INFORMATIKA

Nejkratší cesta z bludiště

STANISLAV TRÁVNÍČEK

Přírodovědecká fakulta UP, Olomouc

Pokračujeme v našem seriálu programových etud ukázkou *průchodu stromem do šířky* na příkladu jedné z mnoha verzí bludiště. O použité metodě se začínající programátor může dočíst např. v [1] a byla použita i v článku [2] v našem časopise.

Bludiště je zadáno jedničkami (zdi) a nulami (chodby). Někde uvnitř (v některé nule) je výchozí místo jedince, který se má dostat z bludiště tak, aby jeho cesta byla *co nejkratší*; existuje-li více cest téže délky, stačí najít libovolnou z nich. (Myšlenka bludiště z nul a jedniček byla převzata z [3].) Na obr. 1 je příklad bludiště o rozměrech 8×13; jsou tu dva východy: (1, 3) a (7, 13). Program nám má popsat nejkratší cestu, případně sdělit, že cesta z bludiště neexistuje – např. z výchozího místa (3, 12). Tím současně potvrzujeme, že se po bludišti (ve schématu) lze pohybovat jen nahoru a dolů a doleva a doprava, nikoli šikmo.

```
1101111111111
1000100100011
1010101101101
1010000000101
1001101101101
1010001000011
1000100011000
1111111111111
```

Obr. 1

Ježto jde o problematiku poměrně známou, nebudeme provádět podrobnou programovou analýzu, jen budeme komentovat jednotlivé části programového zpracování. Nejprve tedy deklarace.

```
program Bludiste;

const
  MaxInd = 50;            {omezeni rozmeru matice bludiste}
  Tah: array [1..4] of    {popis moznych pohybu - tahu}
         record
           Ra, Sl: Integer
         end
         = ((Ra: 0; Sl: 1), (Ra: -1; Sl: 0),
            (Ra: 0; Sl: -1), (Ra: 1; Sl: 0));
type                      {charakteristika mista v bludisti}
  Tady = record
    Ra, Sl: Integer
  end;

var
  Blu: array [1..MaxInd, 1..MaxInd] of Integer; {bludiste}
  Fronta: array [1..MaxInd * MaxInd] of         {fronta}
            record
              Misto: Tady;
              Krok: Integer
            end;
  ZacFr, KonFr, Krok, Smer, M, N, I, J: Integer;
  VychoziBod, KudyVen, Tu1, Tu2: Tady;
  CestaNalezena: Boolean;
```

(Začátek a konec fronty, krok metody – „výška vlny“, směr tahů $1-4$, počet řádků matice – M a sloupců matice – N, pomocné proměnné I, J.)

Každé místo, kam dorazí „vlna“ budeme testovat, zda již leží na okraji bludiště, tj. zda je východem z bludiště. Přitom se nastaví hodnota *CestaNalezena* a zaznamenají se „souřadnice“ (tj. číslo řádku a sloupce) *KudyVen*. To zařídí následující procedura:

```
procedure TestKonce(Je: Tady);
begin
```

```
  if ((Je.Ra = 1) or (Je.Ra = M) or
    (Je.Sl = 1) or (Je.Sl = N)) then
  begin
    CestaNalezena := true;
    KudyVen := Je
  end
  else
    CestaNalezena := false
end;
```

Nějakým způsobem je třeba zařídit vstup matice bludiště do programu a nastavení výchozího bodu cesty z bludiště. Byla zvolena forma načtení textového souboru *Bludiste.dat* sestávajícího z M řádků, v každém z nichž je řetězec nul a jedniček o délce N. Pro snadnější orientaci jsme před požadavkem na zadání řádku a sloupce výchozího místa toto bludiště zobrazili.

```
procedure VstupBludiste;
var
  DatS: Text;
  C: Char;
begin
  assign(DatS, 'Bludiste.dat');
  reset(DatS);
  M := 0;
  repeat
    Inc(M);
    N := 0;
    repeat
      Inc(N);
      Read(DatS, C);
      Blu[M,N] := Ord(C) - Ord('0');
    until EoLn(DatS);
    ReadLn(DatS)
  until Eof(DatS);
  close(DatS);
  WriteLn;
  WriteLn('Cesta z bludiste');
  for I := 1 to M do
  begin
```

```
    for J := 1 to N do
      Write(Blu[I, J]);
    WriteLn
  end;
  Write('vychozi bod ra sl: ');
  ReadLn(VychoziBod.Ra, VychoziBod.Sl);
  WriteLn
end;
```

