VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA STROJNÍHO INŽENÝRSTVÍ
ÚSTAV AUTOMATIZACE A INFORMATIKY

ALGORITMY UMĚLÉ INTELIGENCE

ŘEŠENÍ ÚLOH VE STAVOVÉM PROSTORU

Doc. RNDr. Ing. Tomáš Březina, CSc,

Obsah

1 Stavová reprezentace úloh

Stavový prostor je (uspořádaná) dvojice $S = (D, \Phi)$, kde
$D \equiv \{s_i\}$ je konečná *množina stavů* a
$\Phi \equiv \{\varphi_i\}$ je konečná *množina operátorů* reprezentujících přechody mezi stavy.

Poznámky

- *Každý operátor $\varphi_i : D \to D$ je (parciálním) zobrazením D do D.*
- *Operátor φ_i může být reprezentován hranou grafu.*

Úloha U nad stavovým prostorem S je dvojice $U = (s_0, C)$, kde
$s_0 \in D$ je *počáteční stav* a
$C \subseteq D$ je *množina cílových stavů*.

Plán P ***pro danou úlohu*** U (***řešení úlohy*** U) je posloupnost operátorů

$$P = (\varphi_1, \varphi_2, \ldots, \varphi_n),$$

ke které lze přiřadit posloupnost stavů $(s_1, s_2, \ldots, s_n)$:

$$s_1 = \varphi_1(s_0),$$
$$s_2 = \varphi_2(s_1),$$
$$\vdots$$
$$s_n = \varphi_n(s_{n-1}), s_n \in C.$$

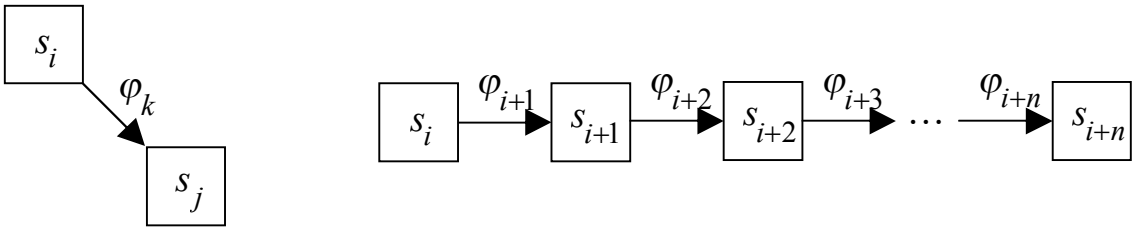
Stavový prostor bývá nejčastěji detailněji reprezentován orientovaným grafem, kde
- stavy jsou reprezentovány uzly grafu a
- operátory φ_i jsou reprezentovány orientovaným hranami grafu stavového prostoru

Tato reprezentace umožňuje využívat obecných poznatků teorie grafů.

Poznámky

- *Orientovaný graf je nástrojem přirozené reprezentace (nekomutativních) binárních vztahů (relací).*
- *Terminologicky bývá ztotožňován stav s uzlem a přechod s hranou.*
- *Uzel s_j bezprostředně následující uzel s_i (**bezprostřední následník**) je uzel vyhovující podmínce $s_j = \varphi_k(s_i)$.*

- ***Orientovaná cesta*** (délky n) z s_i do s_{i+n} znamená posloupnost orientovaných hran, vedoucí z s_i do s_{i+n}.



Stavová reprezentace úloh tvoří teoretický (nebo aspoň metodologický) základ většiny metod UI (= formální chápání řešení úloh jako transformace jistého počátečního stavu na koncový (požadovaný) prostřednictvím posloupnosti operací). Umožňuje definovat řešení problému jako kombinaci

- dílčích algoritmů provádějících řešení dílčích problémů (řešení dílčího problému definuje přechod mezi stavy) a
- algoritmu *pohledávání* zjišťujícího posloupnost operátorů (algoritmů) řešících úlohu.

Formalismus umožňuje odlišovat

- deklarativně reprezentované poznatky – vyjadřují, co je nebo má být poznáno; zjednodušeně – popisují stavy řešení (báze dat).
- procedurálně reprezentované znalosti – vyjadřují, jak poznávat nebo odvozovat; zjednodušeně – popisují přechody mezi stavy řešení (produkční pravidla).

Speciálním a hojně používaným případem stavové reprezentace je *produkční systém*.

2 Produkční systém

Produkční systém tvoří
- ***produkční pravidla***,
- ***báze dat*** a
- ***řídicí strategie***.

2.1 Produkční pravidla

Tvar

$$\{\text{situace S}\} \rightarrow \{\text{akce A}\}$$

Interpretace

„Nastala – li v bázi dat situace S, vykonej akci A“

Činnost produkčního systému

v cyklech

$$\text{rozpoznání situace} \rightarrow \text{vykonání akce}$$

Poznámky

- *Provedení akce znamená provedení pravidla; mění obsah báze dat (přechod stavu).*
- *Produkční pravidla odpovídají operátorům φ_i .*

2.2 Báze dat

Báze dat obsahuje popis okamžitého stavu úlohy (tj. model řešené úlohy, popř. data o řešené úloze). Báze dat odpovídá stavovému prostoru. Bývá členěna na část
- ***trvalou*** , která obsahuje trvale platné skutečnosti, a
- ***dílčí*** , která obsahuje aktuálně platná data (*pracovní paměť*).

2.3 Řídicí strategie

určuje jak a v jakém pořadí aplikovat pravidla na bázi dat. Rozeznáváme
- *přímý režim řízení (**přímé řízení**, strategie řízená daty, data – driven strategy)*, probíhá od počátečního stavu k některému ze stavů cílových a
- zpětný režim řízení (**zpětné řízení**, strategie řízená cílem, goal – driven strategy), probíhá od cílového stavu k počátečnímu stavu.

Poznámky

- *Oba způsoby lze vhodně kombinovat.*

Každá strategie musí

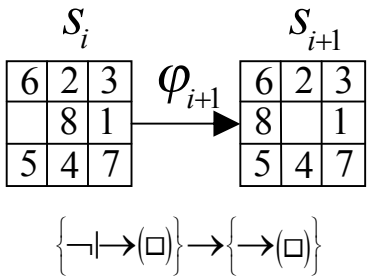
- vést k *prohledávání*, tj. způsobovat pohyb po stavech stavového prostoru a zabraňovat cyklům v posloupnosti pravidel,
- být systematická, tj. vést na postupné prozkoumávání všech stavů stavového prostoru.

Příklad produkčního systému – Lišák

Na čtvercové hrací ploše s 9 (3x3) možnými pozicemi je celkem 8 hracích kamenů očíslovaných od 1 do 8. Jedna z pozic zůstává neobsazena. Hra začíná s libovolným uspořádáním kamenů, účelem hry je dosáhnout postupnými posuny kamenů jediného, předem stanoveného uspořádání.

Úloha

- Stav úlohy je poloha hracích kamenů
- Počáteční stav
- Jediný cílový stav
- Přechod mezi stavy je posun hracího kamene



$$\{\neg|\rightarrow(\square)\}\rightarrow\{\rightarrow(\square)\}$$

Produkční pravidla

$\{\neg\bar{\uparrow}(\square)\}\rightarrow\{\uparrow(\square)\}$	„*Není–li prázdné políčko u horního okraje, hni políčkem nahoru*“
$\{\neg\rightarrow\|(\square)\}\rightarrow\{\rightarrow(\square)\}$	„*Není–li prázdné políčko u pravého okraje, hni políčkem napravo*“
$\{\neg\underline{\downarrow}(\square)\}\rightarrow\{\downarrow(\square)\}$	„*Není–li prázdné políčko u dolního okraje, hni políčkem dolu*“
$\{\neg\|\leftarrow(\square)\}\rightarrow\{\leftarrow(\square)\}$	„*Není–li prázdné políčko u levého okraje, hni políčkem nalevo*“

Řídicí strategie

- postupně zkoušej produkční pravidla,
- nepřipusť cykly v použití pravidel a
- STOP v okamžiku, kdy je dosaženo cíle.

