoučů z **A** pomocí **B** na **C**.
2. přeložit **1** kotouč z **A** na **B**



Příklad – Hanojské věže

- máme tři tyče: **A**, **B** a **C**.
- na tyči **A** je (podle velikosti) n kotoučů.
- úkol: přeskládat z **A** pomocí **C** na tyč **B** (zaps. $n(\mathbf{A}, \mathbf{B}, \mathbf{C})$) bez porušení uspořádání

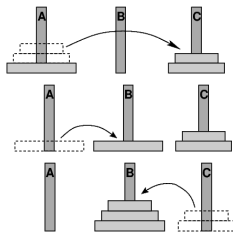


bit.ly/uuihanoi1 (Tower3 je zde cíl)

SliDo

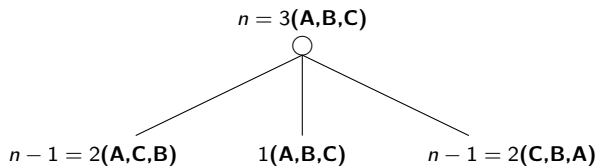
Můžeme rozložit na fáze:

1. přeskládat $n - 1$ kotoučů z **A** pomocí **B** na **C**.
2. přeložit **1** kotouč z **A** na **B**
3. přeskládat $n - 1$ kotoučů z **C** pomocí **A** na **B**



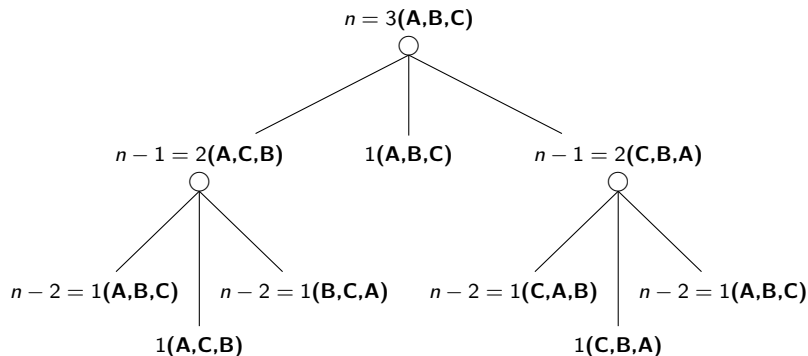
Příklad – Hanojské věže – pokrač.

schéma celého řešení pro $n = 3$:



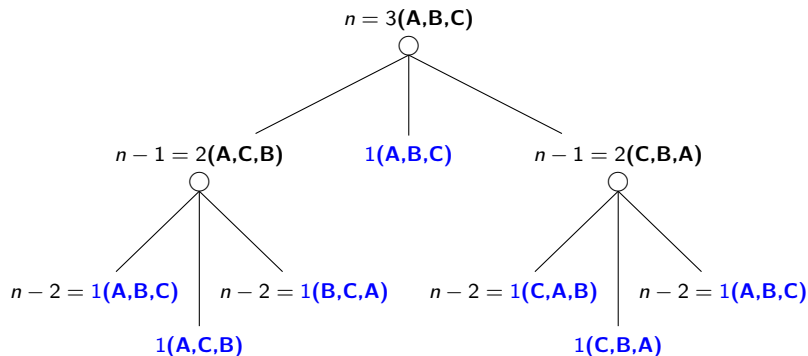
Příklad – Hanojské věže – pokrač.

schéma celého řešení pro $n = 3$:



Příklad – Hanojské věže – pokrač.

schéma celého řešení pro $n = 3$:



Příklad – Hanojské věže – pokrač.

```
function HANOITOWER(n, source, dest, spare)  
  if n = 1 then  
    return (source, dest)    # pro 1 disk – přesuň ho ze source na dest  
  else  
    output = HanoiTower(n - 1, source, spare, dest) # přesuň n - 1 disků na spare  
    output.append((source, dest))    # přesuň zbývající největší disk na dest  
    # přesuň odložených n - 1 disků na dest  
    output.append(HanoiTower(n - 1, spare, dest, source))  
  return output
```

Příklad – Hanojské věže – pokrač.

```
function HANOITOWER(n, source, dest, spare)  
  if n = 1 then  
    return (source, dest)  # pro 1 disk – přesuň ho ze source na dest  
  else  
    output = HanoiTower(n - 1, source, spare, dest) # přesuň n - 1 disků na spare  
    output.append((source, dest)) # přesuň zbývající největší disk na dest  
    # přesuň odložených n - 1 disků na dest  
    output.append(HanoiTower(n - 1, spare, dest, source))  
  return output
```

```
HanoiTower(3, 'a', 'b', 'c'):  
[('a', 'b'), ('a', 'c'), ('b', 'c'), ('a', 'b'), ('c', 'a'),  
('c', 'b'), ('a', 'b')]
```

optimalizace?

Příklad – Hanojské věže – pokrač.

```

function HANOITOWER(n, source, dest, spare)
  if n = 1 then
    return (source, dest)  # pro 1 disk – přesuň ho ze source na dest
  else
    output = HanoiTower(n - 1, source, spare, dest) # přesuň n - 1 disků na spare
    output.append((source, dest)) # přesuň zbývající největší disk na dest
    # přesuň odložených n - 1 disků na dest
    output.append(HanoiTower(n - 1, spare, dest, source))
  return output

```

```

HanoiTower(3, 'a', 'b', 'c'):
  [('a', 'b'), ('a', 'c'), ('b', 'c'), ('a', 'b'), ('c', 'a'),
   ('c', 'b'), ('a', 'b')]

```

optimalizace – ukládat výsledky prvního rekurzivního volání a uložený výsledek vždy převzít **bez nadbytečné další rekurze**

Cesta mezi městy pomocí dekompozice

města:

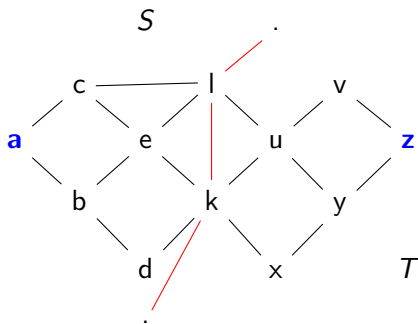
a, ..., **e** ... ve státě *S*

l a **k** ... hraniční přechody

u, ..., **z** ... ve státě *T*

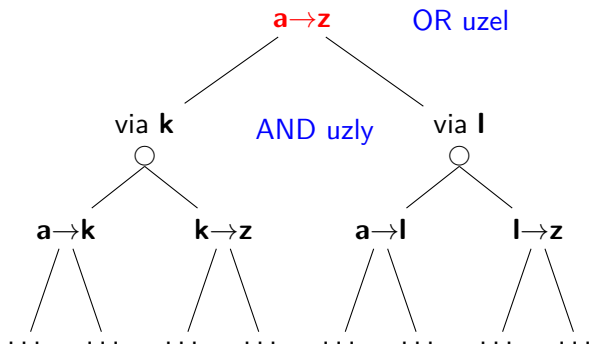
hledáme cestu z **a** do **z**:

- cesta z **a** do hraničního přechodu
- cesta z hraničního přechodu do **z**



Cesta mezi městy pomocí dekompozice – pokrač.

schéma řešení pomocí rozkladu na podproblémy = AND/OR graf



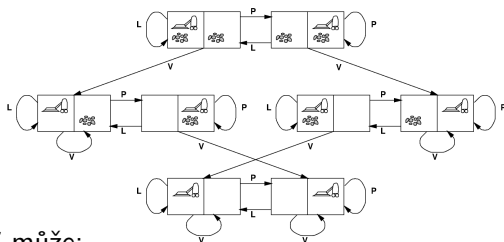
Celkové řešení = prohledávací strom AND/OR grafu, který nevynechává žádného následníka AND-uzlu.