Jádrem programu je algoritmus průchodu stromem do šířky. Vzhledem k funkcím čísel 0 a 1 nastavíme do výchozího bodu bludiště (tj. do pole *Blu*) číslo 2 a výchozí bod zařadíme do fronty i s „výškou vlny“ 2. Pak v cyklu bereme a zkoumáme vždy místo ze začátku fronty a všechna místa, kam se lze jedním tahem dostat a která mají v matici nulovou hodnotu zařadíme do fronty, přičemž *Krok* zvětšíme o 1 vzhledem ke zkoumanému místu a tuto velikost kroku zaznamenáme do pole *Blu*. U každého takového místa se hned ptáme, zda to je východ z bludiště (viz proceduru *TestKonce*) a v kladném případě práci algoritmu ihned ukončíme. Všimněme si, že nám tím odpadá starost – při testování nového místa – zjišťovat, zda toto nové místo není mimo bludiště. Jiný konec nastane, když se fronta vyprázdní a cesta ven nalezena není – protože neexistuje.

```
procedure NajdiCestu;
begin
  Blu[VychoziBod.Ra, VychoziBod.Sl] := 2;
  ZacFr := 1;
  KonFr := 1;
  Fronta[1].Misto := VychoziBod;
  Fronta[1].Krok := 2;
  TestKonce(VychoziBod);
  while ((ZacFr <= KonFr) and (not CestaNalezena)) do
  begin
    Tu1 := Fronta[ZacFr].Misto;
    Krok := Fronta[ZacFr].Krok + 1;
    Inc(ZacFr);
    for Smer := 1 to 4 do
    begin
      Tu2.Ra := Tu1.Ra + Tah[Smer].Ra;
      Tu2.Sl := Tu1.Sl + Tah[Smer].Sl;
```

```
      if (Blu[Tu2.Ra, Tu2.Sl] = 0) then
      begin
        Blu[Tu2.Ra, Tu2.Sl] := Krok;
        Inc(KonFr);
        Fronta[KonFr].Misto := Tu2;
        Fronta[KonFr].Krok := Krok;
        TestKonce(Tu2)
      end
    end
  end
end;
```

Podívejme se (obr. 2), jaké hodnoty jsou v poli *Blu* např. po skončení nastavení velikosti vlny (hodnoty *Krok*) rovné 8. Vidíme, že již při kroku 9 se dosáhne východu (7, 13).

```
1101111111111
1000178145611
1018161131101
1017654323101
1001161131101
1010871545611
1000187611780
1111111111111
```

Obr. 2

Máme-li popsat cestu z místa *VychoziBod* do místa *KudyVen*, budeme ji nejprve rekonstruovat, tj. budeme počínaje *KudyVen* postupně ke každému místu hledat v okolí takové místo, ve kterém je hodnota v poli *Blu* (tedy hodnota výšky vlny) o 1 menší (je-li takových míst více, zvolíme libovolné z nich) – dokud nedojdeme k místu *VychoziBod* s hodnotou 2. Zjištěné souřadnice a hodnotu *Smer* ukládáme – k tomu nám může posloužit deklarovaná *Fronta*. Výslednou cestu pak dostaneme přečtením uložených údajů v obráceném pořadí. Vzhledem k tomu, že je uložen vždy i *Směr*, lze popis cesty doplnit i slovním doprovodem připraveným v konstantě *Kam*. Jelikož jsme při hledání cesty neuložili směr posledního tahu, musíme nyní kvůli prvnímu zpětnému kroku zajistit, že nebudeme prohledávat místa vně Bludiště; to zařídí funkce *JeVBlu*.

```
function JeVBlu(Je: Tady): Boolean;
begin
  if ((Je.Ra >= 1) and (Je.Ra <= M) and
    (Je.Sl >=1) and (Je.Sl <= N)) then
    JeVBlu := true
  else
    JeVBlu := false
end;

procedure PopisCestu;
type
  Tstr6 = string[6];
const
  Kam: array [1..4] of Tstr6
    = ('vlevo', 'dolu', 'vpravo', 'nahoru');
var
  I, DelkaCesty: Integer;
begin
  DelkaCesty := 0;
  Tu1 := KudyVen;
  while ((Tu1.Ra <> VychoziBod.Ra) or
    (Tu1.Sl <> VychoziBod.Sl)) do
  begin
    Smer := 1;
    repeat
      Tu2.Ra := Tu1.Ra + Tah[Smer].Ra;
      Tu2.Sl := Tu1.Sl + Tah[Smer].Sl;
      if (JeVBlu(Tu2) and
        (Blu[Tu2.Ra, Tu2.Sl] = Krok - 1)) then
      begin
        Dec(Krok);
        Inc(DelkaCesty);
        Fronta[DelkaCesty].Misto := Tu1;
        Fronta[DelkaCesty].Krok := Smer;
        Tu1 := Tu2;
        Smer := 0
      end
```

```
      else
        Inc(Smer)
    until Smer = 0
  end;
  WriteLn('Z (', VychoziBod.Ra, ',', VychoziBod.Sl, ')');
  for I := DelkaCesty downto 1 do
    WriteLn(DelkaCesty - I + 1, '. ', Kam[Fronta[I].Krok],
      ' do (',
  Fronta[I].Misto.Ra, ',', Fronta[I].Misto.Sl, ')')
end;
```

