Algoritmizace diskrétních struktur

Petr Kovář

Petr Kovář
Algoritmizace diskrétních struktur

Předmluva

Text, který právě čtete, vzniká jako součást přípravy přednášek předmětu *Diskrétní matematika a úvod do Teorie grafů (DiM)* pro bakalářské studium na technické vysoké škole. Vycházím z osnov předmětu, které byly navrženy pro studenty oborů se zaměřením na informatiku a další technické obory. Čerpal jsem ze skript *Diskrétní matematika* Petra Hliněného [4], z knihy *Discrete Mathematics and Its Applications* od Kennetha Rosena [9], z knihy *Kapitoly z diskrétní matematiky* od Jiřího Matouška a Jaroslava Nešetřila [8], z knihy *Introduction to Graph Theory* od Douglase B. Westa [10] i dalších učebnic a článků. Tento stručný dokument doplňuje důležité téma – jak v programovacích jazycích efektivně pracovat se strukturami zavedenými v první části předmětu. Snažil jsem se, aby text byl jednak přehledný a současně přesný a přitom čtivý a především aby obsahoval dostatek příkladů.

Poděkování patří Michalu Kravčenkovi a Matěji Krbečkovi, kteří text pečlivě prošli, upozornili na řadu nepřesností a pomohli vylepšit kvalitu výsledného textu. Pokud máte pocit, že v textu je přesto nějaká nesrovnalost, dejte mi vědět. Budu rád, když mne upozorníte i na méně srozumitelné pasáže, abych je v dalších verzích textu mohl vylepšit.

Text byl vysázen pomocí sázecího systému TEX ve formátu pdfLATEX.

V Ostravě 1. 11. 2016 Petr Kovář

Obsah

Kapitola 0

Úvodní část

Průvodce studiem

Rozkvět diskrétní matematiky jde ruku v ruce s rozvojem výpočetní techniky. V tomto stručném textu se budeme věnovat algoritmizaci diskrétních struktur. Nejprve však uvedeme stručný přehled pojmů, které budou využity v dalších kapitolách.

Cíle

Po prostudování celého textu kapitoly budete schopni:

- vysvětlit pojem složitosti algoritmu,
- číst algoritmy v pseudokódu,
- pracovat s pointery, jednorozměrnými poli a základními datovými typy.

0.1 Úvod

V tomto textu předpokládáme znalost základních datových struktur i základy programování. Dále předpokládáme, že čtenář zná a umí používat základní datové typy a jednoduché strukturované datové typy jako jsou uspořádané dvojice, jednorozměrná pole, dvou a více-rozměrná pole. Umí přistupovat k prvkům pole pomocí indexů i pracovat s prolinkovanými datovými strukturami. V této sekci přehledně shrneme datové typy a postupy, které se objeví v navazující části textu.

0.2 Datové struktury

Předpokládáme základní znalost práce s pointery (ukazateli).

0.3 Pseudokód

Předpokládáme, že čtenář je zběhlý ve čtení pseudokódů s cykly a rozhodovacími bloky, s rekurzivním voláním funkcí, předáváním parametrů a návratových hodnot.

Na několika místech využijeme ternární operátor ?:

0.4 Cykly

V algoritmech pracujeme jak s cykly řízenými podmínkou, tak s cykly s řídící proměnnou. Často využijeme direktivy `break` a `continue`.

0.5 Paměťová a výpočetní složitost

Klíčovou vlastností algoritmu je jejich paměťová a výpočetní složitost. Symbolem $O(g(n))$ rozumíme takovou množinu funkcí $f(n)$, pro které existují kladné konstanty c a n_0 tak, $\forall n > n_0$ platí $0 \leqq f(n) \leqq c \cdot g(n)$. Pomocí symbolu $O(g(n))$ budeme zapisovat složitost algoritmů.

Kapitola 1

Implementace diskrétních struktur

Průvodce studiem

V rámci prvního tématického celku o diskrétních strukturách si povíme něco o jejich implementaci do programovacích jazyků. Některé probírané struktury a postupy lze naprogramovat snadno, jiné obtížně. Často existuje řada přístupů, které se liší v nároku na paměť a/nebo strojový čas. Obecný přístup bývá na úkor rychlosti. Rychlejších kódů dosáhneme implementací datových typů nebo algoritmů připravenými na míru konkrétního problému, ale ne vždy musí být obecný přístup řádově *pomalejší.*

Tato kapitola je věnována vybraným implementacím.