Příklad – agent vysavač v nestálém prostředí

problém agenta Vysavače

v **nestálém** prostředí:

- dvě **místnosti**, jeden **vysavač**
- v každé místnosti je/není špína
- počet **stavů** je $2 \times 2^2 = 8$
- **akce** = {doLeva, doPrava, Vysávej}
- **nedeterminismus** – akce Vysávej může:
 - ve **špinavé** místnosti – **vysát** místnost a **někdy** i tu **vedlejší**
 - v **čisté** místnosti – **někdy** místnost **zašpinit**

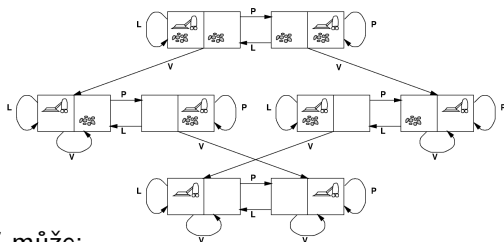


Příklad – agent vysavač v nestálém prostředí

problém agenta Vysavače

v **nestálém** prostředí:

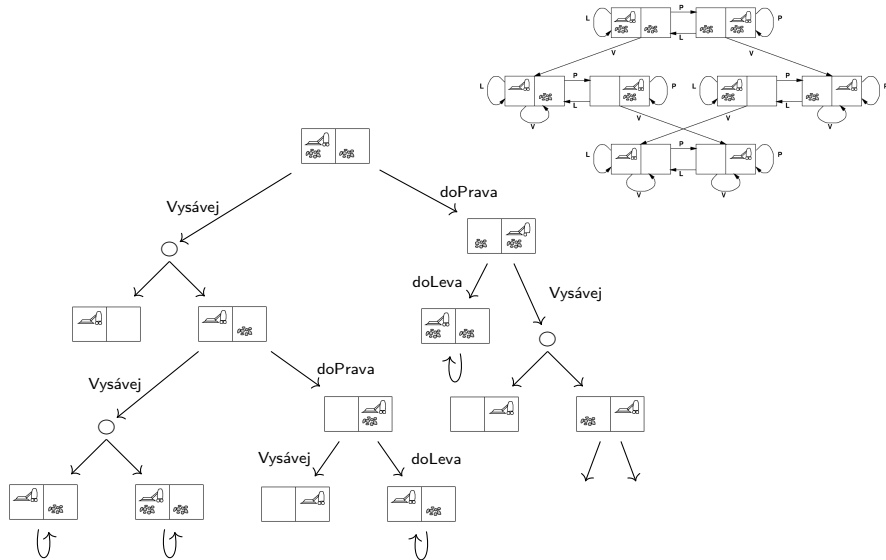
- dvě **místnosti**, jeden **vysavač**
- v každé místnosti je/není špína
- počet **stavů** je $2 \times 2^2 = 8$
- **akce** = {doLeva, doPrava, Vysávej}
- **nedeterminismus** – akce Vysávej může:
 - ve **špinavé** místnosti – **vysát** místnost a **někdy** i tu **vedlejší**
 - v **čisté** místnosti – **někdy** místnost **zašpinit**



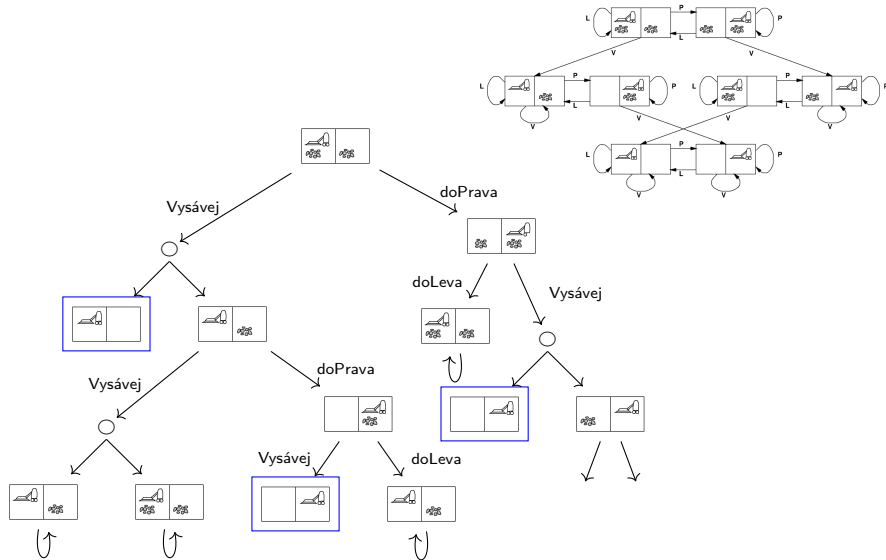
Strategie/kontingenční plán (prohledávací strom) obsahuje 2 typy uzlů:

- deterministické stavy, kde se **prostředí nemůže měnit** – agent jen volí další postup, **OR**
- nedeterministické stavy, kde se **prostředí náhodně může změnit** – agent musí řešit více možností, **AND**
- mohou nastat **cykly**, řešení je jen když nedeterminismus není **vždy negativní**

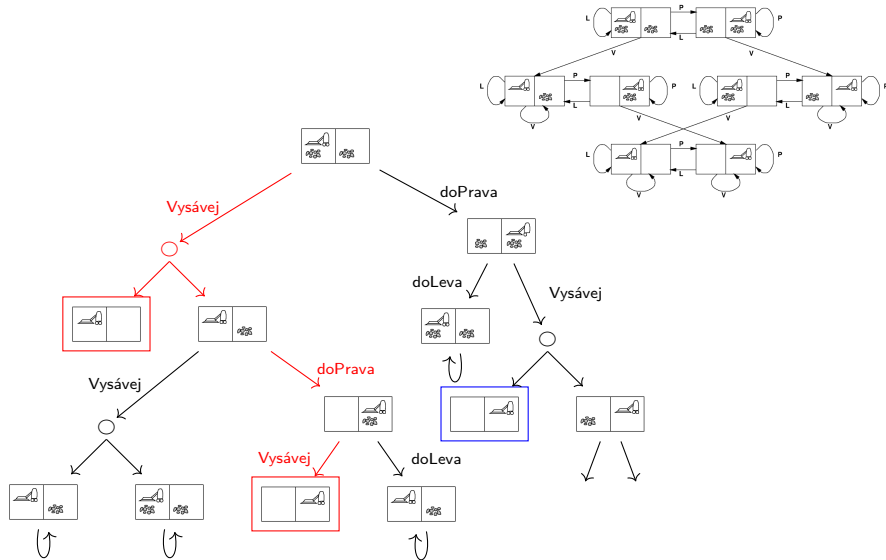
Příklad – agent vysavač v nestálém prostředí pokrač.



Příklad – agent vysavač v nestálém prostředí pokrač.



Příklad – agent vysavač v nestálém prostředí pokrač.

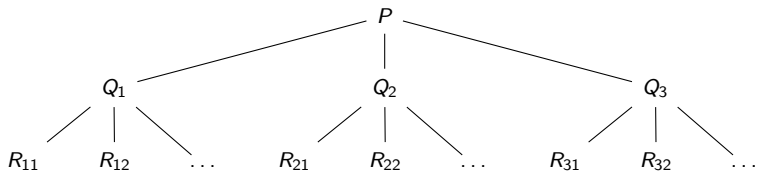


Příklad – výherní strategie

Hra 2 hráčů s perfektními znalostmi, 2 výstupy $\left\{ \begin{array}{l} \text{výhra} \\ \text{prohra} \end{array} \right.$

Výherní strategii je možné formulovat jako AND/OR graf:

- počáteční stav P typu *já-jsem-na-tahu*
- moje tahy vedou do stavů Q_1, Q_2, \dots typu *soupeř-je-na-tahu*
- následně soupeřovy tahy vedou do stavů R_{11}, R_{12}, \dots *já-jsem-na-tahu*