Samotný program už pak může být docela krátký:

```
begin {program}
  VstupBludiste;
  NajdiCestu;
  if CestaNalezena then
    PopisCestu
  else
    WriteLn('Cesta z bludiste neexistuje.');
  ReadLn
end.
```

Procedury, z nichž sestává program, mohou být ovšem řešeny i jinými způsoby nebo prostředky. Některé z nich uvádíme dále v následujícím návrhu cvičení:

- vytvořte i jiná bludiště a ověřte na nich chod programu;
- do popisu cesty přidejte část, která zařídí výstup bludiště na obrazovku s tím, že získaná cesta bude vyplněna hvězdičkami a místa zasažená vlnou tečkami;
- proveďte jiné zobrazení – na výchozí místo umístěte A a na další místa (podle výšky vlny) B, C, D, ...
- v daném programu se jistým způsobem preferuje cesta vpravo (proč?); opravte program tak, aby přednost měla cesta vlevo;
- využijte toho, že v okamžiku dosažení okraje bludiště známe souřadnice předchozího pole (i příslušný směr) a upravte proceduru *PopisCesty* tak, aby funkce *JeVBlu* nebyla potřebná;

– rozšiřte deklaraci fronty tak, aby se s každým místem ukládal i „předchůdce“ (včetně směru), upravte takto proceduru *NajdiCestu* a na základě toho zjednodušte proceduru *PopisCesty;*
– při hledání cesty zvolte jiný způsob: nepoužívejte frontu, ale pro danou hodnotu kroku (výšky vlny) prohledejte vždy celé pole *Blu;*
– upravte program pro případ, že bychom se v bludišti mohli pohybovat i šikmo.

Literatura

[1] *Töpfer, P.:* Algoritmy a programovací techniky. Prometheus, Praha 1995.
[2] *Töpfer, P.:* Prohledávání do šířky trochu jinak. MFI, roč. 11 (2001/02), č. 5.
[3] *Željko, J.:* Primjena BFS (Breadth first search) algoritma napronalaženje najkraceg izlaza iz lavirinta. Triangle (Matematički časopis za učenike i nastavnike osnovnih i srednjih škola), serija A, vol. 4 (2000), No 1.