Poznámky

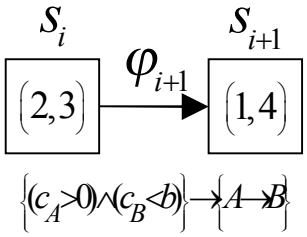
- *řídicích strategií může být více, např. s použitím předdefinovaných priorit pravidel.*

Příklad produkčního systému – Přelévání vody

Je dán neomezený zdroj vody a dvě nádoby, které nemají žádné označení míry. Větší z nich A má obsah a litrů, menší B má obsah b litrů. Na počátku řešení úlohy jsou obě nádoby prázdné (počáteční stav). Cílem řešení úlohy je dosáhnout toho, že nádoba A je prázdná a v nádobě B je např. $2(a-b)$ litrů vody. K dispozici jsou operace – vylití nádoby, naplnění nádoby a přelití vody z nádoby do nádoby.

Úloha

- Stav úlohy je uspořádaná dvojice $\langle c_A, c_B \rangle$ (množství vody c_A, c_B v nádobách A, B)
- Počáteční stav je $\langle 0, 0 \rangle$
- Jediný cílový stav $\langle 0, 2(a-b) \rangle$
- Přechod mezi stavy je použití povolené operace



Produkční pravidla

$\{c_A > 0\} \rightarrow \{A \rightarrow\}$	Je–li $c_A > 0$, vylej A
$\{c_B > 0\} \rightarrow \{B \rightarrow\}$	Je–li $c_B > 0$, vylej B
$\{c_A < a\} \rightarrow \{\rightarrow A\}$	Je–li $c_A < a$, naplň A
$\{c_B < b\} \rightarrow \{\rightarrow B\}$	Je–li $c_B < b$, naplň B
$\{(c_A > 0) \wedge (c_B < b)\} \rightarrow \{A \rightarrow B\}$	Je–li $c_A > 0$ a $c_B < b$, přelej A do B
$\{(c_A < a) \wedge (c_B > 0)\} \rightarrow \{A \leftarrow B\}$	Je–li $c_A < a$ a $c_B > 0$, přelej B do A

Řídicí strategie

viz strategie v předchozím příkladu

3 Složitost stavů X složitost operátorů

Volba prvků stavového prostoru (stavy, přechody mezi nimi, množ. počátečních a cílových stavů), či adekvátního produkčního systému není úloha jednoznačná. Obecně platí, že

- čím jednodušší strukturu mají stavy, tím méně obecná (a tudíž složitější) jsou pravidla, tj. jsou větší časové nároky na realizaci přechodu mezi stavy,
- čím obecnější jsou pravidla, tím více je třeba zavést stavů (mnohdy lišících se v pouze detailech), tj. rozsáhlejší stavový prostor.

Proto je nutný kompromis.

Řešení úloh má obvykle nedeterministický charakter – není pevně definováno pořadí aplikace pravidel v případě, že na daný stav je možno aplikovat více pravidel (***konfliktní množina pravidel***). Výběr konkrétního pravidla řeší řídicí mechanismus.

Prohledávání je velmi důležitý postup vhodný pro řešení (složitých) úloh, které nelze řešit přímo známými výpočetními postupy.

4 Prohledávání stavového prostoru

Řídicí mechanismus realizuje řídicí strategii. Jde o algoritmus, který

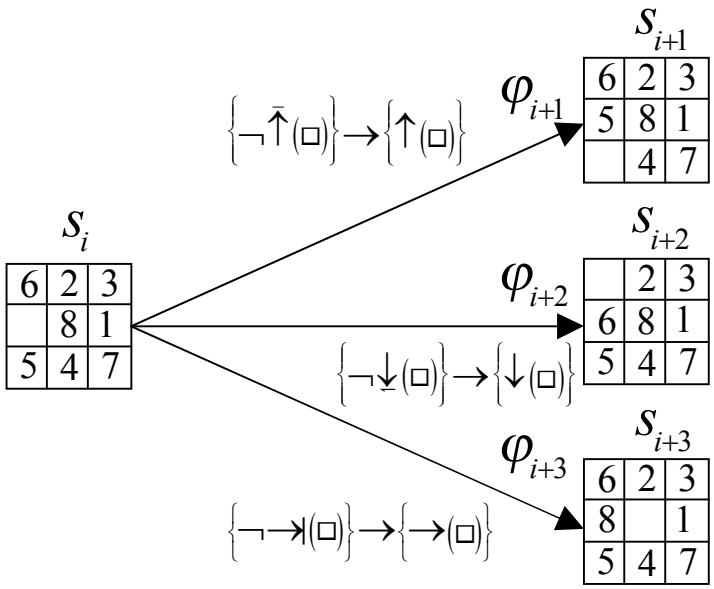
- poskytuje návod pro výběr pravidel z konfliktní množiny pravidel v každém kroku prohledávání stavového prostoru,
- během prohledávání stavového prostoru generuje strom, který je podgrafem orientovaného grafu, reprezentujícího stavový prostor.

Při přímém řízení se nejprve generuje a expanduje počáteční uzel s_0, v dalším procesu prohledávání se pak expandují některé z dříve expandovaných uzlů. Je – li vygenerován uzel $s \in C$, prohledávání končí (ve stromu řešení existuje orientovaná cesta od s_0 do s).

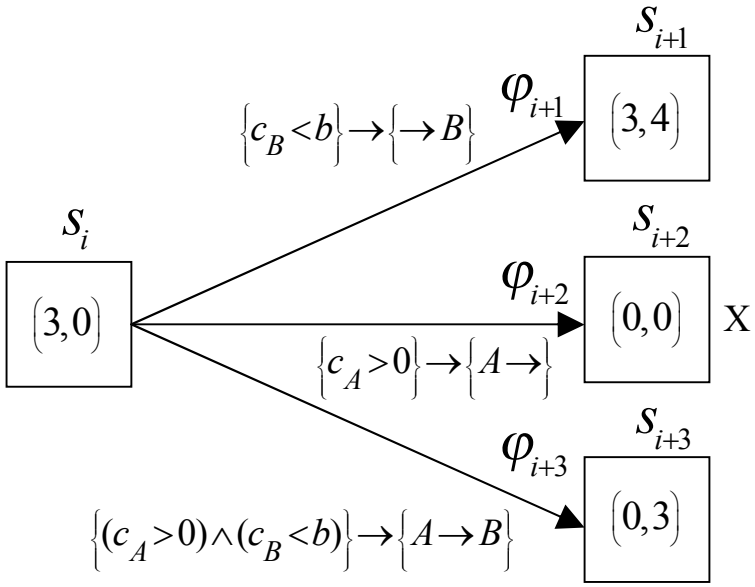
Poznámka

Expanze uzlu znamená nalezení množiny všech možných bezprostředně následujících uzlů.

Příklad expanze – Lišák



Příklad expanze - Přelévání vody



4.1 Metody prohledávání

Vzhledem k velikosti stavového prostoru může být systematické prohledávání stavového prostoru velmi neefektivní (zbytečně se prohledává značná část stavového prostoru, která nevede k cíli). Prohledávání lze omezit znalostí o řešeném problému.

Znalosti mají někdy charakter empirický, mohou to být neexaktní poznatky, které jsou často užitečné při řešení, ale často nezaručují, že povedou k řešení (***heuristické znalosti***, ***heuristiky***).

Heuristiky se používají tam, kde není k dispozici exaktní algoritmus. Ze dvou řešitelů stejného problému je lepší ten, který je vybaven lepší množinou heuristik (prohledává menší část stavového prostoru, postupuje přímočařeji k cíli a jeho způsob řešení se jeví jako „inteligentnější“).