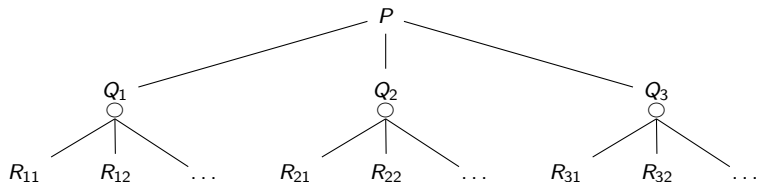


Příklad – výherní strategie

Hra 2 hráčů s perfektními znalostmi, 2 výstupy $\left\{ \begin{array}{l} \text{výhra} \\ \text{prohra} \end{array} \right.$

Výherní strategii je možné formulovat jako AND/OR graf:

- počáteční stav P typu *já-jsem-na-tahu*
- moje tahy vedou do stavů Q_1, Q_2, \dots typu *soupeř-je-na-tahu*
- následně soupeřovy tahy vedou do stavů R_{11}, R_{12}, \dots *já-jsem-na-tahu*
- cíl – stav, který je **výhra** podle pravidel (*prohra* je neřešitelný problém)
- stav P *já-jsem-na-tahu* je **výherní** \Leftrightarrow **některý** z Q_i je výherní, **OR**
- stav Q_i *soupeř-je-na-tahu* je **výherní** \Leftrightarrow **všechny** R_{ij} jsou výherní, **AND**
- **výherní strategie** = řešení AND/OR grafu

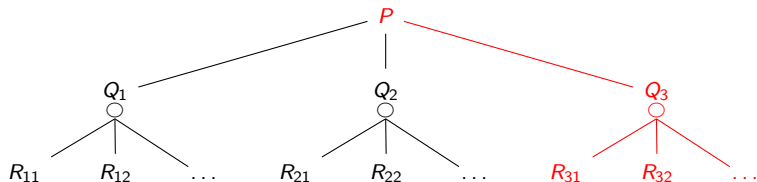


Příklad – výherní strategie

Hra 2 hráčů s perfektními znalostmi, 2 výstupy $\left\{ \begin{array}{l} \text{výhra} \\ \text{prohra} \end{array} \right.$

Výherní strategii je možné formulovat jako AND/OR graf:

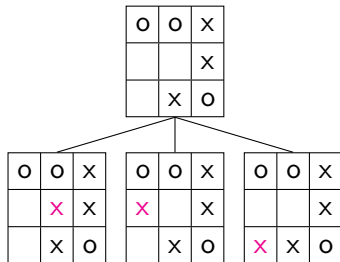
- počáteční stav P typu *já-jsem-na-tahu*
- moje tahy vedou do stavů Q_1, Q_2, \dots typu *soupeř-je-na-tahu*
- následně soupeřovy tahy vedou do stavů R_{11}, R_{12}, \dots *já-jsem-na-tahu*
- cíl – stav, který je **výhra** podle pravidel (*prohra* je neřešitelný problém)
- stav P *já-jsem-na-tahu* je **výherní** \Leftrightarrow **některý** z Q_i je výherní, **OR**
- stav Q_i *soupeř-je-na-tahu* je **výherní** \Leftrightarrow **všechny** R_{ij} jsou výherní, **AND**
- **výherní strategie** = řešení AND/OR grafu



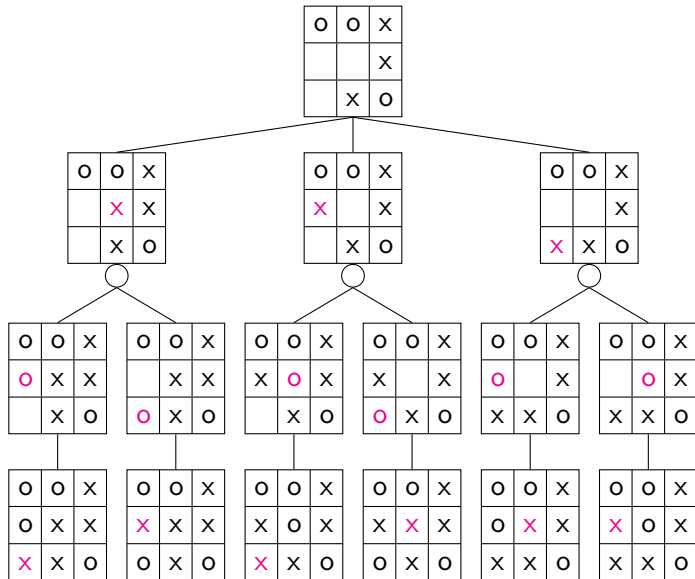
Příklad – výherní strategie

O	O	X
		X
	X	O

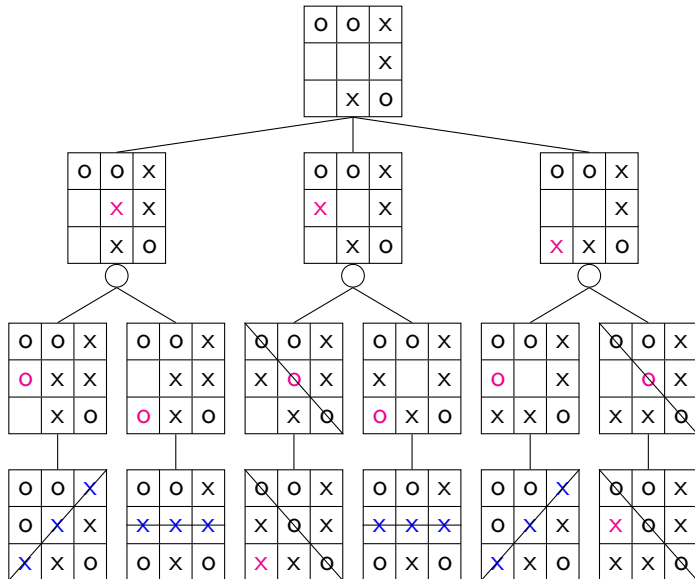
Příklad – výherní strategie



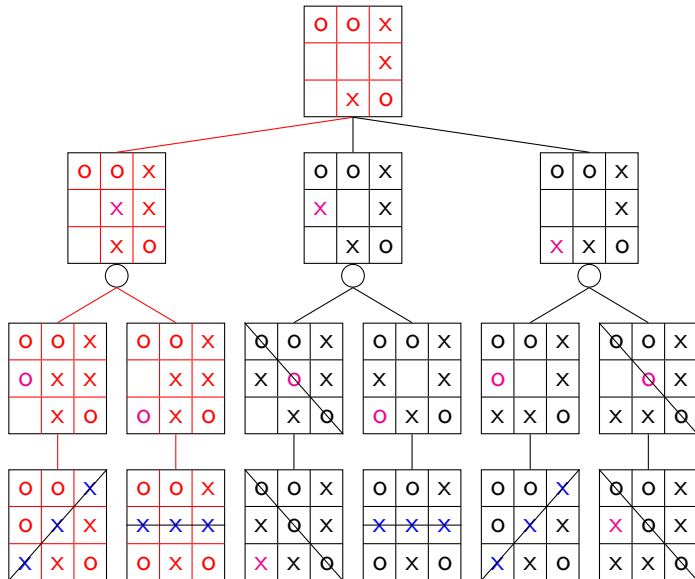
Příklad – výherní strategie



Příklad – výherní strategie



Příklad – výherní strategie



AND/OR graf a strom řešení

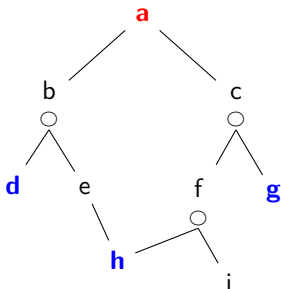
AND/OR graf = graf s 2 typy vnitřních uzlů – **AND uzly** a **OR uzly**

- *AND uzel* jako součást řešení vyžaduje průchod všech svých poduzlů
- *OR uzel* se chová jako běžný uzel klasického grafu

AND/OR graf a strom řešení

AND/OR graf = graf s 2 typy vnitřních uzlů – **AND uzly** a **OR uzly**

- **AND uzel** jako součást řešení vyžaduje průchod všech svých poduzlů
- **OR uzel** se chová jako běžný uzel klasického grafu