Cíle

Po prostudování této kapitoly budete schopni:

- navrhnout vhodnou implementaci posloupností, zobrazení, relací a permutací,
- efektivně ověřit základní vlastnosti konkrétní relace,
- využít v algoritmech skládání zobrazení,
- vypsat permutaci pomocí cyklů.

1.1 Posloupnosti

Připomeňme, že posloupností rozumíme množinu čísel, která je uspořádaná: tj. uspořádání určuje první, druhý, třetí člen posloupnosti atd.

V analýze jsme *posloupnost* definovali jako funkci $\mathbb{N} \to \mathbb{R}$. Takto definovaná posloupnost má *nekonečně* mnoho členů. Každou konečnou posloupnost můžeme vždy přidáním nulových členů doplnit na nekonečnou posloupnost

Poznámka 1.1 (O indexování polí). V tomto textu jsme se rozhodli indexovat prvky polí od nuly. První prvek jednorozměrného pole `a` s n prvky je `a[0]`, poslední

prvek tohoto pole je `a[n-1]`. Velká část programovacích jazyků indexuje pole právě od nuly.

Věta 1.2. *(Konečnou) posloupnost* $(a_0, a_1, \ldots, a_{n-1})$ *implementujeme jako jednorozměrné pole* `a[ ]`*, kde* `a[i] =` a_i.

Příklad 1.3. Máme dánu (konečnou) posloupnost $(7, 5, 5, 7, 5, 6, 8)$. Uložíme ji do pole `p = [ 7 5 5 7 5 6 8 ]`. První prvek posloupnosti je `p[0]` s hodnotou 7, poslední (sedmý) prvek je `p[6]` s hodnotou 8.

S konečnými posloupnostmi obvykle neprovádíme složité manipulace, nalezení největšího a nejmenšího prvku zajistí následující klasický kód.

Algoritmus 1.1 (Nalezení indexu největšího a nejmenšího prvku).

```
// posloupnost je uložena v poli a[ ]
// počet prvků značíme n

if (n<1)
   return PRAZDNA_POSLOUPNOST;

max=min=0;                      // inicializace indexu extrémů
for (i=1; i<n; i++) {
   if (a[i] > a[max]) {
      max=i;                    // našli jsme větší
   } elseif (a[i] < a[min]) {
      min=i;                    // našli jsme menší
   }
}
// výsledek
výstup: 'nejmenší prvek je a[min], největší prvek je a[max]'
```

Složitost uvedeného algoritmu je $O(n)$, kde n udává počet prvků posloupnosti.

1.2 Zobrazení

Z kapitoly o relacích v Úvodu do diskrétní matematiky [7] už víme, že *zobrazení* $f : A \to B$ je speciálním případem relace mezi množinami A a B, kdy ke každému prvku a z množiny A existuje právě jeden prvek b z množiny B tak, aby dvojice (a, b) patřila do této relace. Proto můžeme psát

$$b = f(a).$$

Mějme konečné množiny $A = \{a_0, a_1 \ldots, a_{n-1}\}$ a $B = \{b_0, b_1, \ldots, b_{m-1}\}$. Zobrazení $f : A \to B$ implementujeme stejně jako posloupnost: jako jednorozměrné pole `f[ ]`, ve kterém `f[i]=j` vyjadřuje $f(a_i) = b_j$.

Všimněte si, že proměnná `i` nabývá hodnot indexu prvků v A, zatímco prvky `j` pole `f[ ]` jsou indexy prvků v množině B. Výhodou je, že taková implementace funkce nezávisí na konkrétním datovém typu prvků množin A, B. Nevýhodou je, že při výpisu prvků musíme „překládat“ prvky na jejich indexy, což je snadné a rychlé. Potřebujeme-li vypsat prvek s příslušným indexem, tak prvek snadno najdeme v poli strukturovaných proměnných či pointerů. Jestliže ale známe prvek a potřebujeme určit jeho index, musíme uložené prvky prohledávat, což je náročné na strojový čas, nebo využijeme datové typy jako hašovací tabulky.

Příklad 1.4. Máme dáno zobrazení $f : [0, 5] \to [0, 5]$, kde $f(0) = 4$, $f(1) = 5$, $f(2) = 3$, $f(3) = 3$, $f(4) = 2$, $f(5) = 2$.

Zobrazení uložíme do pole `f = [ 4 5 3 3 2 2 ]`.