Podle využití znalostí o úloze je prohledávání
- ***neinformované*** (nevyužívá znalostí o úloze)
- ***informované*** (využívá znalostí o úloze)

4.1.1 Neinformované metody prohledávání

Metody neinformovaného prohledávání se dělí z hlediska pořadí, v jakém jsou uzly expandovány, na:

4.1.1.1 slepé prohledávání do šířky

- nejdříve se expanduje uzel s nejmenší hloubkou,
- nalezne nejmenší řešení (s nejmenším počtem operátorů – nejkratší cestu).

4.1.1.2 slepé prohledávání do hloubky

- nejdříve se expanduje uzel s největší hloubkou,
- má nižší nároky na paměť (uchovává pouze uzly na cestě od počátečního stavu ke stavu právě expandovanému),
- často spojeno s omezením maximální prohledávané hloubky, při jejím dosažení se používá mechanismu navracení.

Poznámky

- *Hloubka uzlu = počet hran od počátečního uzlu k uzlu.*
- *Kompromis mezi prohledáváním do šířky a do hloubky:*

4.1.1.3 Algoritmus $DFID$ (algoritmus iterativního prohlubování)

- Úplné prohledávání do hloubky tak, že se v každé iteraci zvyšuje povolená hloubka prohledávání o 1.
- První nalezené řešení je optimální (ve smyslu nejkratší délky cesty).

Poznámky

- *Neinformované metody jsou použitelné jen v nejjednodušších úlohách.*
- *Nepoužívání znalostí o úloze vede na prohledávání příliš velkých částí stavového prostoru.*
- *Představují metodologický podklad pro složitější, informované strategie.*

4.1.2 Informované metody prohledávání

4.1.2.1 Hodnotící funkce f:

- pro každý uzel stromu řešení určí jeho ohodnocení,
- hodnoty se používají pro výběr uzlu k expanzi,
- pokud hodnotící funkce dobře postihují vlastnosti a charakter úlohy, budou vždy expandovány „nejperspektivnější“ uzly a zabrání se prohledávání cest, které nevedou k cíli,
- čím kvalitnější heuristické znalosti o dané úloze se v f využijí, tím efektivnější bude prohledávání.

4.1.2.2 Gradientní algoritmus

- Vždy se expanduje ten uzel, který měl dosud nejlepší hodnotící funkci, a vyhodnocují se jeho následníci.
- „Rodič“ stejně jako „sourozenci“ tohoto uzlu jsou okamžitě zapomenuty. Pracuje se pouze s právě expandovaným uzlem.
- Hledání je zastaveno, když je dosaženo stavu, který má lepší ohodnocení funkcí f než jeho následníci.

Poznámky

- *Základní nevýhodou všech gradientních metod je riziko uvíznutí v lokálním extrému.*
- *Není vyloučeno zacyklení (pohyb po nekonečně dlouhé cestě, např. s konstantním ohodnocením).*

4.1.2.3 Algoritmus uspořádaného prohledávání

- Je gradientní algoritmus rozšířený o paměť.
- Expandují se stavy s minimální hodnotou funkce f.
- Používá obvyklé seznamy:
  - OPEN – seznam neexpandovaných stavů
  - CLOSED – seznam již expandovaných stavů
- Prvky seznamů jsou trojice $\langle \text{jméno uzlu}, f, \text{jméno rodičovského uzlu} \rangle$ (zápis stavu do seznamu označuje zápis uvedené trojice).

4.1.2.3.1 Obecný algoritmus

1. Počáteční stav zapiš do seznamu OPEN, seznam CLOSED je prázdný.
2. Pokud je seznam OPEN prázdný, řešení neexistuje, ukonči prohledávání.

3. Ze seznamu OPEN vyber stav i s nejmenší hodnotou $f(i)$. V případě většího počtu stavů se stejnou minimální hodnotou $f(i)$ prověř, zda některý z těchto stavů není stavem cílovým, v takovém případě jej vyber; jinak vyber mezi stavy se stejnou minimální hodnotou $f(i)$ libovolně.
4. Vymaž stav i ze seznamu OPEN a zařaď jej do seznamu CLOSED.
5. Je–li stav i cílovým stavem, řešení je nalezeno, ukonči prohledávání.
6. Expanduj stav i; pro každého následníka j stavu i vypočítej hodnotu $f(j)$. Pokud stav j není ani v seznamu OPEN, ani v seznam CLOSED, zařaď jej do seznamu OPEN. Pokud je stav j v seznamu OPEN nebo CLOSED, avšak s ohodnocením větším než právě vypočtené $f(j)$, změň jeho ohodnocení na $f(j)$, změň jméno rodičovského uzlu v zápisu uzlu a zařaď ho do seznamu OPEN.
7. Pokračuj krokem č.2.

Poznámky

Algoritmus uspořádaného prohledávání má celou řadu modifikací. Velmi známý je:

4.1.2.3.2 Algoritmus paprskového prohledávání

- pracuje se seznamem OPEN konečné délky m, do seznamu OPEN se zapisují pouze ty nově expandované uzly, které mají lepší ohodnocení než uzly dosud obsažené v seznamu.
- To má za následek značnou redukci expandovaného stromu, ale nemusí vést k nalezení cílového stavu.

4.1.2.3.3 Algoritmus A

jde o algoritmus uspořádaného prohledávání s hodnotící funkcí
$f(i) = g(i) + h(i)$,
kde $g(i)$ je cena přechodu z počátečního stavu s_0 do stavu i, $h(i)$ je cena optimální cesty ze stavu i do některého z cílových stavů $s \in C$.

Poznámky

Funkci $f(i)$ lze chápat jako cenu přechodu z počátečního stavu do cílového stavu přes stav i.

- Ve většině úloh $g(i)$, $h(i)$ není známo, používá se jejich odhadů $g^*(i)$, $h^*(i)$.
- Funkci $g(i)$ odhadneme minimální dosud zjištěnou cenou $g^*(i)$ přechodu z počátečního stavu do stavu i
- Funkci $h(i)$ nahrazujeme funkcí $h^*(i)$, která kvantitativně vyjadřuje náš odhad ceny cesty z uzlu i do některého z cílových stavů. Odhad vyjadřuje naši heuristickou znalost o tom, jaké jsou šance nalézt (optimální) řešení, pokračovali–

li bychom expanzí daného uzlu. Funkce $h^*(i)$ je proto *nositelem heuristické informace* a bývá proto nazývána ***heuristickou funkcí***.

Poznámky

Heuristická funkce je pro efektivnost prohledávání podstatná.

Řekneme, že algoritmus prohledávání je přípustný, jestliže vždy nalezne optimální cestu, pokud tato cesta existuje.

Algoritmus A s přípustnou heuristickou funkcí je algoritmus A^*.

Heuristická funkce je přípustná, je–li $h^*(i) \geq 0$ a $h^*(i) \leq h(i)$, pro všechny uzly i (tj. h^* je nezáporný dolní odhad h).
Čím je h^* lepším dolním odhadem h, tím menší část stavového prostoru se prohledává při hledání optimálního řešení (při $h^* = h$ expanduje algoritmus A^* pouze stavy na cestě k cílovému stavu).

Říkáme, že algoritmus A_1^* je *lépe informovaným algoritmem* než A_2^*, jestliže pro každý stav i je $h_1^*(i) \geq h_2^*(i)$. A_2^* pak expanduje všechny stavy, které expanduje A_1^*. Mohou však existovat stavy, které expanduje A_2^* a neexpanduje A_1^*.

Funkce $g^*(i)$

- Umožňuje vybírat stavy k expanzi na základě posouzení jak „dobrá“ byla cesta z počátečního do daného stavu. Tj. ne vždy jsou k expanzi vybírány uzly, které se zdají být nejblíže k cíli.
- Velmi důležitá, pokud nejde pouze o nalezení řešení, ale i o optimalizaci procesu nalézání řešení. Pro nalezení řešení jakýmkoliv způsobem stačí volba $g^* = a$ (konstantní funkce), obvykle $a = 0$.