AND/OR graf a strom řešení

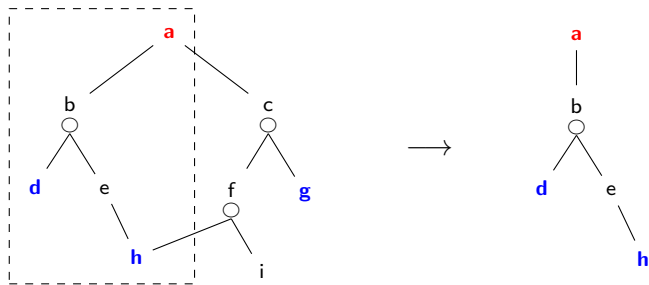
strom řešení T problému P s AND/OR grafem G :

- problém P je **kořen** stromu T
- jestliže P je **OR uzel** grafu $G \Rightarrow$ právě jeden z jeho následníků se svým stromem řešení je v T
- jestliže P je **AND uzel** grafu $G \Rightarrow$ všichni jeho následníci se svými stromy řešení jsou v T
- každý list stromu řešení T je **cílovým uzlem** v G

AND/OR graf a strom řešení

strom řešení T problému P s AND/OR grafem G :

- problém P je **kořen** stromu T
- jestliže P je **OR uzel** grafu $G \Rightarrow$ právě jeden z jeho následníků se svým stromem řešení je v T
- jestliže P je **AND uzel** grafu $G \Rightarrow$ všichni jeho následníci se svými stromy řešení jsou v T
- každý list stromu řešení T je **cílovým uzlem** v G

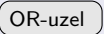
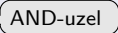


Obsah

- 1 Dekompozice a AND/OR grafy
 - Příklad – Hanojské věže
 - Další příklady
 - AND/OR graf a strom řešení
- 2 Prohledávání AND/OR grafů
 - Prohledávání AND/OR grafu do hloubky
 - Heuristické prohledávání AND/OR grafu (AO*)
- 3 Problémy s omezujícími podmínkami
 - Varianty CSP podle hodnot proměnných
 - Varianty omezení
- 4 CLP – Constraint Logic Programming
 - Řešení problémů s omezujícími podmínkami
 - Prohledávání s navracením
 - Ovlivnění efektivity prohledávání s navracením

Prohledávání AND/OR grafu do hloubky

```

function ANDORDEPTHFIRSTSEARCH(problem, path ← []) # vrací řešení nebo "failure"
  if length(path) = 0 then
    return AndOrDepthFirstSearch(problem, [problem.init_state])
  current_node ← path.last() # poslední prvek cesty
  if problem.is_goal(current_node) then
    return [] # prázdné (elementární) řešení
  else if problem.is_or_state(current_node) then
    
    foreach child in problem.moves(current_node) do
      if child ∈ path then next
      result = AndOrDepthFirstSearch(problem, path + [child])
      if result ≠ "failure" then return [child] + result
    return "failure"
  else if problem.is_and_state(current_node) then
    
    results = []
    foreach child in problem.moves(current_node) do
      if child ∈ path then return "failure"
      result = AndOrDepthFirstSearch(problem, path + [child])
      if result = "failure" then return "failure"
      results.append(result)
    return [child] + [results]

```


Prohledávání AND/OR grafu do hloubky

```

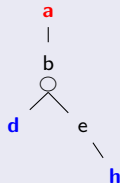
function ANDORDEPTHFIRSTSEARCH(problem, path ← []) # vrací řešení nebo "failure"
  if length(path) = 0 then
    return AndOrDepthFirstSearch(problem, [problem.init_state])
  current_node ← path.last() # poslední prvek cesty
  if problem.is_goal(current_node) then
    return [] # prázdné (elementární) řešení
  else if problem.is_or_state(current_node) then
    # OR-uzel
    foreach child in problem.moves(current_node) do
      if child ∈ path then next
      result = AndOrDepthFirstSearch(problem, path + [child])
      if result ≠ "failure" then return [child] + result
    return "failure"
  else if problem.is_and_state(current_node) then
    # AND-uzel
    results = []
    foreach child in problem.moves(current_node) do
      if child ∈ path then return "failure"
      result = AndOrDepthFirstSearch(problem, path + [child])
      if result = "failure" then return "failure"
      results.append(result)
    return [child] + [results]

```

```

AndOrDepthFirstSearch(problem):
  ['a', 'b', [['d'], ['e', 'h']]]

```



Heuristické prohledávání AND/OR grafu (AO*)

- algoritmus **AO*** má stejné charakteristiky a složitost jako **A***
- **cena přechodové hrany** = míra složitosti podproblému:
uzel = $\begin{matrix} \text{and} \\ \text{or} \end{matrix}, [(NasUzel1, Cena1), (NasUzel2, Cena2), \dots, (NasUzelN, CenaN)]$
- definujeme **cenu uzlu** jako cenu optimálního řešení jeho podstromu
- pro každý uzel N máme daný **odhad** jeho **ceny**:

$h(N)$ = heuristický odhad ceny optimálního podgrafu s kořenem N

- pro každý uzel N , jeho následníky N_1, \dots, N_b a jeho předchůdce M definujeme:

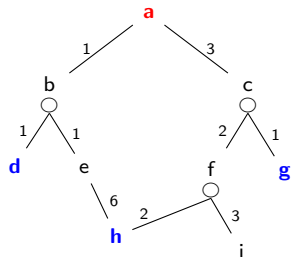
$$F(N) = \text{cena}(M, N) + \begin{cases} h(N), & \text{pro ještě neexpandovaný uzel } N \\ 0, & \text{pro cílový uzel (elementární problém)} \\ \min_i (F(N_i)), & \text{pro OR-uzel } N \\ \sum_i F(N_i), & \text{pro AND-uzel } N \end{cases}$$

Pro **optimální strom řešení** S je tedy $F(S)$ právě **cena** tohoto **řešení**

Heuristické prohledávání AND/OR grafu – příklad

setříděný seznam částečně expandovaných stromů řešení =
 [Nevyřešený₁, Nevyřešený₂, ..., Vyřešený₁, ...]

$$F_{\text{Nevyřešený}_1} \leq F_{\text{Nevyřešený}_2} \leq \dots$$

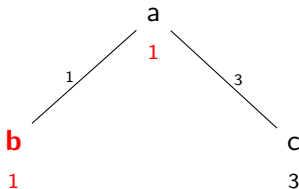
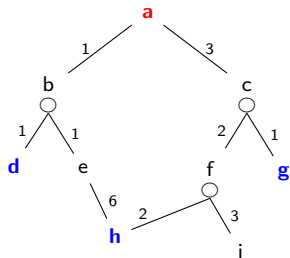


předp. $\forall N : h(N) = 0$

Heuristické prohledávání AND/OR grafu – příklad

setříděný seznam částečně expandovaných stromů řešení =
 [Nevyřešený₁, Nevyřešený₂, ..., Vyřešený₁, ...]

$$F_{\text{Nevyřešený}_1} \leq F_{\text{Nevyřešený}_2} \leq \dots$$

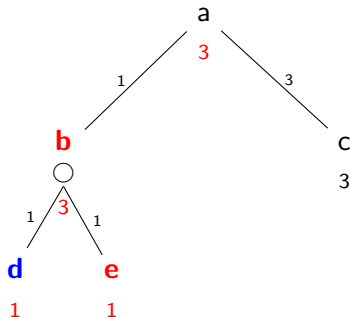
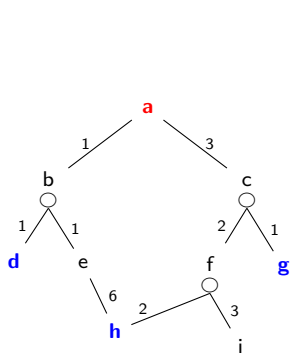


předp. $\forall N : h(N) = 0$

Heuristické prohledávání AND/OR grafu – příklad

setříděný seznam částečně expandovaných stromů řešení =
 [Nevyřešený₁, Nevyřešený₂, ..., Vyřešený₁, ...]