Logika

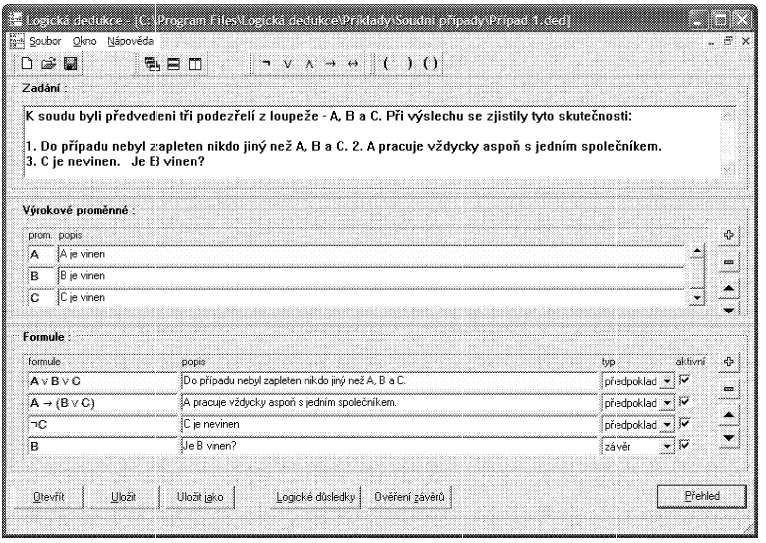
HASHIM HABIBALLA – LIBUŠE PAVLISKOVÁ – TIBOR KMEŤ

Přírodovědecká fakulta OU, Ostrava – Gymnázium Olgy Havlové, Ostrava-Poruba – Fakulta prírodných vied UKF, Nitra

(*Pokračování*)

2. Automatizovaná dedukce

Automatizace dedukce vyžaduje najít efektivní algoritmy pro generování důsledku nebo pro prověřování konsistence množin formulí. V praktických situací jde o automatizované zjišťování, zda z logických axiómů (vybrané tautologie) a speciálních axiomů (formalizované předpoklady) vyplývá závěr. V zásadě existují **metody sémantické a formální**. Tabulky z předchozí kapitoly jsou typickou sémantickou metodou. U sémantických metod musíme provádět interpretaci formulí, což je s ohledem na již zmiňované obrovské množství ohodnocení elementárních výroků velmi neefektivní přístup. I když některé sémantické metody vylepšují tuto nevýhodu, v zásadě je sémantický přístup pro automatizaci zcela nevhodný.



Obr. 1 Grafické rozhraní aplikace pro dedukci

Druhý formální (syntaktický) přístup se snaží se zcela oprostit od interpretace (smyslu) a používat prověřená pravidla pro práci se symboly (formulemi) bez ohledu na to, co znamenají. To je v principu velice efektivní, protože časová složitost pak nezávisí na možných interpretacích, ale na velikosti předpokladů jako symbolických formulí. Tyto metody vyžadují tedy jisté „know-how“, je složitější je pochopit, ale v konečném důsledku jsou zásadně lepší. Lze je rovněž naprogramovat a pak již mohou sloužit v aplikacích na „inteligentní“ usuzování. Právě pro složitost těchto metod (jsou dnes zcela v režii vysokoškolské výuky) je nebudeme všechny ani naznačovat. Existuje však velmi dobře použitelná a precizně zpracovaná práce autorky Libuše Pavliskové (diplomová práce na Ostravské Univerzitě [6]). Tato práce jednak obsahuje populární výklad problematiky dedukce a zejména je její součástí aplikace pro dedukci. Tato aplikace pracuje s výrokovou logikou (tudíž je dostatečně jednoduchá i pro středoškolskou výuku) a zároveň umožňuje zapsat libovolnou množinu předpokladů a buď generovat všechny možné důsledky nebo o konkrétním závěru zjistit, zda je to důsledek. Možná ještě významnější je pro výuku na středních školách existence velkého množství připravených příkladů s atraktivním zadáním jako jsou soudní případy podobné tomu z kapi-

toly 1. a další (samozřejmě i mnohem složitější). Didaktický text, popis aplikace i samotnou aplikaci pro operační systém Windows lze získat na adrese: **http://www1.osu.cz/home/habibal/dedukce/**

Formální metody dedukce lze dále v principu provádět dvěma způsoby – **přímo** a **nepřímo**. Zjednodušeně řečeno, při přímé metodě z předpokladů generujeme určitý důsledek pomocí odvozovacích (nebo jiných) pravidel. Při tomto způsobu tedy vždy hledáme potenciálně jiný důsledek podle toho, který závěr chceme dokázat. To v sobě samozřejmě skrývá jednu nevýhodu, jde především o možnost vygenerovat obrovské množství (potenciálně také nekonečné) různých důsledků a ten náš konkrétní se skrývá někde mezi nimi. To vede k neefektivitě a zejména se těžko hledají různé pomocné postupy, jak dedukci urychlit. Proto se používají spíše postupy nepřímé. Jsou podobné principu známého nepřímého důkazu. K předpokladům přidáme náš zkoumaný závěr, ovšem opačný (tedy se spojkou negace). Jelikož závěr vyplývá pokud je jeho interpretace pravda ve všech případech, kdy jsou pravdivé předpoklady, pak při opačném (negovaném) závěru taková množina nemůže být splnitelná. Při nepřímé metodě tedy s negovaným závěrem chceme dokázat nesplnitelnost. Tím se vyhneme základnímu problému přímé metody, protože náš cíl při dokazování je vždy stejný. Je jím prokázání nesplnitelnosti bez ohledu na dokazovaný závěr. To činí dedukci efektivnější. I přestože je stále obrovské množství možností, jak můžeme pomocí pravidel nakládat s předpoklady, můžeme již najít obecné postupy, jak urychlit (v určitých případech) proces hledání důkazu. Například v nepřímé rezoluční metodě (kterou si krátce vysvětlíme níže) jde o to, najít prázdnou formuli. Dá se tedy použít heuristika (metoda, která nefunguje zcela dokonale, ale v mnoha případech urychluje řešení), že je výhodnější pro následující krok hledání použít formule s co nejmenším počtem unikátních atomů (atomem se rozumí ve výrokové logice elementární výrok bez spojek). To samozřejmě není vždy pravda, ale intuitivně to je docela rozumné, pokud chceme dospět k formuli prázdné.