Funkce $h^*(i)$

- ***Algoritmus se stejnoměrnou cenou*** $h^*(i) = a$ (prohledávání řízeno pouze funkcí $g^*(i)$)
- ***Algoritmus větví a mezí***
  je algoritmus se stejnoměrnou cenou, který si po nalezení cílového stavu zapamatuje jeho cenu a pokračuje v prohledávání. Ze seznamu OPEN však automaticky vyřazuje uzly z vyšší cenou, než je zapamatovaná cena cílového stavu. Pokud je nalezen cílový uzel s nižší cenou, pamatuje se nižší cena a algoritmus pokračuje až do okamžiku, kdy je seznam prázdný.
- ***Algoritmus náhodného prohledávání***

$$g^*(i) = a,\ h^*(i) = b$$

Poznámky

Při $g^*(i) = \text{hloubka}(i)$, $h^*(i) = 0$ *degeneruje algoritmus* A^* *na algoritmus slepého prohledávání do šířky.*

- *Nelze ověřit, že jde o* A^* *(tj. podmínku přípustnosti* $0 \le h^* \le h$*). Přípustnost heuristiky lze prokázat pouze kompletním prozkoumáním stavového prostoru, což lze provést pouze u jednoduchých úloh. Pak ale není heuristického prohledávání zapotřebí.*
- *Nemožnost posouzení přípustnosti heuristické funkce v praktických úlohách je největší slabinou algoritmů* A^*.

Zmírnění požadavku přípustnosti

Pokud h^* někdy (ale málokdy) nadhodnocuje h a hodnotu větší než δ, pak algoritmus A (nejde už o A^*) zřídkakdy nachází řešení, jehož cena je větší o více než δ ve srovnání s cenou skutečně optimální cesty.

4.1.2.3.4 Algoritmus A^* *s monotonní* $h^*(i)$

Přípustnost algoritmu A^* nezávisí na volbě $g(i)$ (viz 6. krok algoritmu uspořádaného prohledávání, který prověřuje, jestli během prohledávání nebyla nalezena kratší cesta do již dříve expandovaného stavu).

Je–li $h^*(i)$ monotonní, lze 6. krok algoritmu A podstatně zjednodušit (je možné vypustit větu „*Pokud stav* j *je již v seznamu OPEN nebo CLOSED, ... a zařaď ho do seznamu OPEN.*“)

Poznámky

- $h^*(i)$ *je monotonní, platí–li* $h^*(i) - h^*(j) \le \text{cena}(i,j)$, *kde* $\text{cena}(i,j)$ *znamená cenu přechodu od stavu* i *do stavu* j.
- *Podstatné zjednodušení nemusí znamenat obecně zrychlení prohledávání.*

4.1.2.3.5 Algoritmus IDA^*

- je kombinací algoritmu A^* s algoritmem $DFID$.
- Algoritmus provádí iterativní prohledávání do hloubky, přičemž vylučuje ty uzly, jejichž cena přesahuje prahovou hodnotu.
- Hodnota prahu se dynamicky mění, na začátku je rovna odhadu f^* ceny počátečního uzlu.
- V každé iteraci se práh zvyšuje o minimální hodnotu, o kterou byl v předchozím kroku překročen.

Poznámky

- *Využívá–li přípustnou heuristickou funkci, zaručuje nalezení optimálního řešení.*
- *Nízká paměťová náročnost (preferováno prohledávání do hloubky).*

4.1.2.4 Dosud uvedené algoritmy

- mají charakter základních (elementárních) algoritmů prohledávání stavového prostoru,
- prakticky bývají kombinovány s některými dalšími metodami, vycházejícími z představy stavového prostoru se složitostí redukovanou znalostmi o řešené úloze.

4.1.3 Další metody informovaného prohledávání

- Pravidla se složitějšími předpoklady,
- Metaznalosti,
- Metoda generování a testování,
- Metoda užití analogie,
- Rozklad úlohy na podúlohy.

4.1.3.1 Pravidla se složitějšími předpoklady

- Zahrnují znalosti do předpokladů pravidel v podobě dalších předpokladů. Tím se sníží počet stavů, na které je pravidlo použitelné (a tak i nově generovaných).
- Prohledává se menší část stavového prostoru.

Příklad - Přelévání vody

Znalosti o úloze:

1. Nevyprazdňuj obě nádoby.
2. Nenaplňuj obě nádoby.
3. Nenaplňuj nádobu, je–li druhá prázdná a není–li naplňovaná prázdná
4. Nevyprazdňuj nádobu, je–li druhá plná.
5. Nepřelévej, pokud by potom byla jedna nádoba plná a druhá prázdná.

Produkční pravidla

$$\{(c_A > 0) \wedge (c_B > 0) \wedge (c_B \neq b)\} \rightarrow \{A \rightarrow\} \quad \text{Vylej } A$$

$$\{(c_A > 0) \wedge (c_B > 0) \wedge (c_A \neq a)\} \rightarrow \{B \rightarrow\} \quad \text{Vylej } B$$

$$\{(c_A < a) \wedge (c_B \neq b) \wedge ((c_A = 0) \vee (c_B \neq 0))\} \rightarrow \{\rightarrow A\} \quad \text{Naplň } A$$

$$\{(c_A \neq a) \wedge (c_B < b) \wedge ((c_A \neq 0) \vee (c_B = 0))\} \rightarrow \{\rightarrow B\} \quad \text{Naplň } B$$

$$\{(c_A > 0) \wedge (c_B < b) \wedge (c_A + c_B \neq b)\} \rightarrow \{A \rightarrow B\} \quad \text{Přelej } A \text{ do } B$$

$$\{(c_A < a) \wedge (c_B > 0) \wedge (c_A + c_B \neq a)\} \rightarrow \{A \leftarrow B\} \quad \text{Přelej } B \text{ do } A$$

Metaznalosti

jsou pravidla (***metapravidla***) používání pravidel definujících úlohu.

Poznámky

Metapravidla mají nejčastěji tvar

$$\{\text{situace S}\} \rightarrow \{\text{preference A}\}$$

s interpretací

„Nastala – li v bázi dat situace S, preferuj pravidlo A“.

4.1.3.2 Metoda generování a testování

Znalosti o úloze jsou rozděleny mezi

- ***generátor***, který expanduje stav (nabízí jednotlivé stavy), a
- ***tester***, který stavy nabízené generátorem testuje a odmítá, nesplňují–li zadaná omezení.

Poznámky

- Velmi efektivní je ***metoda hierarchicky uspořádaného generování a testování***, která umožňuje odmítnout stav na základě jenom jeho částečného popisu (jediným testem lze naráz zamítnout větší počet nevyhovujících stavů).

4.1.3.3 Metoda užití analogie

Řešení úlohy v prostoru I lze použít jako podklad pro řešení úlohy v prostoru II

- je–li odhalena určitá podobnost mezi dvěma stavovými prostory pro dvě, jinak i zcela odlišné úlohy, nebo
- byl–li řešen obdobný případ.

Poznámky

Toho využívá tzv. ***usuzování na základě příkladů*** *(****případové usuzování****)*

4.1.3.4 Rozklad úlohy na podúlohy

Pro řadu úloh je velmi obtížné (ne–li nemožné) definovat heuristiky ve formě hodnotící funkce, pravidel, či metapravidel. Někdy je však možné získat heuristiky, kterými lze původní úlohu rozložit na (snadněji řešitelné) podúlohy. Není–li podúloha elementární, můžeme se ji znovu pokusit rozložit na podúlohy atd.

Rozklad úlohy lze reprezentovat orientovaným ***AND/OR grafem***, jehož uzly znázorňují úlohu a podúlohy. Uzly, které nejsou listy, mohou být typu:

- ***AND*** – představují konjunkci podúloh (k řešení úlohy je nezbytné, aby každá z podúloh byla řešitelná),
- ***OR*** – představují disjunkci podúloh (k řešení úlohy stačí, aby aspoň jedna z podúloh byla řešitelná).

Jestliže víme, že výsledné řešení musí procházet jistým mezilehlým stavem (tzv. mostem), lze původní úlohu rozdělit na dvě podúlohy:

- podúlohu I s počátečním stavem s_0 a cílovým stavem s_M,
- podúlohu II s počátečním stavem s_M a cílovým stavem $s_c \in C$.