$$F_{\text{Nevyřešený}_1} \leq F_{\text{Nevyřešený}_2} \leq \dots$$

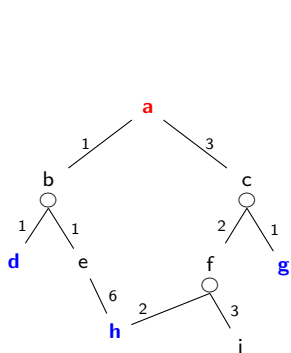


předp. $\forall N : h(N) = 0$

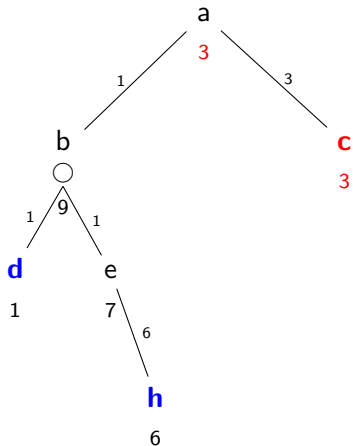
Heuristické prohledávání AND/OR grafu – příklad

setříděný seznam částečně expandovaných stromů řešení =
 [Nevyřešený₁, Nevyřešený₂, ..., Vyřešený₁, ...]

$$F_{\text{Nevyřešený}_1} \leq F_{\text{Nevyřešený}_2} \leq \dots$$



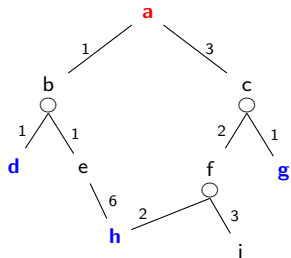
předp. $\forall N : h(N) = 0$



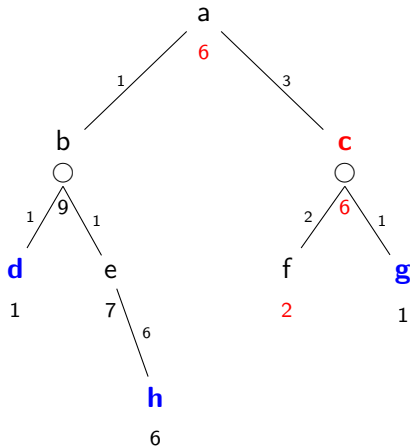
Heuristické prohledávání AND/OR grafu – příklad

setříděný seznam částečně expandovaných stromů řešení =
 [Nevyřešený₁, Nevyřešený₂, ..., Vyřešený₁, ...]

$$F_{\text{Nevyřešený}_1} \leq F_{\text{Nevyřešený}_2} \leq \dots$$



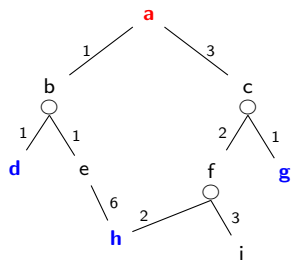
předp. $\forall N : h(N) = 0$



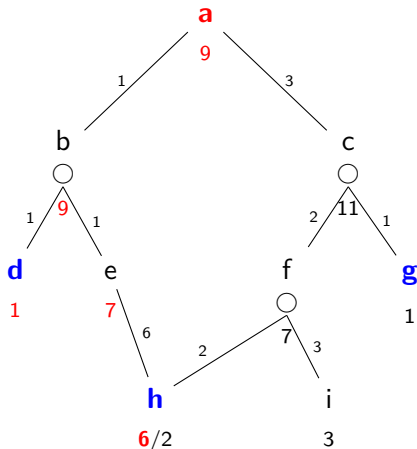
Heuristické prohledávání AND/OR grafu – příklad

setříděný seznam částečně expandovaných stromů řešení =
 [Nevyřešený₁, Nevyřešený₂, ..., Vyřešený₁, ...]

$$F_{\text{Nevyřešený}_1} \leq F_{\text{Nevyřešený}_2} \leq \dots$$



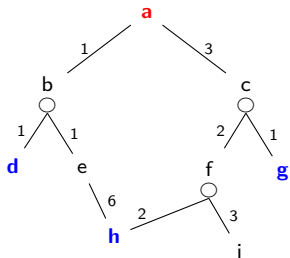
předp. $\forall N : h(N) = 0$



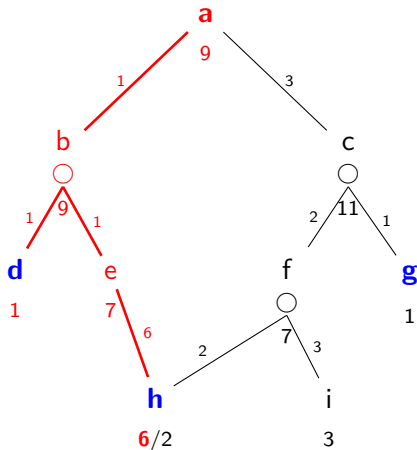
Heuristické prohledávání AND/OR grafu – příklad

setříděný seznam částečně expandovaných stromů řešení =
 [Nevyřešený₁, Nevyřešený₂, ..., Vyřešený₁, ...]

$$F_{\text{Nevyřešený}_1} \leq F_{\text{Nevyřešený}_2} \leq \dots$$

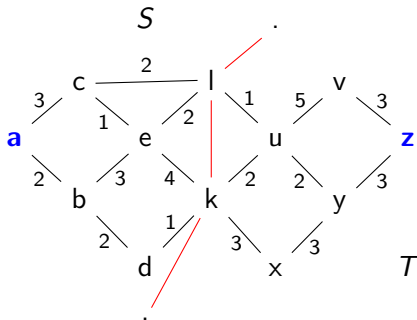


předp. $\forall N : h(N) = 0$



Cesta mezi městy heuristickým AND/OR hledáním

- cesta mezi **sousedícími městy Mesto1** a **Mesto2** – ohodnocené hrany **problem.moves(Mesto1) → [(Mesto2, Vzda2), ...]**
- klíčové postavení** města **Mesto3** na cestě z **Mesto1** do **Mesto2** – funkce **problem.key(Mesto1, Mesto2) → [Mesto3, ...]**.



Cesta mezi městy heuristickým AND/OR hledáním

vlastní hledání cesty:

1. $\exists Y_1, Y_2, \dots$ **klíčové body** mezi městy **A** a **Z**.

Hledej **jednu z cest**:

- cestu z **A** do **Z** přes **Y1**
- cestu z **A** do **Z** přes **Y2**
- ...

2. **Není-li** mezi městy **A** a **Z** **klíčové město** \Rightarrow hledej **sousedu** **Y** města **A** takového, že existuje cesta z **Y** do **Z**.

Cesta mezi městy heuristickým AND/OR hledáním

Konstrukce příslušného AND/OR grafu:

“manuální” výpis všech uzlů:

```
# kterákoliv cesta přes klíčové město
```

```
'a-z' → ('or', [( 'a-z via k', 0), ('a-z via l', 0)])
```

```
'a-v' → ('or', [( 'a-v via k', 0), ('a-v via l', 0)])
```

```
...
```

```
# kterákoliv cesta přes sousední města
```

```
'a-l' → ('or', [( 'c-l', 3), ('b-l', 2)])
```

```
'b-l' → ('or', [( 'e-l', 3), ('d-l', 2)])
```

```
...
```

```
# cesta do klíčového města a z klíčového města
```

```
'a-z via l' → ('and', [( 'a-l', 0), ('l-z', 0)])
```

```
'a-v via l' → ('and', [( 'a-l', 0), ('l-v', 0)])
```

```
...
```

```
# cíle – elementární problémy
```

```
goal('a-a'); goal('b-b'); ...
```

Cesta mezi městy heuristickým AND/OR hledáním

Konstrukce příslušného AND/OR grafu:

“pravidlová” definice grafu:

```

# kterákoliv cesta přes klíčové město 'a-z' → ('or', [( 'a-z via k', 0), ('a-z via l', 0)]) ...
for X ∈ problem.cities do for Z ∈ problem.cities do
  nodes = []; for Y ∈ problem.key(X, Z) do nodes.append(('X-Z via Y', 0))
  if length(nodes) > 0 then problem.add('X-Z' → ('or', nodes))

# kterákoliv cesta přes sousední města 'a-l' → ('or', [( 'c-l', 3), ('b-l', 2)]) ...
for X ∈ problem.cities do for Z ∈ problem.cities do
  nodes = []; for Y, V ∈ problem.moves(X) do nodes.append(('Y-Z', V))
  if length(nodes) > 0 then problem.add('X-Z' → ('or', nodes))

# cesta do a z klíčového města 'a-z via l' → ('and', [( 'a-l', 0), ('l-z', 0)]) ...
for X ∈ problem.cities do for Z ∈ problem.cities do
  for Y ∈ problem.key(X, Z) do
    problem.add('X-Z via Y' → ('and', [( 'X-Y', 0), ('Y-Z', 0)]))

# cíle – elementární problémy goal('a-a'); goal('b-b'); ...
for X ∈ problem.cities do
  problem.add(goal('X-X'))

```

Cesta mezi městy heuristickým AND/OR hledáním – pokrač.

