KSI 2012/2013
Úloha 3-4: Taxonimie havěti

Jan Horáček
Gymnázium, Brno, Vídeňská 47; jan.horacek@seznam.cz

27. prosince 2012

1 Úvod

Tato úloha splňuje typické znaky grafové úlohy: její řešení je založeno na rekurzi a tím pádem ne zcela triviální na pochopení.

2 Program

K algoritmu slovně popsanému v tomto řešení přikládám zdrojový kód programu, který daný problém řeší. Je psán v prostředí Delphi 2009 a jeho cílem je vygenerovat všechny existující stromy s daným počtem vrcholů a pak na nich ověřit, zda-li jsou tyto stromy validními housenkami a/nebo humry.

Tento program je zde spíš jako bonus, mým původním cílem bylo ověřit, pro jaký maximalní počet vrcholů mohu jakýkoliv strom považovat za housenku a pro jaký za humra. Mimo to, že jsem se dostal na 3 628 800 stromů při 11 vrcholech a tím pádem i slušné paměťové náročnosti (program není napsán jako paměťově efektivní, ale časově), jsem metodou vyzkoušení všech kombinací zjistil, že pro méně, než 7 vrcholů můžeme rovnou považovat daný strom za housenku a pro počet vrcholů menší, než 10 můžeme strom automaticky považovat za humra.

Toto výrazně snižuje časovou náročnost algoritmu pro malý počet vrcholů.

Přímé vyloučení několika stromů ovšem neznamená, že by níže popsaný algoritmus nefungoval pro zadaný počet vrcholů (tedy min. 2).

3 Popis řešení

Řešení této úlohy je psáno pseudokódem založeném na jazyce Object pascal.

Řešení této úlohy je založeno na rozdělení celého problému na 2 části:

1. Nalezení páteře
2. Procházení páteře

Následuje kompetní slovní popis obou algoritmů:

3.1 Nalezení páteře

Následující funkce slouží k nalezení páteře. Pro lepší pochopení kontextu doporučuji shlédnout přiložený zdrojový kód. Pomůže Vám zejména v pohopení toho, jak je graf ukládán do paměti.

Návratovou hodnotou níže napsane funkce je *true*, pokud je housenka/humr validní, $false$, pokud je housenka/humr nevalidní.

Tato funkce má následující parametry:

1. *havet_type*

   Může být buď *const* $\_T\_HOUSENKA = 1$;, nebo *const* $\_T\_HUMR = 2$; Tento parametr určuje, jaký typ havěti má algoritmus hledat.

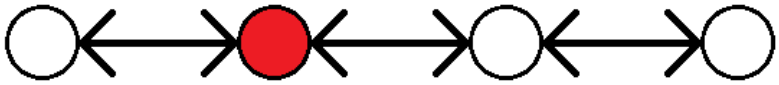
2. *actual*

   Je ukazatel na aktuální vrchol. Na začátku tuto funkci voláme s libovolným vrcholem daného stromu. Tento parametr je zapotřebí, protože je funkce rekurzivní.

   Pozn.: v přiložených obrázcích je tento vrchol značen červeně.

3. *sender*

   Nepovinný argument *sender* je využíván při rekurzivním volání této funkce. Při prvním volání, tedy z hlavní části programu, se tento parametr nezadává, tedy nabývá defaultní hodnoty *nil*. Tento parametr je využit jako prostředek k předejití zacyklení rekurze. Pro názornost potřeby tohoto parametru si zkuste představit, co by se stalo, kdybychom tento parametr neměli a funkci zavolali s následujícím stromem:



Obrázek 1: 4 vrcholy

```
function HledejPater(havet_typ:Byte; actual:PTVrchol;
 sender:PTVrchol = nil):boolean;
var i,more1cnt:Integer;
    more1:array [0..1] of PTVrchol;
begin
 Result := true;
 more1cnt := 0;
  //najdeme deti, ktere maji hloubku > 1 v pripade housenky
  //> 2 v pripade humra
 for i := 0 to actual^.Childs.Count-1 do
  begin
   if (CountChild(actual^.Childs.Data[i],actual) > havet_typ) then
    begin
     if (more1cnt >= 2) then
      begin
       writeln('ProjdiPater: Prilis mnoho pateri');
       Result := false;
       Exit;
      end;

     more1[more1cnt] := actual^.Childs.Data[i];
     more1cnt := more1cnt + 1;
    end;//if > 1
  end;//for i

 case (more1cnt) of
  0:Exit; //povazujeme za validni strom
  1:if (more1[0] <> sender) then
      Result := HledejPater(havet_typ, more1[0], actual);
      //pokud jen 1 smer ma > 1, tak postupujeme v tomto smeru
  2:Result := ProjdiPater(havet_typ, actual);
   //pokud 2 smery, jsme na pateri -> zavolame prochazeni patere
 end;//case
end;//procedure
```

Listing 1: Nalezení páteře

U této funkce bych se rád zastavil u jediné věci, která se může zdát nejasnou: case.

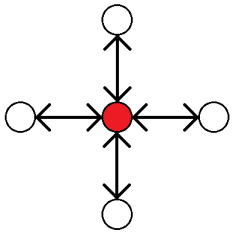
Tento case rozvětvuje počet nalezených vrcholů větších, než 1 v případě housenky a větších, než 2 v případě humra.

Pozn.: v přiložených obrázcích je vrchol, který má více, než 1 dítě, značen zeleně. Cesty k tomuto vrcholu jsou také značeny zeleně. Zeleně jsou značeny také všechny tyto děti.

Pozn.: přiložené obrázky jsou vztaženy k housence, v případě humra by musela být hloubka zanoření > 2.

1. **0**

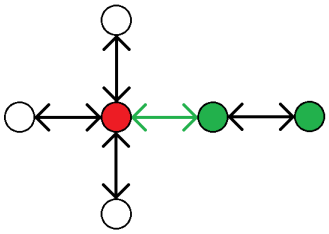
   Pokud neexistuje zádné dítě, po kterém následuje více dětí, strom je velmi malý. U takto malých stromů bylo experimentálně zjištěno, že je můžeme považovat za housenky a tudíž i za humry.



Obrázek 2: 0 dětí větší 1

2. **1**

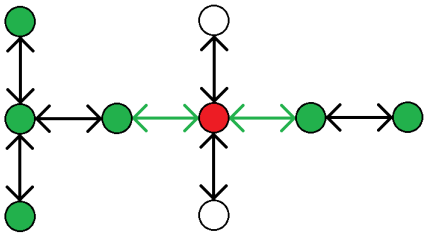
   Někde v našem stromu existuje páteř, ale my v ní aktuaálně nejsme. Tedy se rekurzivně zavoláme pro tento následující vrchol a pokusíme se tam najít hledanou páteř, která tam určitě je.



Obrázek 3: 1 dětí větší 1

3. **2**

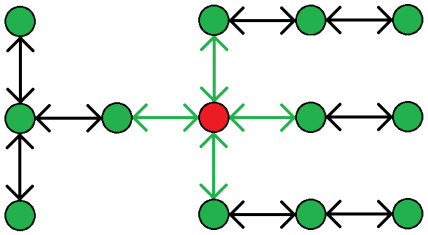
   2 znamená, že existují 2 směry, ve kterých je více dětí. Tato situace může nastat pouze, pokud jsme na páteři. Tedy usoudíme, že jsme na páteři a zavoláme funkci zajišťující procházení páteře.



Obrázek 4: 2 dětí větší 1

4. **>2**

   Tato možnost vpodstatě také existuje, jen se řeší výše v kódu a znamená, že existuje více páteří, což neodpovídá definici sotonožky ani humra, tudíž je funkce ukončena s návratovou hodnotou $false$.



Obrázek 5: 3 dětí větší 1

Výstupem této funkce je tedy proměnná boolean, která říká, zda-li je strom validní housenkou nebo humrem (podle zadaného parametru).