Test zobrazení

Pozor: ne každé jednorozměrné pole nutně odpovídá nějaké funkci $f : A \to B$ ve smyslu, jak byla zavedena v předchozím odstavci. Je-li B množina s m prvky, tak prvky pole `f[ ]` musí být indexy z intervalu $[0, m - 1]$. To lze snadno ověřit algoritmem se složitostí $O(n)$, kde n je počet prvků pole `f[ ]`.

Algoritmus 1.2 (Test, zda v poli `f[ ]` je funkce).

```
for (i=0; i<n; i++)
   if (f[i]<0 || f[i]>m-1) {    // index mimo rozsah obrazů?
%      return NENÍ_FUNKCE;
   }
return JE_FUNKCE;
```

Příklad 1.5. Pole `g = [ 1 5 10 ]` neobsahuje funkci $g : [1, 3] \to [1, 10]$, protože $f(3) = 10$, ale 10 je index mimo rozsah indexů 0 až 9 pravé množiny $[1, 10]$.

Vlastnosti zobrazení

Zobrazení mohou být injektivní, surjektivní, nebo bijektivní. Máme-li zobrazení implementováno pomocí jednorozměrných polí obsahujících indexy, tak pomocí následujícího kódu můžeme ověřit, zda zobrazení $f : A \to B$ je injektivní. Předpokládáme, že $|A| = n$, $|B| = m$. V algoritmu počítáme, kolikrát je který prvek množiny B obrazem nějakého prvku z A. Oba následující algoritmy spoléhají na to, že v poli `f` jsou funkční hodnoty uloženy pomoci indexů prvků $b_1, b_2, \ldots, b_m$, tj. v poli jsou uložena čísla z intervalu $[0, m-1]$. Pokud by tomu uložená data neodpovídala, algoritmy nebudou fungovat správně.

Algoritmus 1.3 (Test, zda v poli `f[ ]` je injektivní funkce).

```
for (i=0; i<m; i++)  u[i] = 0;            // pomocné pole
for (i=0; i<n; i++)  ++u[f[i]];           // je použit
for (i=0; i<m; i++)
   if (u[i]>1) {
      return NENÍ_INJEKTIVNÍ_FUNKCE;
   }
return JE_INJEKTIVNÍ_FUNKCE;
```

Složitost uvedeného algoritmu je $O(n)$.

Surjektivita funkce se ověří podobně, stačí jen zkontrolovat, zda každý prvek množiny B je obrazem nějakého prvku z A.

Algoritmus 1.4 (Test, zda v poli `f[ ]` je surjektivní funkce).

```
for (i=0; i<m; i++)  u[i] = 0;            // pomocné pole
for (i=0; i<n; i++)  u[f[i]] = 1;         // je použit
for (i=0; i<m; i++)
   if (u[i]!=1) {
      return NENÍ_SURJEKTIVNÍ_FUNKCE;
   }
return JE_SURJEKTIVNÍ_FUNKCE;
```

Složitost uvedeného algoritmu je opět $O(n)$.

Protože injektivní zobrazení $f : A \to B$ je pro dvě množiny stejné velikosti současně surjektivní, tak pro ověření bijektivity stačí porovnat velikosti množin: $|A|$, $|B|$ a ověřit jen kteroukoliv z obou vlastností jedním ze dvou předchozích algoritmů.

1.3 Relace

Připomeňme, že *binární relace R na množině A* je podmnožina kartézské mocniny A^2, tj.

$$R \subseteq A \times A.$$

Pro *malé* konečné množiny $A = \{a_0, a_1, \ldots, a_{n-1}\}$ implementujeme obecnou relaci dvourozměrným polem `r[ ][ ]`, ve kterém je

$$\texttt{r[i][j]} = \begin{cases} \texttt{0} & \text{pokud } (a_i, a_j) \notin R \text{ a} \\ \texttt{1} & \text{pokud } (a_i, a_j) \in R. \end{cases}$$

Pro velké množiny s miliony prvků by se však taková matice nevešla do paměti. Konkrétní implementace je pak obvykle připravena na míru řešené úloze.

V dalším textu budeme opět pro jednoduchost předpokládat, že pracujeme s množinami malých celých čísel $[0, n]$, které odpovídají indexům prvků.

Příklad 1.6. Mějme relaci $R \subseteq [0,4]^2$, kde $(0,0) \in R$, $(0,4) \in R$, $(1,3) \in R$, $(2,2) \in R$ a $(4,0) \in R$.