Poznámky

Uvažujeme–li právě 2 aplikovatelné operátory v každém ze stavů, je v nejhorším případě slepého prohledávání prostorů obou podúloh třeba vygenerovat

$2^{n_I} + 2^{n_{II}}$ *uzlů,*

kde n_I, n_{II} je přípustná hloubka stromu ve stavovém prostoru podúlohy I, resp. II. V nejhorším případě slepého prohledávání prostoru celé úlohy je třeba vygenerovat

$2^{n_I+n_{II}}$ *uzlů.*

Protože $2^{n_I}+2^{n_{II}}<2^{n_I+n_{II}}$, je výhodnější rozdělení úlohy, přičemž úspora generování může být značná.

Nechť je každý stav popsán R složkami

$$s_i \equiv \left(s_{i,1}, s_{i,2}, \ldots, s_{i,R}\right).$$

Potom lze úlohu U stručně zapsat jako

$$U: \quad s_0 \equiv \left(s_{0,1}, s_{0,2}, \ldots, s_{0,R}\right) \rightarrow s_c \equiv \left(s_{c,1}, s_{c,2}, \ldots, s_{c,R}\right).$$

Úloha U bude vyřešena nalezením operátoru, který odstraňuje všechny diference. V případě

- triviální úlohy, kdy $s_{0,r}=s_{c,r}$, $r=1,2,\ldots,R$, je úloha vyřešena okamžitě,
- netriviální úlohy, kdy $s_{0,r}\neq s_{c,r}$, $r\in\{1,2,\ldots,R\}$, tj. „existuje diference aspoň jedné z odpovídajících si složek“, lze diference někdy měřit, jindy lze pouze konstatovat „diference existuje/neexistuje“.

Nejjednodušší by bylo rozložit úlohu na R podúloh, z nichž by každá zajišťovala odstranění diference jedné složky. To není obecně použitelné díky možným ***vedlejším (postranním) efektům*** jednotlivých operátorů, kdy kromě odstranění diference jedné složky může operátor změnit i jiné diference.

Předem není známo, která pravidla budou použita, proto díky vedlejším efektům není možno stanovit ani cílové stavy podúloh a počáteční stavy navazujících podúloh (mosty). Vedlejší efekt může navíc obnovit dříve odstraněnou diferenci.

Řídicí algoritmus musí být schopen obecně formulovat podúlohy podle předchozího průběhu řešení úlohy.

Příklad - systém GPS

Metoda analýzy prostředků a cílů (systém GPS)
Používá procedury formulované na základě poznatků a psychologie lidského myšlení:

- TRANSFORM sestavuje a řeší (znovu) podúlohu, tj. převedení současného stavu do cílového,
- REDUCE vybírá pravidlo, které by odstranilo (zmenšilo) nejdůležitější diferenci mezi současným a cílovým stavem,
- APPLY aplikuje vybrané pravidlo na daný stav.

Cílový stav poslední vyřešené podúlohy se nachází na vrcholu zásobníku. Poslední vyřešená podúloha měla za úkol odstranit jistou diferenci (v té době nejzávažnější). Vedlejší efekty však mohly způsobit vznik ještě závažnější diference. Takto vzniklá, nyní nejzávažnější má být odstraněna korekční podúlohou sestavenou prostřednictvím TRANSFORM.

Vlastní řešení

- začíná pomocí REDUCE, která však nebere v úvahu vedlejší efekty vybraného pravidla.
- Po vybrání pravidla se aktivuje APPLY.
- Nejde–li pravidlo použít, je procedurou určeno proč a znovu použito REDUCE. Ta nyní hledá pravidlo, které by odstranilo diference bránící použití původně vybraného pravidla.

Proces se rekurzívně opakuje tak dlouho, až je buď

- možné celou sekvenci nalezených pravidel aplikovat (a tím odstranit diferenci pomocí TRANSFORM, tj. vyřešit danou úlohu) nebo
- se prokáže, že úloha není řešitelná.

> ***Poznámky***
>
> *Zásobník je struktura LIFO.*

Hlavní nevýhoda

Je nutno definovat diference a uspořádat je podle závažnosti a dodat návod, jaká pravidla vybírat pro odstranění diferencí.

Příklad - systém STRIPS

Má schopnost samostatně vybírat diference a pravidla. Sám zjišťuje diference mezi stavy, interpretuje je jako cílové a vybírá vhodná pravidla.

Stavy jsou formulovány jako množiny formulí predikátové logiky. Každý operátor je tvořen trojicí

$$\langle B_\varphi, D_\varphi, F_\varphi \rangle,$$

kde B_φ je podmínka aplikovatelnosti operátoru,

D_φ je množina klauzulí, které mají být ze stávající množiny vypuštěny,

F_φ je množina klauzulí, které mají být přidány.

Zadat je třeba pouze počáteční množinu platných formulí, cílovou formuli a množinu operátorů.

V případě

- triviální úlohy je cílová formule obsažena v počáteční množině formulí a úloha je vyřešena,
- netriviální úlohy jsou hledány diference mezi množinou platných formulí a cílových formulí a pak je hledán vhodný operátor k odstranění. Je–li nalezen, je jeho podmínka aplikovatelnosti považována za nový podcíl, vzhledem k němuž se celý postup opakuje.

Příklad - systém PLANNER

Explicitně využívá procedurální reprezentaci znalostí. Striktně odlišuje

- Databázi obsahující fakta, která se týkají konkrétní, právě řešené úlohy a jsou reprezentována deklarativně,
- Bázi znalostí obsahující obecné znalosti z problémové oblasti ve formě pravidel. Tato pravidla jsou vlastně procedury působící na databázi a představují procedurální reprezentaci znalostí.

Poznámky

- *PLANNER nebyl nikdy implementován, pouze zjednodušené verze PROPLER, MICRO-PLANNER. V jistém smyslu na něj navazuje programovací jazyk PROLOG.*
- *Systémy GPS, STRIPS, PLANNER nedosáhly praktického rozšíření, jsou teoretickým zdrojem prací v oblasti řešení úloh.*
- *Moderní systémy UI kladou daleko větší důraz na využívání vysoce speciálních znalostí. Do popředí vystupují mj. otázky efektivní reprezentace znalostí.*

5 Příklady algoritmů

5.1 Řešící stroj

5.1.1 Krok 1

```
unit uStateSpace_1;

// Hodláme realizovat implementovat obecný algoritmus
// informovaného prohledávání stavového prostoru
// K tomu bude třeba implementovat
// a) jeho stavy a
// b) metodu stavového prostoru search, která bude
// prohledávání provádět.

interface

type
  TState = class(TObject)
  end;

  TStateSpace = class(TObject)
    procedure Search(aStart, aTarget: TState);
  end;

implementation
{ TStateSpace }

procedure TStateSpace.Search(aStart, aTarget: TState);
begin

// 1.  Počáteční stav zapiš do seznamu OPEN, seznam CLOSED je
//     prázdný.
// 2.  Pokud je seznam OPEN prázdný, řešení neexistuje, ukonči
//     prohledávání.
// 3.  Ze seznamu OPEN vyber stav i s nejmenší hodnotou f(i).
//     V případì většího počtu stavů se stejnou minimální
//     hodnotou f(i) prověř, zda některý z těchto stavů není
//     stavem cílovým, v takovém případě jej vyber; jinak
//     vyber mezi stavy se stejnou minimální hodnotou f(i)
//     libovolně.
// 4.  Vymaž stav ze seznamu OPEN a zařaď jej do seznamu
//     CLOSED.
// 5.  Je-li stav i cílovým stavem, řešení je nalezeno, ukonči
//     prohledávání.
// 6.  Expanduj stav i; pro každého následníka j stavu i
//     vypočítej hodnotu f(j). Pokud stav j není ani v seznamu
//     OPEN, ani v seznamu CLOSED, zařaď jej do seznamu OPEN.
//     Pokud je stav j v seznamu OPEN nebo CLOSED, avšak s
//     ohodnocením větším než právě vypočtené f(j), změň jeho
//     ohodnocení na f(j), změň jméno rodičovského uzlu v
//     zápisu uzlu a zařaď ho do seznamu OPEN.
// 7.  Pokračuj krokem č.2.
end;
end.
```