Tuto funkci bychom z programu volali následovně:

```
HledejPater(_T_HOUSENKA,libovolny vrchol);
HledejPater(_T_HUMR,libovolny vrchol);
```

Listing 2: Volání nalezení páteře

3.2 Počítání dětí

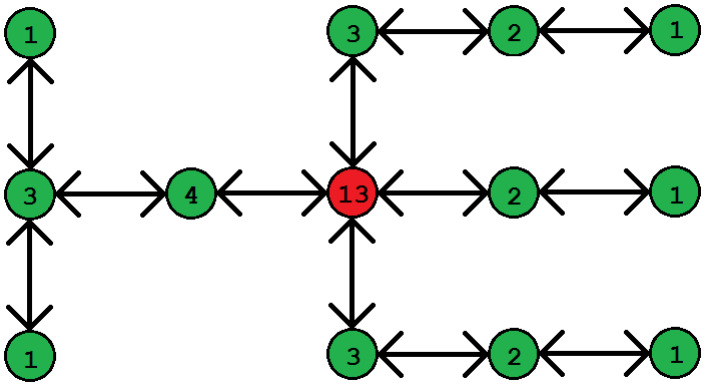
K předchozí funkci je potřeba ještě dodat funkci na počítání dětí, která vypadá následovně:

```
function CountChild(vrchol:PTVrchol;parent:PTVrchol = nil):Cardinal;
var i:Integer;
begin
 Result := 1;
 for i := 0 to vrchol^.Childs.Count-1 do
  if (parent <> vrchol^.Childs.Data[i]) then
    Result := Result + CountChild(vrchol^.Childs.Data[i],vrchol);
end;//function
```

Listing 3: Počítání dětí

Důležité zde je, že pokud má vrchol více dětí, jejich počty se sčítají.

Tuto funkci ilustruje např. tento obrázek:



Obrázek 6: počítání dětí

3.3 Procházení páteře

K procházení páteře slouží následující funkce, která na vstupu předpokládá:

1. *havet_type*

   Může být buď *const _T_HOUSENKA* = 1;, nebo *const _T_HUMR* = 2; Tento parametr určuje, jaký typ havěti má algoritmus hledat.

2. *actual*

   Je ukazatel na aktuální vrchol. Na začátku tuto funkci voláme s libovolným vrcholem daného stromu, který se nachází v páteři.

   Pozn.: v přiložených obrázcích je tento vrchol značen červeně.

3. *sender*

   Nepovinný argument *sender* je využíván při rekurzivním volání níže definované funkce a slouží jako ochrana proti zacyklení.

   Při externím volání funkce se nevyplňuje a tudíž nabývá hodnoty *nil*.

Tato funkce prochází celou páteř a kontroluje, zda-li jsou odbočky z páteře validní. Detailní rozbor funkce:

1. **Nalezení dětí, které mají více, než** *havet_type* **dětí**

   Protože pouze tyto vrcholy mohou být v páteři.

2. **Rozbor nalezených dětí**

   Představme si, že jsme uvnitř poměrně velké housenky. Tedy jsme v páteři. Jelikož jsme někde uprostřed, 1. cyklus for nalezl 2 děti, které mají další děti, protože existují 2 směry (jsme uprostřed páteře).

   2. cyklus for tedy pro každý směr rekurzivně zavolá tuto funkci. Samozřejmě nezapomene dodat 2. parametr *sender*, tedy ochranu proti zacyklení. Celá páteř se tedy projde jedním směrem. Pokud v tomto směru nastal problém, tj. $result = false$, funkce se ukončí a vrátí $false$, tedy se nejedná o validní housenku/humra. Pokud v 1. směru chyba nenastane, obdobně se zkontroluje 2. směr.

V zanořených funkcích dojde také ke snaze projít i směr zpět (tj. jít tam, odkud byla funkce zavolána), ovšem této snaze, která by způsobila *Stack overflow*, je zabráněno podmínkou $if(more1[i] <> sender)$.