Relaci R uložíme do dvojrozměrného pole, kde řádkový index dvojice (i,j) odpovídá prvnímu prvku a sloupcový index druhému prvku uspořádané dvojice.

$$\mathtt{R} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Vlastnosti relací

Předpokládejme, že máme binární relaci R na množině $\{0, 1, \ldots, n-1\}$ reprezentovanou dvourozměrným polem `r[ ][ ]`. Při ověřování, zda se jedná o relaci ekvivalence nebo relaci částečného uspořádání, musíme ověřit, zda relace má některé „pěkné“ vlastnosti: ověřuje se reflexivita, symetrie, antisymetrie a tranzitivita. Máme-li například v relační databázi uloženou informaci o vztazích mezi objekty, tak znázornit tuto strukturu pomocí hasseovského diagramu má smysl pouze v případě, že není porušena vlastnost antisymetrie a tranzitivity. Jinak diagram *nemůže* být správně a nemůže odpovídat údajům v databázi.

V dalším textu předpokládáme, že relace `r` na množině $[0, n-1]$ je uložena v dvojrozměrném poli `r[ ][ ]`.

Algoritmus 1.5 (Test, zda relace `r` je reflexivní).

```
for (i=0; i<n; i++)
   if (!r[i][i]) {    // jsou jedničky?
      return NENÍ_REFLEXIVNÍ_RELACE;
   }
return JE_REFLEXIVNÍ_RELACE;
```

Algoritmus má složitost $O(n)$. Je-li relace reflexivní, musíme tuto vlastnost ověřit pro *každý* prvek množiny $[0, n-1]$. Jestliže je vlastnost porušena byť u jediného prvku, můžeme prohledávání okamžitě ukončit.

Podobně můžeme ověřit, zda relace uložená v dvojrozměrném poli `r[ ][ ]` je symetrická.

Algoritmus 1.6 (Test, zda relace `r` je symetrická).

```
for (i=0; i<n; i++)
   for (j=i+1; j<n; j++)
      if (r[i][j]!=r[j][i]) {    // je symetrická matice?
      return NENÍ_SYMETRICKÁ_RELACE;
   }
return JE_SYMETRICKÁ_RELACE;
```

Protože vlastnost symetrie je nutno ověřit pro každou dvojici prvků, je složitost algoritmu $O(n^2)$. Ovšem v případě, že najdeme dvojici, pro kterou je `r[i][j]` $\neq$ `r[j][i]`, můžeme prohledávání okamžitě ukončit, neboť relace symetrická není.

Analogicky můžeme ověřit, zda některá dvojice prvků pole neporuší vlastnost antisymetrie.

Algoritmus 1.7 (Test, zda relace `r` je antisymetrická).

```
for (i=0; i<n; i++)
   for (j=i+1; j<n; j++)
      if (r[i][j]+r[j][i]==2) {    // je antisymetrická matice?
      return NENÍ_ANTISYMETRICKÁ_RELACE;
   }
return JE_ANTISYMETRICKÁ_RELACE;
```

Test, zda je relace uložená v poli `r[ ][ ]` tranzitivní, znamená ověřit

$$\forall i,j,k:\ r[i][j]\ \wedge\ r[j][k]\ \Rightarrow\ r[i][k].$$

Můžeme využít například následující kód.

Algoritmus 1.8 (Test, zda relace `r` tranzitivní).

```
for (i=0; i<n; i++)
   for (j=0; j<n; j++) {
      if (!r[i][j]) continue;          // může
      for (k=0; k<n; k++) {
         if (!r[j][k]) continue;       // může
         if (r[i][k]) continue;        // musí být!
         return NENÍ_TRANZITIVNÍ_RELACE;
      }
   }
return JE_TRANZITIVNÍ_RELACE;
```

Všimněte si, že vlastnost ověřujeme pro každou uspořádanou(!) trojici prvků, tj. pracujeme s každou trojicí indexů `i`, `j`, `k`. Proto složitost uvedeného algoritmu je $O(n^3)$ a pro obecnou relaci nelze počet testů zredukovat, neboť všechny trojice musí být prověřeny.

Pro úplnost přidáme algoritmus, který prověří, zda relace uložená v dvojrozměrném poli `r[ ][ ]` je úplná.