5.1.2 Krok 2

```
unit uStateSpace_2;

// Propracujeme Search. To si vynutí zavedení
// procedur/fcí/metod, typů a polí vyznačeních červeně.
// Aby bylo možno v budoucnu jednoduše definovat konkrétní
// chování algoritmu, musí být metody TState.IsEqual a
// TStateSpace.Evaluate_f virtuální.

interface

type
  TValue = integer;

  TState = class;
  TStateSpace = class;

  TStateList = array of TState;

  TState = class(TObject)
  private
    FPrev: TState;
    FNext: TStateList;
    Ff: TValue;
  public
    function IsEqual(aState: TState): boolean; virtual;
      abstract;
  end;

  TStateSpace = class(TObject)
  private
    OPEN, CLOSED: TStateList;
    procedure Evaluate_f(aState: TState); virtual; abstract;
    procedure Expand(aState: TState);
  public
    procedure Search(aStart, aTarget: TState);
  end;

implementation

function IsEmpty(aStateList: TStateList): boolean;
begin
  result := false;
end;

procedure PutTo(aStateList: TStateList; aState: TState);
begin
end;

procedure RemoveFrom(aStateList: TStateList; aState: TState);
```

```
begin
end;

function GetFrom(aStateList: TStateList): TState;
begin
  result := nil;
end;

function IsIn(aStateList: TStateList; aState: TState): TState;
begin
  result := nil;
end;

{ TState }

procedure TStateSpace.Expand(aState: TState);
begin
end;

procedure TStateSpace.Search(aStart, aTarget: TState);
var
  i, j, j_inOPEN, j_inCLOSED: TState;
  indj: integer;
begin

// 1.  Počáteční stav zapiš do seznamu OPEN, seznam CLOSED je
//     prázdný.
  PutTo(OPEN, aStart);
  repeat
// 2.  Pokud je seznam OPEN prázdný, řešení neexistuje, ukonči
//     prohledávání.
    if IsEmpty(OPEN) then break;
// 3.  Ze seznamu OPEN vyber stav i s nejmenší hodnotou f(i).
//     V případì většího počtu stavů se stejnou minimální
//     hodnotou f(i) prověř, zda některý z těchto stavů není
//     stavem cílovým, v takovém případě jej vyber; jinak
//     vyber mezi stavy se stejnou minimální hodnotou f(i)
//     libovolně.
// 4a. Vymaž stav ze seznamu OPEN ...
    i := GetFrom(OPEN);
// 4b. ... a zařaď jej do seznamu CLOSED.
    PutTo(CLOSED, i);
// 5.  Je-li stav i cílovým stavem, řešení je nalezeno, ukonči
//     prohledávání.
    if i.IsEqual(aTarget) then break;
// 6a. Expanduj stav i;
    Expand(i);
// 6b.  pro každého následníka j stavu i
    for indj := 0 to Length(i.FNext) - 1 do
      begin
        j := i.FNext[indj];
```

```
// 6c.  vypočítej hodnotu f(j).
        Evaluate_f(j);
// 6d. Pokud stav j není ani v seznamu OPEN, ani v seznamu
//     CLOSED, zařaď jej do seznamu OPEN. Pokud je stav j v
//     seznamu OPEN nebo CLOSED, avšak s ohodnocením větším
//     než právě vypočtené f(j), změň jeho ohodnocení na f(j),
//     změň jméno rodičovského uzlu v zápisu uzlu a zařaď ho
//     do seznamu OPEN.
        j_inOPEN := IsIn(OPEN, j);
        j_inCLOSED := IsIn(CLOSED, j);
        if (j_inOPEN = nil) and (j_inCLOSED = nil) then
          PutTo(OPEN, j)
        else if (j_inOPEN <> nil) and (j_inOPEN.Ff > j.Ff)
           then
          begin
            j_inOPEN.Ff := j.Ff;
            j_inOPEN.FPrev := j.FPrev;
          end
        else if (j_inCLOSED <> nil) and (j_inCLOSED.Ff > j.Ff)
           then
          begin
            RemoveFrom(OPEN, j);
            PutTo(OPEN, j);
          end;
      end;
// 7.  Pokračuj krokem č.2.
  until true;

end;
end.
```

5.1.3 Krok 3

```
unit uStateSpace_3;

// Propracujeme Expand. To si vynutí zavedení dalších
// procedur/fcí/metod:
// Dále propracujeme procedury pro práci s dynamickými poli.

interface

type

  TValue = integer;

  TState = class;
  TStateSpace = class;

  TStateList = array of TState;

  TState = class(TObject)
  private
    FPrev: TState;
    FNext: TStateList;
    Ff: TValue;
  public
    function IsEqual(aState: TState): boolean; virtual;
      abstract;
  end;

  TStateSpace = class(TObject)
  private
    OPEN, CLOSED: TStateList;
    function RulesCount: integer; virtual; abstract;
    function Condition(aNo: integer; aState: TState): boolean;
      virtual; abstract;
    function Apply(aNo: integer; aState: TState): TState;
      virtual; abstract;
    procedure Evaluate_f(aState: TState); virtual; abstract;
    procedure Expand(aState: TState);
  public
    procedure Search(aStart, aTarget: TState);
  end;

implementation

function IsEmpty(aStateList: TStateList): boolean;
begin
  result := Length(aStateList) <> 0;
end;

procedure PutTo(var aStateList: TStateList; aState: TState);
```

```
var
  ind, l: integer;
begin
  for ind := 0 to Length(aStateList) - 1 do
    if aStateList[ind] = nil then
      begin
        aStateList[ind] := aState;
        exit;
      end;
  l := Length(aStateList);
  SetLength(aStateList, l + 1);
  aStateList[l] := aState;
end;

procedure RemoveFrom(aStateList: TStateList; aState: TState);
var
  ind: integer;
begin
  for ind := 0 to Length(aStateList) - 1 do
    if (aStateList[ind] <> nil) and
       aState.IsEqual(aStateList[ind]) then
      aStateList[ind] := nil;
end;

function GetFrom(aStateList: TStateList): TState;
var
  minim: TValue;
  ind, l: integer;
  minimlist: TStateList;
begin
  result := nil;
  if aStateList = nil then exit;

  minim := aStateList[0].Ff;
  for ind := 1 to Length(aStateList) - 1 do
    if (aStateList[ind] <> nil) and
       (minim > aStateList[ind].Ff) then
      minim := aStateList[ind].Ff;

  minimlist := nil;
  for ind := 1 to Length(aStateList) - 1 do
    if (aStateList[ind] <> nil) and
       (minim = aStateList[ind].Ff) then
      begin
        l := Length(minimlist);
        SetLength(minimlist, l + 1);
        minimlist[l] := aStateList[ind];
      end;

  result := minimlist[random(Length(minimlist))];
  SetLength(minimlist, 0);
```

```
end;

function IsIn(aStateList: TStateList; aState: TState): TState;
var
  ind: integer;
begin
  result := nil;
  for ind := 0 to Length(aStateList) - 1 do
    if (aStateList[ind] <> nil) and
       aState.IsEqual(aStateList[ind]) then
      begin
        result := aStateList[ind];
        break;
      end;
end;

{ TState }

procedure TStateSpace.Expand(aState: TState);
var
  No: integer;
  NewState: TState;
begin
  for No := 0 to RulesCount - 1 do
    if Condition(No, aState) then
      begin
        NewState := Apply(No, aState);
        NewState.FPrev := aState;
        PutTo(aState.FNext, NewState);
      end;
end;

procedure TStateSpace.Search(aStart, aTarget: TState);
var
  i, j, j_inOPEN, j_inCLOSED: TState;
  indj: integer;
begin

// 1.  Počáteční stav zapiš do seznamu OPEN, seznam CLOSED je
//     prázdný.
  PutTo(OPEN, aStart);
  repeat
// 2.  Pokud je seznam OPEN prázdný, řešení neexistuje, ukonči
//     prohledávání.
    if IsEmpty(OPEN) then break;
// 3.  Ze seznamu OPEN vyber stav i s nejmenší hodnotou f(i).
//     V případì většího počtu stavů se stejnou minimální
//     hodnotou f(i) prověř, zda některý z těchto stavů není
//     stavem cílovým, v takovém případě jej vyber; jinak
//     vyber mezi stavy se stejnou minimální hodnotou f(i)
//     libovolně.
```

```
// 4a. Vymaž stav ze seznamu OPEN ...
    i := GetFrom(OPEN);
// 4b. ... a zařaď jej do seznamu CLOSED.
    PutTo(CLOSED, i);
// 5.  Je-li stav i cílovým stavem, řešení je nalezeno, ukonči
//     prohledávání.
    if i.IsEqual(aTarget) then break;
// 6a. Expanduj stav i;
    Expand(i);
// 6b.  pro každého následníka j stavu i
    for indj := 0 to Length(i.FNext) - 1 do
      begin
        j := i.FNext[indj];
// 6c.  vypočítej hodnotu f(j).
        Evaluate_f(j);
// 6d. Pokud stav j není ani v seznamu OPEN, ani v seznamu
//     CLOSED, zařaď jej do seznamu OPEN. Pokud je stav j v
//     seznamu OPEN nebo CLOSED, avšak s ohodnocením větším
//     než právě vypočtené f(j), změň jeho ohodnocení na f(j),
//     změň jméno rodičovského uzlu v zápisu uzlu a zařaď ho
//     do seznamu OPEN.
        j_inOPEN := IsIn(OPEN, j);
        j_inCLOSED := IsIn(CLOSED, j);
        if (j_inOPEN = nil) and (j_inCLOSED = nil) then
          PutTo(OPEN, j)
        else if (j_inOPEN <> nil) and (j_inOPEN.Ff > j.Ff)
                then
          begin
            j_inOPEN.Ff := j.Ff;
            j_inOPEN.FPrev := j.FPrev;
          end
        else if (j_inCLOSED <> nil) and (j_inCLOSED.Ff > j.Ff)
                then
          begin
            RemoveFrom(OPEN, j);
            PutTo(OPEN, j);
          end;
      end;
// 7.  Pokračuj krokem č.2.
  until true;

end;

end.
```