V tomto článku s omezeným rozsahem bychom se ještě chtěli zmínit o dvou formálních metodách. První trochu méně známá, ale i přesto velmi účinná je **metoda tablová**. Je založena na rozkladu formule ve stromu podle logických spojek a hledání větví, které obsahují navzájem negativní atomy (stejný atom s negací a bez negace). To ji činí velice účinnou, protože její časová náročnost je závislá jen na složitosti formule (počtu spojek). Bohužel v predikátové logice musí být obohacena o některé kroky, které její využití v praxi poněkud snižují. Druhou mnohem známější a široce pou-

živanou metodou je takzvaný **rezoluční princip** (rezoluce) resp. metody odvozené od něj. Myšlenku rezolučního principu formuloval v roce 1965 A. Robinson a od té doby byla velmi rozvinuta a zejména aplikována v různých systémech pro automatizované usuzování. I když existuje samozřejmě i její varianta pro logiku predikátovou, omezíme se zde pro jednoduchost pouze na logiku výrokovou. V klasickém pojetí rezoluce funguje pouze na formulích převedených do formy tzv. klauzulí (existují i zobecnění pro libovolné formule, ale zatím nejsou používány ani příliš prozkoumány). Klauzule je taková formule, kde se vyskytuje pouze binární spojka disjunkce, která spojuje jednotlivé atomy s negací nebo bez negace. Každou formuli lze pomocí logických pravidel (ekvivalencí – viz např. příklad 2) převést na ekvivalentní množinu klauzulí (může jich být i velmi mnoho). Rezoluční pravidlo pak umožňuje generovat ze dvou klauzulí obsahujících stejný atom (symbol elementárního výroku), který je v jednom případě s negací a v druhém bez negace, novou klauzuli spojenou disjunkcí obou výchozích klauzulí a zároveň vypustíme onen pár atomů, na kterých jsme prováděli rezoluci. Schématicky to lze znázornit následovně (atomy a jejich negace lze v klauzuli libovolně přesunovat bez změny interpretace dané formule).

Rezoluční pravidlo

$$\frac{C_1 \vee x \quad C_2 \vee \neg x}{C_1 \vee C_2} \tag{1}$$

C_1 a C_2 jsou zbývající části klauzulí a x je atom, na kterém rezoluci provádíme.

Máme-li pravidlo, jsme schopni z výchozích předpokladů (speciálních axiomů) a logických axiomů pomocí tohoto pravidla konstruovat sekvenci formulí, které říkáme důkaz. U každé syntaktické metody nebo formálního systému je důležité mít ověřeny dvě vlastnosti. Jde o to, aby metoda nebo systém, který obchází problematickou sémantiku tím, že pracuje pouze „slepě“ se symboly bez ohledu na jejich interpretaci, byl s touto sémantikou v souladu. První vlastnost, které se říká korektnost systému, zaručuje, že používaná pravidla generují pouze logické důsledky axiomů. Kdyby tomu tak nebylo, systém by z nekorektních pravidel generoval s předpoklady zcela nesouvisející formule (nesmyslné). Druhou vlastností je tzv. úplnost systému. Ta zaručuje, abychom pouze s danou množinou pravidel a axiomů byli schopni vygenerovat veškeré možné logické důsledky. Kdyby

tomu tak nebylo, měli bychom sice korektní systém, který však neumí některé důsledky generovat/ověřovat, což je jako univerzální algoritmus opět nefunkční.