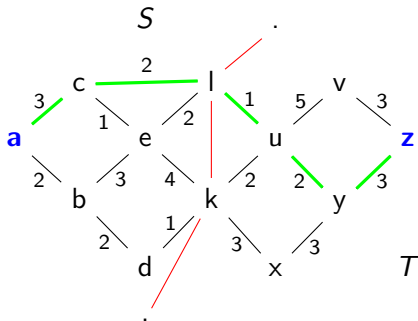
heuristika $h(X - Z \mid X - Z \text{ via } Y) = \text{vzdušná vzdálenost}$

Když $\forall n : h(n) \leq h^*(n)$, kde h^* je minimální cena řešení uzlu $n \Rightarrow$ najdeme **vždy optimální řešení**

Cesta mezi městy heuristickým AND/OR hledáním – pokrač.

heuristika $h(X - Z \mid X - Z \text{ via } Y) = \text{vzdušná vzdálenost}$

Když $\forall n : h(n) \leq h^*(n)$, kde h^* je minimální cena řešení uzlu $n \Rightarrow$ najdeme **vždy optimální řešení**



AO*Search('a-z'):

```

[('a-z',11),
 ('a-z via l',11),
  [(['l-z',6),
   ('u-z',6),
   ('y-z',5),
   ('z-z',3)],
  [(['a-l',5),
   ('c-l',5),
   ('l-l',2)]]]
  
```

AND-uzel

Obsah

- 1 Dekompozice a AND/OR grafy
 - Příklad – Hanojské věže
 - Další příklady
 - AND/OR graf a strom řešení
- 2 Prohledávání AND/OR grafů
 - Prohledávání AND/OR grafu do hloubky
 - Heuristické prohledávání AND/OR grafu (AO*)
- 3 Problémy s omezujícími podmínkami
 - Varianty CSP podle hodnot proměnných
 - Varianty omezení
- 4 CLP – Constraint Logic Programming
 - Řešení problémů s omezujícími podmínkami
 - Prohledávání s navrácením
 - Ovlivnění efektivity prohledávání s navrácením

Problémy s omezujícími podmínkami

- **standardní problém** řešený prohledáváním stavového prostoru → **stav** je “*černá skříňka*” – pouze **cílová podmínka** a **přechodová funkce**

Problémy s omezujícími podmínkami

- **standardní problém** řešený prohledáváním stavového prostoru → **stav** je “černá skříňka” – pouze **cílová podmínka** a **přechodová funkce**
- **problém s omezujícími podmínkami**, *Constraint Satisfaction Problem*, CSP:
 - n -tice **proměnných** X_1, X_2, \dots, X_n s hodnotami z **domén** D_1, D_2, \dots, D_n , $D_i \neq \emptyset$
 - množina **omezení** C_1, C_2, \dots, C_m nad proměnnými X_i

Problémy s omezujícími podmínkami

- **standardní problém** řešený prohledáváním stavového prostoru → **stav** je "černá skříňka" – pouze **cílová podmínka** a **přechodová funkce**
- **problém s omezujícími podmínkami**, *Constraint Satisfaction Problem*, CSP:
 - n -tice **proměnných** X_1, X_2, \dots, X_n s hodnotami z **domén** D_1, D_2, \dots, D_n , $D_i \neq \emptyset$
 - množina **omezení** C_1, C_2, \dots, C_m nad proměnnými X_i
 - **stav** = **přiřazení hodnot** proměnným $\{X_i = v_i, X_j = v_j, \dots\}$
 - konzistentní přiřazení neporušuje žádné z omezení C_i
 - úplné přiřazení zmiňuje každou proměnnou X_i
 - **řešení** = **úplné konzistentní přiřazení hodnot** proměnným někdy je ještě potřeba maximalizovat *cílovou funkci*

Problémy s omezujícími podmínkami

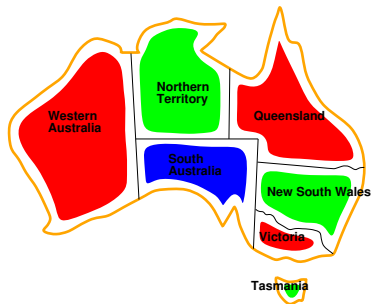
- **standardní problém** řešený prohledáváním stavového prostoru → **stav** je "černá skříňka" – pouze **cílová podmínka** a **přechodová funkce**
- **problém s omezujícími podmínkami**, *Constraint Satisfaction Problem*, CSP:
 - n -tice **proměnných** X_1, X_2, \dots, X_n s hodnotami z **domén** D_1, D_2, \dots, D_n , $D_i \neq \emptyset$
 - množina **omezení** C_1, C_2, \dots, C_m nad proměnnými X_i
 - **stav** = **přiřazení hodnot** proměnným $\{X_i = v_i, X_j = v_j, \dots\}$
 - konzistentní přiřazení neporušuje žádné z omezení C_i
 - úplné přiřazení zmiňuje každou proměnnou X_i
 - **řešení** = **úplné konzistentní přiřazení hodnot** proměnným někdy je ještě potřeba maximalizovat *cílovou funkci*
- **výhody**:
 - jednoduchý **formální jazyk** pro specifikaci problému
 - může využívat **obecné heuristiky** (ne jen specifické pro daný problém)

Příklad – obarvení mapy



- Proměnné WA, NT, Q, NSW, V, SA, T
- Domény $D_i = \{\text{červená}, \text{zelená}, \text{modrá}\}$
- Omezení – sousedící oblasti musí mít různou barvu
tj. pro každé dvě sousedící: $WA \neq NT$ nebo
 $(WA, NT) \in \{(\text{červená}, \text{zelená}), (\text{červená}, \text{modrá}), (\text{zelená}, \text{modrá}), \dots\}$

Příklad – obarvení mapy – pokrač.