5.1.4 Krok 4

```
unit uStateSpace_4;

interface

// Ověříme použitelnost implementace na ulohách "Lišák" a
// "Nádoby".
// Při té příležitosti upravíme viditelnost polí a metod.
// Třídy dokompletujeme vlastnostmi.

type

  TValue = integer;

  TState = class;
  TStateSpace = class;

  TStateList = array of TState;

  TState = class(TObject)
  private
    FPrev: TState;
    FNext: TStateList;
    Ff: TValue;
  protected
    function IsEqual(aState: TState): boolean; virtual;
      abstract;
  public
    property Prev: TState read FPrev;
    property Next: TStateList read FNext;
    property f: TValue read Ff write Ff;
  end;

  TStateSpace = class(TObject)
  private
    OPEN, CLOSED: TStateList;
  protected
    function RulesCount: integer; virtual; abstract;
    function Condition(aNo: integer; aState: TState): boolean;
      virtual; abstract;
    function Apply(aNo: integer; aState: TState): TState;
      virtual; abstract;
    procedure Evaluate_f(aState: TState); virtual; abstract;
    procedure Expand(aState: TState);
  public
    procedure Search(aStart, aTarget: TState);
  end;

implementation
```

```
function IsEmpty(aStateList: TStateList): boolean;
begin
  result := Length(aStateList) <> 0;
end;

procedure PutTo(var aStateList: TStateList; aState: TState);
var
  ind, l: integer;
begin
  for ind := 0 to Length(aStateList) - 1 do
    if aStateList[ind] = nil then
      begin
        aStateList[ind] := aState;
        exit;
      end;
  l := Length(aStateList);
  SetLength(aStateList, l + 1);
  aStateList[l] := aState;
end;

procedure RemoveFrom(aStateList: TStateList; aState: TState);
var
  ind: integer;
begin
  for ind := 0 to Length(aStateList) - 1 do
    if (aStateList[ind] <> nil) and
       aState.IsEqual(aStateList[ind]) then
      aStateList[ind] := nil;
end;

function GetFrom(aStateList: TStateList): TState;
var
  minim: TValue;
  ind, l: integer;
  minimlist: TStateList;
begin
  result := nil;
  if aStateList = nil then exit;

  minim := aStateList[0].Ff;
  for ind := 1 to Length(aStateList) - 1 do
    if (aStateList[ind] <> nil) and
       (minim > aStateList[ind].Ff) then
      minim := aStateList[ind].Ff;

  minimlist := nil;
  for ind := 1 to Length(aStateList) - 1 do
    if (aStateList[ind] <> nil) and
       (minim = aStateList[ind].Ff) then
      begin
        l := Length(minimlist);
```

```
        SetLength(minimlist, l + 1);
        minimlist[l] := aStateList[ind];
      end;

  result := minimlist[random(Length(minimlist))];
  SetLength(minimlist, 0);
end;

function IsIn(aStateList: TStateList; aState: TState): TState;
var
  ind: integer;
begin
  result := nil;
  for ind := 0 to Length(aStateList) - 1 do
    if (aStateList[ind] <> nil) and
       aState.IsEqual(aStateList[ind]) then
      begin
        result := aStateList[ind];
        break;
      end;
end;

{ TState }

procedure TStateSpace.Expand(aState: TState);
var
  No: integer;
  NewState: TState;
begin
  for No := 0 to RulesCount - 1 do
    if Condition(No, aState) then
      begin
        NewState := Apply(No, aState);
        NewState.FPrev := aState;
        PutTo(aState.FNext, NewState);
      end;
end;

procedure TStateSpace.Search(aStart, aTarget: TState);
var
  i, j, j_inOPEN, j_inCLOSED: TState;
  indj: integer;
begin

// 1.  Počáteční stav zapiš do seznamu OPEN, seznam CLOSED je
//     prázdný.
  PutTo(OPEN, aStart);
  repeat
// 2.  Pokud je seznam OPEN prázdný, řešení neexistuje, ukonči
//     prohledávání.
    if IsEmpty(OPEN) then break;
```

```
// 3.  Ze seznamu OPEN vyber stav i s nejmenší hodnotou f(i).
//     V případì většího počtu stavů se stejnou minimální
//     hodnotou f(i) prověř, zda některý z těchto stavů není
//     stavem cílovým, v takovém případě jej vyber; jinak
//     vyber mezi stavy se stejnou minimální hodnotou f(i)
//     libovolně.
// 4a. Vymaž stav ze seznamu OPEN ...
    i := GetFrom(OPEN);
// 4b. ... a zařaď jej do seznamu CLOSED.
    PutTo(CLOSED, i);
// 5.  Je-li stav i cílovým stavem, řešení je nalezeno, ukonči
//     prohledávání.
    if i.IsEqual(aTarget) then break;
// 6a. Expanduj stav i;
    Expand(i);
// 6b.  pro každého následníka j stavu i
    for indj := 0 to Length(i.FNext) - 1 do
      begin
        j := i.FNext[indj];
// 6c.  vypočítej hodnotu f(j).
        Evaluate_f(j);
// 6d. Pokud stav j není ani v seznamu OPEN, ani v seznamu
//     CLOSED, zařaď jej do seznamu OPEN. Pokud je stav j v
//     seznamu OPEN nebo CLOSED, avšak s ohodnocením větším
//     než právě vypočtené f(j), změň jeho ohodnocení na f(j),
//     změň jméno rodičovského uzlu v zápisu uzlu a zařaď ho
//     do seznamu OPEN.
        j_inOPEN := IsIn(OPEN, j);
        j_inCLOSED := IsIn(CLOSED, j);
        if (j_inOPEN = nil) and (j_inCLOSED = nil) then
          PutTo(OPEN, j)
        else if (j_inOPEN <> nil) and (j_inOPEN.Ff > j.Ff)
                then
          begin
            j_inOPEN.Ff := j.Ff;
            j_inOPEN.FPrev := j.FPrev;
          end
        else if (j_inCLOSED <> nil) and (j_inCLOSED.Ff > j.Ff)
                then
          begin
            RemoveFrom(OPEN, j);
            PutTo(OPEN, j);
          end;
      end;
// 7.  Pokračuj krokem č.2.
  until true;

end;

end.
```