Pokud je systém korektní a úplný, pak se chová zcela ekvivalentně sémantice a přesto ji nijak během dedukce nemusíme uvažovat.

Nyní rezoluci aplikujeme na již sémanticky řešený soudní případ.

Příklad 5:

Nejprve je nutné formule 1. $A \vee B \vee C$, 2. $A \rightarrow (B \vee C)$, 3. $\neg C$ převést do klauzulí. Formule 1. a 3. jsou klauzulemi bez převodu. Formule 2. vyžaduje převod. Využít můžeme pravidla pro přepis implikace na disjunkci. Toto pravidlo lze schématicky zapsat jako $A \rightarrow B \Leftrightarrow \neg A \ \vee B$ (čtenář si může lehce ověřit pomocí tabulky, že jde opravdu o formule s totožnou interpretací). Tímto pravidlem pak formuli 2. převedeme na formuli (disjunkce je navíc komutativní a asociativní): 2. $\neg A \ \vee B \vee C$. Nyní již můžeme buď přímo nebo nepřímo ověřovat závěry. Nejprve se pokusíme pomocí rezolučního pravdila (rezoluce) přímo vygenerovat, že B je vinen (B). Navíc přitom použijeme pomocná pravidla (například vyskytuje-li v klauzuli atom vícenásobně, je klauzule s jedním výskytem ekvivalentní; tyto kroky označíme pomocí $\Rightarrow$).

Dalším platným pravidlem je, že formule $\perp \vee F$ je ekvivalentní s F. Toto využijeme v případě, že jedna z premis obsahuje pouze jeden atom a tím pádem je po rezoluci ekvivaletní s $\perp$.

1. $A \vee B \vee C$ (axiom),
2. $\neg A \ \vee B \vee C$ (axiom),
3. $\neg C$ (axiom),
4. (rezoluce na A v 1. a 2.): $(B \vee C) \vee (B \vee C) \Rightarrow B \vee C$,
5. (rezoluce na C v 4. a 3. – z formule 3. nezbylo nic): B

Tím jsme provedli důkaz toho, že B je vinen (jelikož systém založený na rezoluci je korektní a úplný).

Nyní se stejný závěr pokusíme dokázat nepřímo. Přitom přidáme k předpokladům negaci závěru a budeme se snažit prokázat nesplnitelnost (to znamená vygenerovat prázdnou klauzuli, která je nesplnitelná).

1. $A \vee B \vee C$ (axiom),
2. $\neg A \ \vee B \vee C$ (axiom),
3. $\neg C$ (axiom),
4. $\neg B$ (negace závěru),

5. (rezoluce na B v 4. a 2. – z formule 4. nezbylo nic): $\neg A \vee C$,
6. (rezoluce na B v 4. a 1. – z formule 4. nezbylo nic): $A \vee C$,
7. (rezoluce na A v 5. a 6.): $C \vee C \Rightarrow C$,
8. (rezoluce na C v 7. a 3. – z formule 7. ani 3. nezbylo nic): $\perp$

Jelikož jsme dospěli k prázdné formuli, prokázali jsme nesplnitelnost množiny formulí 1. – 4., čímž je také prokázáno nepřímo, že B je logickým důsledkem množiny formulí 1. – 3.

Nepřímý důkaz v předchozím příkladě jsme samozřejmě mohli realizovat různými způsoby. Univerzálním přístupem je konstrukce nepřímého důkazu pomocí přímého přidáním jediného kroku, kdy provedeme rezoluci na vygenerovaný přímý důsledek a jeho negaci (pokud jde jen o atom). Na tom je vidět, že zřejmě existuje vždy mnoho způsobů, jak důkaz provést. Z hlediska časové složitosti jde principiálně o úlohu s exponenciální složitostí, o kterých jsme se již zmínili v předchozím článku v MFI. Nicméně existují přístupy, jak důkaz urychlovat a těm se říká rezoluční strategie. Některé pomáhají málo, ale jsou v principu úplné (tedy zachovávají úplnost systému) a některé jsou velmi efektivní za cenu neúplnosti (ale ve většině praktických úloh to nevadí). V některých formálních systémech je rezoluce nazývána jinak, resp. je její obecná myšlenka skryta v jiné symbolice. Například se můžete setkat s tzv. klauzulární logikou, kde se rezoluční pravidlo skrývá v tzv. pravidle řezu. Řez je výstižným pojmenováním, neboť jsme viděli, že rezoluce vlastně „vyřezává“ atomy z původních formulí.