- Řešení – konzistentní přiřazení všem proměnným:
{WA = červená, NT = zelená, Q = červená, NSW = zelená, V = červená,
SA = modrá, T = zelená}

Slido

Varianty CSP podle hodnot proměnných

- **diskrétní hodnoty proměnných** – spočetně mnoho jednotlivých hodnot
 - **konečné domény**
 - např. Booleovské (včetně NP-úplných problémů splnitelnosti)
 - výčtové
 - **nekonečné domény** – čísla, řetězce, ...
 - např. rozvrh prací – proměnné = počáteční/koncový den každého úkolu
 - vyžaduje **jazyk omezení**, např. $StartJob_1 + 5 \leq StartJob_3$
 - číselné *lineární* problémy jsou řešitelné, *nelineární* obecné řešení nemají
- **spojité hodnoty proměnných**

Varianty CSP podle hodnot proměnných

- **diskrétní hodnoty proměnných** – spočetně mnoho jednotlivých hodnot
 - **konečné domény**
 - např. Booleovské (včetně NP-úplných problémů splnitelnosti)
 - výčtové
 - **nekonečné domény** – čísla, řetězce, ...
 - např. rozvrh prací – proměnné = počáteční/koncový den každého úkolu
 - vyžaduje **jazyk omezení**, např. $StartJob_1 + 5 \leq StartJob_3$
 - číselné *lineární* problémy jsou řešitelné, *nelineární* obecné řešení nemají
- **spojité hodnoty proměnných**
 - časté u reálných problémů
 - např. počáteční/koncový čas měření na Webbově teleskopu (závisí na astronomických, precedenčních a technických omezeních)
 - *lineární omezení* řešené pomocí **Lineárního programování** (omezení = lineární (ne)rovnice tvořící konvexní oblast) → jsou řešitelné v polynomiálním čase

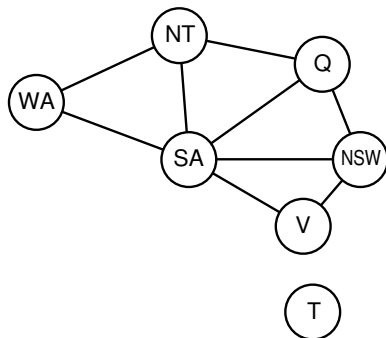
Varianty omezení

- **unární** omezení zahrnuje jedinou proměnnou
např. $SA \neq \text{zelená}$
- **binární** omezení zahrnují dvě proměnné
např. $SA \neq WA$
- omezení **vyššího řádu** zahrnují 3 a více proměnných
např. kryptoaritmetické omezení na sloupce u algebrogramu
- **preferenční** omezení (soft constraints), např. 'červená je lepší než zelená'
možno reprezentovat pomocí **ceny přiřazení** u konkrétní hodnoty a konkrétní proměnné \rightarrow hledá se **optimalizované řešení** vzhledem k ceně

Graf omezení

Pro **binární** omezení: **uzly** = proměnné, **hrany** = reprezentují jednotlivá omezení

Pro ***n*-ární** omezení: **hypergraf**: **○ uzly** = proměnné, **□ uzly** = omezení, **hrany** = použití proměnné v omezení



Algoritmy pro řešení CSP využívají této grafové reprezentace omezení

CLP – Constraint Logic Programming

X in 1..5, Y in 2..8, $X+Y \# = T$:

X in 1..5

Y in 2..8

T in 3..13

CLP – Constraint Logic Programming

`?X in +Min..+Max``?X in +Domain ...``A in 1..3 \\/8..15 \\/5..9 \\/100.``+VarList ins +Domain``fd_dom(?Var,?Domain) zjištění domény proměnné``X in 1..5, Y in 2..8, X+Y #= T:``X in 1..5``Y in 2..8``T in 3..13`

CLP – Constraint Logic Programming

?X in +Min..+Max

?X in +Domain ...

A in 1..3 \\/8..15 \\/5..9 \\/100.

+VarList ins +Domain

fd_dom(?Var,?Domain) zjištění domény proměnné

X in 1..5, Y in 2..8, X+Y # = T:

X in 1..5

Y in 2..8

T in 3..13

aritmetická omezení ...

- rel. operátory # =, # \=, # <, # <=, # >, # >=
- sum(Variables,RelOp,Suma)

výroková omezení ...

- # \ negace, # / \ konjunkce, # \ / disjunkce, # < == > ekvivalence

kombinatorická omezení ...

- all_distinct(List), global_cardinality(List, KeyCounts)

CLP – Constraint Logic Programming

`?X in +Min..+Max``?X in +Domain ...``A in 1..3 \\/8..15 \\/5..9 \\/100.``+VarList ins +Domain``fd_dom(?Var,?Domain)` zjištění domény proměnné`X in 1..5, Y in 2..8, X+Y # = T:``X in 1..5``Y in 2..8``T in 3..13`

aritmetická omezení ...

- rel. operátory `# =`, `# \ =`, `# <`, `# = <`, `# >`, `# > =`
- `sum(Variables, RelOp, Suma)`

výroková omezení ...

- `# \` negace, `# / \` konjunkce, `# \ /` disjunkce, `# < == >` ekvivalence

kombinatorická omezení ...

`all_distinct(List), global_cardinality(List, KeyCounts)``X in 1..5, Y in 2..8, X+Y # = T, labeling([X, Y, T]):``T = 3``X = 1``Y = 2`

hledá hodnoty podle omezení

Příklad – algebrogram

```
  S E N D
+ M O R E
-----
M O N E Y
```

Příklad – algebrogram

```
  S E N D
+ M O R E
-----
M O N E Y
```

Proměnné

Domény

Omezení

Příklad – algebrogram

```
  S E N D
+ M O R E
-----
M O N E Y
```

Proměnné {S, E, N, D, M, O, R, Y}
Domény
Omezení

Příklad – algebrogram

```
  S E N D
+ M O R E
-----
M O N E Y
```

Proměnné

$\{S, E, N, D, M, O, R, Y\}$

Domény

$D_i = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

Omezení

Příklad – algebrogram

```

  S E N D
+ M O R E
-----
M O N E Y

```

Proměnné $\{S, E, N, D, M, O, R, Y\}$

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

Omezení – $S > 0, M > 0$

– $S \neq E \neq N \neq D \neq M \neq O \neq R \neq Y$

– $1000 * S + 100 * E + 10 * N + D + 1000 * M +$
 $100 * O + 10 * R + E =$
 $10000 * M + 1000 * O + 100 * N + 10 * E + Y$

Příklad – algebrogram

```

  S E N D
+ M O R E
-----
M O N E Y

```

Proměnné $\{S, E, N, D, M, O, R, Y\}$

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

Omezení – $S > 0, M > 0$

– $S \neq E \neq N \neq D \neq M \neq O \neq R \neq Y$

– $1000 * S + 100 * E + 10 * N + D + 1000 * M + 100 * O + 10 * R + E =$

$10000 * M + 1000 * O + 100 * N + 10 * E + Y$

```
function MOREMONEY([S,E,N,D,M,O,R,Y])
```

```
  [S,E,N,D,M,O,R,Y] ins 0..9
```

```
  S #> 0; M #> 0
```

```
  all_distinct([S,E,N,D,M,O,R,Y])
```

```
  1000*S + 100*E + 10*N + D + 1000*M + 100*O + 10*R + E
```

```
    #= 10000*M + 1000*O + 100*N + 10*E + Y
```

```
  labeling([S,E,N,D,M,O,R,Y])
```

Příklad – algebrogram

$$\begin{array}{r}
 \text{S E N D} \\
 + \text{M O R E} \\
 \hline
 \text{M O N E Y}
 \end{array}$$

Proměnné $\{S, E, N, D, M, O, R, Y\}$

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

Omezení – $S > 0, M > 0$

– $S \neq E \neq N \neq D \neq M \neq O \neq R \neq Y$

– $1000 * S + 100 * E + 10 * N + D + 1000 * M + 100 * O + 10 * R + E = 10000 * M + 1000 * O + 100 * N + 10 * E + Y$

```
function MOREMONEY([S,E,N,D,M,O,R,Y])
```

```
  [S,E,N,D,M,O,R,Y] ins 0..9
```

```
  S #> 0; M #> 0
```

```
  all_distinct([S,E,N,D,M,O,R,Y])
```

```
  1000*S + 100*E + 10*N + D + 1000*M + 100*O + 10*R + E
```

```
    #= 10000*M + 1000*O + 100*N + 10*E + Y
```

```
  labeling([S,E,N,D,M,O,R,Y])
```

```
MoreMoney([S,E,N,D,M,O,R,Y]):
```

```
S = 9, E = 5, N = 6, D = 7, M = 1, O = 0, R = 8, Y = 2
```

Inkrementální formulace CSP

CSP je možné převést na **standardní prohledávání** takto:

- **stav** – přiřazení hodnot proměnným
- **počáteční stav** – prázdné přiřazení $\{\}$
- **přechodová funkce** – přiřazení hodnoty libovolné dosud nenastavené proměnné tak, aby výsledné přiřazení bylo konzistentní
- **cílová podmínka** – aktuální přiřazení je úplné
- **cena cesty** – konstantní (např. 1) pro každý krok