Výstupem této funkce je tedy proměnná boolean, která říká, zda-li je strom validní housenkou nebo humrem (podle zadaného parametru).

```
function ProjdiPater(havet_typ:byte; actual:PTVrchol;
      sender:PTVrchol = nil):boolean;
var i,more1cnt:Integer;
    more1:array [0..1] of PTVrchol;
 begin
  Result := true;
  more1cnt := 0;

  //najdeme deti, ktere maji hloubku > 1 v pripade housenky
  //> 2 v pripade humra
   for i := 0 to actual^.Childs.Count-1 do
    begin
     if (CountChild(actual^.Childs.Data[i],actual) > havet_typ) then
      begin
       if (more1cnt >= 2) then
        begin
         writeln('Prilis mnoho pateri');
         Result := false;
         Exit;
        end;

       more1[more1cnt] := actual^.Childs.Data[i];
       more1cnt := more1cnt + 1;
      end;//if > 1
    end;//for j

  //a postupujeme dale az ke kraji patere
   for i := 0 to more1cnt-1 do
    begin
     if (more1[i] <> sender) then
      begin
       Result := ProjdiPater(havet_typ, more1[i], actual);
       //v pripade dvou smeru a jednoho nevalidniho ukoncime hned
       if (not result) then Exit;
      end;//if more1[i] <> sender
    end;//for i
 end;//procedure
```

Listing 4: Procházení páteře

3.4 Housenka, humr, nebo keřík

Tato funkce spojuje všechny předchozí a řeší úkol uložený v zadání úlohy:

```
procedure KSI();
var i:Integer;
    return:boolean;
begin
 for i := 0 to Stromy.Count-1 do
  begin
   return:=HledejPater(_T_HOUSENKA,Stromy.Data[i]^.Vrcholy.Data[0]);
   if (return) then
    begin
     writeln('STROM ',Integer(Stromy.Data[i]),' : housenka');
    end else begin
     return:=HledejPater(_T_HUMR,Stromy.Data[i]^.Vrcholy.Data[0]);
     if (return) then
       writeln('STROM ',Integer(Stromy.Data[i]),' : humr')
      else writeln('STROM ',Integer(Stromy.Data[i]),' : kerik');
    end;//else return
  end;//for i
end;//procedure
```

Listing 5: funkce KSI

4 Závěr

Na závěr bych rád poznamenal jednu zajímavost, kterou tento algoritmus vykazuje: tento algoritmus se snaží nalézt co nejmenší páteř, nejlépe ji vůbec nenalézt.

Paměťová náročnost: je zapotřebí uložit do paměti celý graf a pak pomocné proměnné, jako je například i, $more1cnt$, $more1$, které se ovšem vytváří při každém rekurzivním zanoření.

Časová náročnost se určuje poměrně složitě, protože je vysoce proměnná. Je zapotřebí nalézt 1. bod páteře a pak projít všechny body páteře. Faktem tedy je, že je úměrná počtu bodů páteře. Tento algoritmus, jak už bylo zmíněno, se snaží mít páteř co nejmenší, tedy co nejmenší časovou a paměťovou náročnost. Nezapomeňme poznamenat úvodní informace, tedy, že pro housenku s počtem vrcholů menším, než 7 a pro humra s počtem vrcholů menším, než 10 není potřeba nic procházet. V takovémto případě lze graf považovat za validní a časovou i paměťovou náročnost tedy za minimální.

Grafové úlohy nikdy nejsou triviální, alespoň né z mého pohledu. Tabule, která mi byla k dispozici, byla velmi vhodným nástrojem k řešení této úlohy.

Na závěr jedna demografická statistika zobrazující výskyt housenek a humrů mezi všemi existujícími stromy s daným počtem vrcholů:

vrcholů	**stromů**	**housenek**	**housenek %**	**humrů**	**humrů %**
7	720	684	95,00	720	100,00
8	5 040	4 262	84,56	5 040	100,00
9	40 320	28 598	70,93	40 320	100,00
10	362 880	204 962	56,48	357 040	98,39
11	3 628 800	1 559 440	42,97	3 437 160	94,72

Reference

Řešení bylo vytvořeno pouze autorovou vlastní silou - bez použití externích zdrojů.