V praxi se rezoluční metoda uplatňuje především v logickém programování. **Logické programování** není tak rozšířeno jako procedurální programování (např. algoritmizace v jazyce Pascal). Při procedurálním přístupu programátor musí vymyslet způsob, jak dosáhnout řešení cíle a to pomocí řízení výpočtu (musí správně použít proměnné, přiřazování, podmínky, cykly atd.). Logické programování vychází z myšlenky automatizace dedukce. Programátor nedefinuje postup řešení, ale pouze zadává formule (pravidla a fakta), která specifikují „logiku“ řešení úlohy. Na takto zapsaný program se pak může dotazovat, podobně jako při prověřování závěrů dedukce a systém sám odpoví na dotaz. Způsob řešení je univerzální a programátor se o něj nemusí starat. Jako jednoduchý příklad může sloužit výpočet faktoriálů. V procedurálním programování musí programátor vytvořit řízený výpočet, tj. musí naprogramovat cyklus nebo rekurzivní proceduru. V logickém programování stačí naprogramovat dvě pravidla (resp. jedno pravidlo a jeden fakt). Pravidlo udává hodnotu faktoriálu pro argument předchozího přirozeného čísla pomocí vazby na term s proměn-

nou (v logickém programování se nemá používat klasické přiřazování, měly by se používat výlučně prostředky logiky). Fakt udává hodnotu faktoriálu pro argument 0. V praxi by používání pouze logických prostředků způsobovalo neefektivní řešení, proto implementace logického programování často umožňují použití také některých omezených procedurálních prvků (např. řez). Asi nejznámějším prostředkem logického programování je jazyk PROLOG (PROgramming in LOGic). Vzhledem k omezenému rozsahu tohoto článku jej nebudeme rozebírat, ale čtenář má možnost využít elektronický učební text s mnohými příklady [2]. Pro vlastní pokusy doporučujeme získat z Internetu některou z freewarových implementací. Logické programování je výhodné především u úloh, kdy hledáme řešení v rozsáhlém stavovém prostoru a v úlohách typických v umělé inteligenci.

Literatura

[2] *Habiballa, H.:* Prolog. studijní text, Ostravská Univerzita, 2004, dostupné na: http://www.volny.cz/habiballa (odkaz Výuka → PROLOG)

[6] *Pavlisková, L.:* Principy dedukce ve výrokové logice – výukový program. diplomová práce, Ostravská Univerzita, 2003.

(*Pokračování*)

(*Dokončení ze s. 512*)

Autor uvádí velké množství zajímavých poznatků, některé mimo okruh odborníků méně známé, které jistě uvítají všichni zájemci o astronomii, dějiny přírodních věd, odborníci různých příbuzných oborů, ale i lidé zajímající se o obecné a kulturní dějiny. Velkou pomocí se může kniha stát učitelům všech typů škol, žákům a studentům jako významný podnět k dalšímu studiu, proto by neměla chybět v žádné učitelské a žákovské knihovně. Obsah knihy vhodně doplňuje 120 černobílých obrázků, chronologická tabulka významných astronomů, v textu pod čarou vysvětlující poznámky a odkazy na citované prameny včetně uvedení čísla stránek, v závěru ke každé kapitole seznam doporučené literatury k dalšímu studiu a jmenný rejstřík. Podobných přehledných publikací z dějin astronomie, čtivě a srozumitelně napsaných, máme u nás velmi málo, proto je velmi záslužné, že se F. Jáchim takové náročné práce ujal, a možno říci, že se jí zhostil velmi dobře. Pokud mohu vyslovit nějaké přání, doporučil bych autorovi připravit druhé vydání, v němž by se více než dosud zaměřil na astronomické objevy v 19. století a zejména pak ve 20. století, které významným způsobem rozšířily naše vědomosti a podstatně ovlivnily naše představy o stavbě a vývoji vesmíru. Rovněž bude třeba v příštím textu odstranit některé rušivé tiskové chyby, nepřesnosti a nedopatření, která se vždy v publikacích s velkým počtem údajů a dat objevují.

Knize popřejme zasloužený úspěch a mnoho vnímavých a pozorných čtenářů.

Rudolf Kolomý