Inkrementální formulace CSP

CSP je možné převést na **standardní prohledávání** takto:

- **stav** – přiřazení hodnot proměnným
- **počáteční stav** – prázdné přiřazení $\{\}$
- **přechodová funkce** – přiřazení hodnoty libovolné dosud nenastavené proměnné tak, aby výsledné přiřazení bylo konzistentní
- **cílová podmínka** – aktuální přiřazení je úplné
- **cena cesty** – konstantní (např. 1) pro každý krok

1. platí beze změny pro **všechny** CSP!
2. prohledávací strom dosahuje hloubky n (počet proměnných) a řešení se nachází v této hloubce ($d = n$) \Rightarrow je vhodné použít **prohledávání do hloubky**

Prohledávání s navracením

- přiřazení proměnným jsou **komutativní**
tj. [1. $WA = \text{červená}$, 2. $NT = \text{zelená}$] je totéž jako
[1. $NT = \text{zelená}$, 2. $WA = \text{červená}$]
- stačí uvažovat pouze **přiřazení jediné proměnné** v každém kroku \Rightarrow
počet listů max. $|D_i|^n$, větvení jde ovlivnit **obecnými strategiemi**
- prohledávání do hloubky pro CSP – tzv. **prohledávání s navracením**
(*backtracking search*)
- **prohledávání s navracením** je základní **neinformovaná strategie** pro
řešení problémů s omezujícími podmínkami
- schopný vyřešit např. problém n -dam pro $n \approx 25$ (naivní řešení 10^{69} ,
vlastní sloupce 10^{25})

Příklad – problém N dam

```
function QUEENS( $N$ )
```

```
   $L = [q_{y1}, q_{y2}, \dots, q_{yN}]$  # seznam  $N$  proměnných
```

```
   $L$  ins 1.. $N$ 
```

```
  for  $i \leftarrow 1$  to  $N-1$  do
```

```
    for  $j \leftarrow i+1$  to  $N$  do
```

```
      NoThreat( $L[i]$ ,  $L[j]$ ,  $j-i$ )
```

```
  labeling( $L$ )
```

```
function NOTHREAT( $Y_1$ ,  $Y_2$ ,  $J$ )
```

```
  return  $Y_1 \neq Y_2$  and  $Y_{1+J} \neq Y_2$  and  $Y_{1-J} \neq Y_2$ 
```

Příklad – problém N dam

function QUEENS(N)

$L = [q_{y1}, q_{y2}, \dots, q_{yN}]$ # seznam N proměnných

L ins 1.. N

1. definice proměnných a domén

for $i \leftarrow 1$ to $N-1$ do

for $j \leftarrow i+1$ to N do

2. definice omezení

NoThreat($L[i]$, $L[j]$, $j-i$)

labeling(L)

3. hledání řešení

function NOTHREAT(Y_1 , Y_2 , J)

return $Y_1 \# \neq Y_2$ and $Y_{1+J} \# \neq Y_2$ and $Y_{1-J} \# \neq Y_2$

Příklad – problém N dam

function QUEENS(N)

$L = [q_{y1}, q_{y2}, \dots, q_{yN}]$ # seznam N proměnných

L ins 1.. N

1. definice proměnných a domén

for $i \leftarrow 1$ to $N-1$ **do**

for $j \leftarrow i+1$ to N **do**

2. definice omezení

 NoThreat($L[i]$, $L[j]$, $j-i$)

labeling(L)

3. hledání řešení

function NOTHREAT(Y_1 , Y_2 , J)

return $Y_1 \# \neq Y_2$ and $Y_1+J \# \neq Y_2$ and $Y_1-J \# \neq Y_2$

Queens(4):

[2,4,1,3]

[3,1,4,2]

Ovlivnění efektivity prohledávání s navracením

Obecné metody **ovlivnění efektivity**:

- Která proměnná dostane hodnotu v tomto kroku?
- V jakém pořadí zkoušet přiřazení hodnot konkrétní proměnné?
- Můžeme předčasně detekovat nutný neúspěch v dalších krocích?

Ovlivnění efektivity prohledávání s navracením

Obecné metody **ovlivnění efektivity**:

- Která proměnná dostane hodnotu v tomto kroku?
- V jakém pořadí zkoušet přiřazení hodnot konkrétní proměnné?
- Můžeme předčasně detekovat nutný neúspěch v dalších krocích?

používané **strategie**:

- nejomezenější proměnná → vybrat proměnnou s nejméně možnými hodnotami



Ovlivnění efektivity prohledávání s navracením

Obecné metody **ovlivnění efektivity**:

- Která proměnná dostane hodnotu v tomto kroku?
- V jakém pořadí zkusit přiřazení hodnot konkrétní proměnné?
- Můžeme předčasně detekovat nutný neúspěch v dalších krocích?

používané strategie:

- nejomezenější proměnná → vybrat proměnnou s nejméně možnými hodnotami
- nejvíce omezující proměnná → vybrat proměnnou s nejvíce omezeními na zbývající proměnné



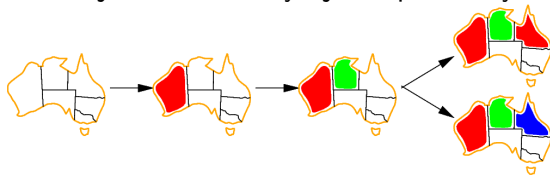
Ovlivnění efektivity prohledávání s navracením

Obecné metody **ovlivnění efektivity**:

- **Která proměnná** dostane hodnotu v tomto kroku?
- **V jakém pořadí** zkusit **přiřazení hodnot** konkrétní proměnné?
- Můžeme **předčasně detekovat** nutný **neúspěch** v dalších krocích?

používané strategie:

- **nejomezenější proměnná** → vybrat proměnnou s nejméně možnými hodnotami
- **nejvíce omezující proměnná** → vybrat proměnnou s nejvíce omezeními na zbývající proměnné
- **nejméně omezující hodnota** → pro danou proměnnou – hodnota, která zruší nejmíň hodnot zbývajících proměnných **SliDo**



umožňuje **1 hodnotu** pro SA

umožňuje **0 hodnot** pro SA

Ovlivnění efektivity prohledávání s navracením

Obecné metody **ovlivnění efektivity**:

- **Která proměnná** dostane hodnotu v tomto kroku?
- **V jakém pořadí** zkoušet **přiřazení hodnot** konkrétní proměnné?
- Můžeme **předčasně detekovat** nutný **neúspěch** v dalších krocích?

používané strategie:

- **nejomezenější proměnná** → vybrat proměnnou s nejméně možnými hodnotami
- **nejvíce omezující proměnná** → vybrat proměnnou s nejvíce omezeními na zbývající proměnné
- **nejméně omezující hodnota** → pro danou proměnnou – hodnota, která zruší nejmíň hodnot zbývajících proměnných **Slido**
- **dopředná kontrola** → udržovat seznam možných hodnot pro zbývající proměnné
- **propagace omezení** → navíc kontrolovat možné nekonzistence mezi zbývajícími proměnnými

Ovlivnění efektivity v CLP

V Prologu (CLP) možnosti ovlivnění efektivity – **labeling(Typ, ...)**:

```
?- constraints (Vars, Cost),  
   labeling ([ ff , bisect , down, min(Cost)], Vars).
```

- výběr proměnné – **leftmost**, **min**, **max**, **ff**, ...
- dělení domény – **step**, **enum**, **bisect**
- prohledávání domény – **up**, **down**
- uspořádání řešení – bez uspořádání nebo **min(X)**, **max(X)**, ...

Systemy pro řešení omezujících podmínek

- **Prolog** – SWI, CHIP, ECLiPSe, SICStus Prolog, Prolog IV, GNU Prolog, IF/Prolog
- **C/C++** – CHIP++, ILOG Solver, Gecode
- **Java** – JCK, JCL, Koalog
- **LISP** – Screamer
- **Python** – logilab-constraint www.logilab.org/852, python-constraint