5.2 Použití řešícího stroje

5.2.1 Příklad „Lišák"

```
unit DogFoxes;

interface

uses uStateSpace_4;

// Implementace úlohy "Lišák" "A star" algoritmem. V dědici
// třídy TState je zavedena konkrétní struktura stavu úlohy
// (hrací plocha) a v dědici TStateSpace jsou implementována
// 4 pravidla protenciálně použitelná při expanzi uzlu se
// stavem.
// K implementaci je využito přepsání původních abstraktních
// virtuálních metod metodami realizujícími danou úlohu.

type

  TDogFoxesState = class(TState)
  private
    g, h: TValue;
    Grid: array[0..2, 0..2] of integer;
  protected
    function IsEqual(aState: TState): boolean; override;
  end;

  TDogFoxes = class(TStateSpace)
  private
    Target: TDogFoxesState;
  protected
    function RulesCount: integer; override;
    function Condition(aNo: integer; aState: TState): boolean;
      override;
    function Apply(aNo: integer; aState: TState): TState;
      override;
    procedure Evaluate_f(aState: TState); override;
  end;

implementation

{ TDogFox }

function TDogFoxesState.IsEqual(aState: TState): boolean;
var
  i, j: integer;
begin
  result := (aState is TDogFoxesState);
  if not result then exit;
```

```
  result := true;
  for i := 0 to 2 do
    for j := 0 to 2 do
      if (Grid[i, j] <> (aState as TDogFoxesState).Grid[i, j])
         then
        begin
          result := false;
          break
        end;
end;

{ TDogFoxes }

function TDogFoxes.RulesCount: integer;
begin
  result := 4;
end;

function TDogFoxes.Condition(aNo: integer; aState: TState):
  boolean;
begin
  result := false;
  if not (aState is TDogFoxesState) then exit;

  with aState as TDogFoxesState do
    case aNo of
      0: result := not ((Grid[0, 0] = 0) or (Grid[0, 1] = 0)
           or (Grid[0, 2] = 0)); // ne horní
      1: result := not ((Grid[0, 2] = 0) or (Grid[1, 2] = 0)
           or (Grid[2, 2] = 0)); // ne pravý
      2: result := not ((Grid[2, 0] = 0) or (Grid[2, 1] = 0)
           or (Grid[2, 2] = 0)); // ne dolní
      3: result := not ((Grid[0, 0] = 0) or (Grid[1, 0] = 0)
           or (Grid[2, 0] = 0)); // ne levý
    end;
end;

function TDogFoxes.Apply(aNo: integer; aState: TState):
  TState;
var
  i_empty, j_empty: integer;
  i, j, ie, je: integer;
begin
  result := nil;
  if not (aState is TDogFoxesState) then exit;

  result := TDogFoxesState.Create;
  with aState as TDogFoxesState do
    begin
      ie := 2; je := 2;
      for i_empty := 0 to 2 do
```

```
        for j_empty := 0 to 2 do
          if Grid[i_empty, j_empty] = 0 then
            begin
              ie := i_empty;
              je := j_empty;
              break;
            end;
        (result as TDogFoxesState).Grid := Grid;
        i := ie;
        j := je;
        case aNo of
          0: dec(i);
          1: inc(j);
          2: inc(i);
          3: dec(j);
        end;
        (result as TDogFoxesState).Grid[ie, je] := Grid[i, j];
        (result as TDogFoxesState).Grid[i, j] := 0;
    end;
end;

procedure TDogFoxes.Evaluate_f(aState: TState);
var
  i, j: integer;
begin
  with aState as TDogFoxesState do
    begin
      g := 0;
      if Prev <> nil then
        g := (Prev as TDogFoxesState).g + 1;
      h := 0;
      for i := 0 to 2 do
        for j := 0 to 2 do
          if (Grid[i, j] <> 0) and (Target.Grid[i, j] <> 0)
             and (Grid[i, j] <> Target.Grid[i, j]) then
            inc(h);
      f := g + h;
    end;
end;

end.
```

5.2.2 Příklad „Džbány"

```
sunit Jugs;

interface

uses uStateSpace_4;

// Implementace úlohy "Džbány" metodou stejných cen. V dědici
// třídy TState je zavedena konkrétní struktura stavu úlohy
// (obsahy obou nádob) a v dědici TStateSpace je
// implementováno 6 pravidel protenciálně použitelných při
// expanzi uzlu se stavem.
// K implementaci je využito přepsání původních abstraktních
// virtuálních metod metodami realizujícími danou úlohu.

const

  A = 4;
  B = 3;

type

  TJugsState = class(TState)
  private
    FcA, FcB: integer;
  protected
    function IsEqual(aState: TState): boolean; override;
  public
    property cA: integer read FcA;
    property cB: integer read FcB;
  end;

  TJugs = class(TStateSpace)
  protected
    function RulesCount: integer; override;
    function Condition(aNo: integer; aState: TState): boolean;
      override;
    function Apply(aNo: integer; aState: TState): TState;
      override;
    procedure Evaluate_f(aState: TState); override;
  end;

implementation

uses Math;

{ TJugsState }

function TJugsState.IsEqual(aState: TState): boolean;
begin
```

```
  result := (aState is TJugsState) and
    (FcA = (aState as TJugsState).FcA) and
    (FcB = (aState as TJugsState).FcB);
end;

{ TJugs }

function TJugs.RulesCount: integer;
begin
  result := 6;
end;

function TJugs.Condition(aNo: integer; aState: TState):
  boolean;
begin
  result := false;
  if not (aState is TJugsState) then exit;

  with aState as TJugsState do
    case aNo of
      0: result := FcA > 0;
      1: result := FcB > 0;
      2: result := FcA < A;
      3: result := FcB < B;
      4: result := (FcA > 0) and (FcB < B);
      5: result := (FcA < A) and (FcB > 0);
    end;
end;

function TJugs.Apply(aNo: integer; aState: TState): TState;
var
  p: integer;
begin
  result := nil;
  if not (aState is TJugsState) then exit;

  result := TJugsState.Create;
  with aState as TJugsState do
    case aNo of
      0: (result as TJugsState).FcA := 0;
      1: (result as TJugsState).FcB := 0;
      2: (result as TJugsState).FcA := A;
      3: (result as TJugsState).FcB := B;
      4: begin
           p := min(FcA, B - FcB);
           (result as TJugsState).FcA := FcA - p;
           (result as TJugsState).FcB := FcB + p;
         end;
      5: begin
           p := min(A - FcA, FcB);
           (result as TJugsState).FcA := FcA + p;
```

```
            (result as TJugsState).FcB := FcB - p;
          end;
    end;
end;

procedure TJugs.Evaluate_f(aState: TState);
var
  lcA, lcB: integer;
begin
  aState.f := 0;
  if not (aState is TJugsState) then exit;

  lcA := 0;
  if aState.Prev <> nil then
    lcA := (aState.Prev as TJugsState).FcA;
  lcB := 0;
  if aState.Prev <> nil then
    lcB := (aState.Prev as TJugsState).FcB;

  aState.f := abs((aState as TJugsState).FcA - lcA) +
    abs((aState as TJugsState).FcB - lcB);
end;

end.
```