Algoritmus 1.9 (Test, zda relace `r` je úplná).

```
for (i=0; i<n; i++)
   for (j=i; j<n; j++)
      if (r[i][j]+r[j][i]==0) {    // je úplná matice?
         return NENÍ_ÚPLNÁ_RELACE;
      }
return JE_ÚPLNÁ_RELACE;
```

Uvedený algoritmus má složitost $O(n^2)$.
Všimněte si, že relace, která není reflexivní, nemůže být úplná.

1.4 Permutace

Připomeňme, že *permutace* konečné množiny $[0, n-1]$ jsme v Základech diskrétní matematiky [7] zavedli jako bijektivní zobrazení $p : [0, n-1] \to [0, n-1]$. A protože na straně 4 jsme zobrazení ukládali do jednorozměrného pole, budeme i permutaci p implementovat, jako jednorozměrné pole `p[ ]`.

Dříve, než budeme s permutacemi v nějakém algoritmu pracovat, měli bychom si být jisti, že data jsou ve správném formátu. Později uvedeme algoritmy pro skládání permutací, nebo zápis permutací pomocí cyklů, které spoléhají na to, že data uložená v poli `p[ ]` odpovídají definici permutace, tj. pole obsahuje právě čísla z množiny $[0, n-1]$ každé právě jednou. Jestliže by data v poli `p[ ]` neodpovídaly permutaci, uvedené algoritmy by nefungovaly správně.

Jak poznáme, že v poli je uložena permutace? Stačí ověřit zda je zobrazení uložené v poli `p[ ]` na (neboli surjektivní).

Algoritmus 1.10 (Test, zda `p[ ]` je permutace).

```
for (i=0; i<n; i++)  u[i] = 0;            // pomocné pole
for (i=0; i<n; i++)  u[p[i]] = 1;         // 1 = je použit
for (i=0; i<n; i++)
   if (u[i]!=1) {
      return NENÍ_PERMUTACE;
   }
return JE_PERMUTACE;
```

Všimněte si, že algoritmus má složitost $O(n)$. Uvedené tři cykly *nejsou* vnořené.

Příklad 1.7. Mějme permutaci $\pi = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 4 & 2 & 1 & 3 & 0 & 5 \end{pmatrix}$.

Permutaci π uložíme do pole `pi = [ 4 2 1 3 0 5 ]`

Nejčastěji diskutovanou operací bylo skládání permutací. Jestliže hledáme permutaci $r = q \circ p$ a premutace p a q máme uloženy v polích `p[ ]` a `q[ ]`, tak složenou permutaci r uložíme do jednorozměrného pole `r[ ]` například pomocí následujícího algoritmu.

Algoritmus 1.11 (Složení permutací `p[ ]` a `q[ ]` je permutace `r[ ]`).

```
for (i=0; i<n; i++)
     r[i] = q[p[i]];
```

Je-li v poli `p[ ]` uložena permutace, tak její zápis pomocí disjunktních cyklů získáme například kódem:

Algoritmus 1.12 (Výpis všech cyklů `n`-prvkové permutace `p[ ]`).

```
for (i=0; i<n; i++)  u[i] = 0;          // pomocné pole
for (i=0; i<n; i++) if (u[i]==0) {      // ještě nepoužit
   printf("(%d",i);  u[i] = 1;          // začátek cyklu
   for (j=p[i]; j!=i; j=p[j])  {        // další v cyklu
      printf(",%d",j);  u[j] = 1;
   }
   printf(")");                         // konec cyklu
}
```

Příklad 1.8. Mějme například permutaci

$$\pi = \begin{pmatrix} 0\,1\,2\,3\,4\,5 \\ 4\,2\,1\,3\,0\,5 \end{pmatrix}$$

uloženou v jednorozměrném poli `pi = [ 4 2 1 3 0 5 ]`. Předchozí algoritmus vypíše

```
(0,4)(1,2)(3)(5)
```

Kapitola 2

Implementace množin

Průvodce studiem

V této kapitole ukážeme několik implementací konečných množin. Ukážeme jejich přednosti a slabé stránky tak, aby jsme při implementaci konkrétního algoritmu uměli zvolit nejvhodnější variantu.

Cíle

Po prostudování této kapitoly budete schopni:

- implementovat konečné množiny několika způsoby,
- porovnat výhody a nevýhody jednotlivých implementací,
- zvolit vhodnou implementaci pro konkrétní úlohu.

2.1 Množiny

Množinou rozumíme soubor několika různých prvků. Množin mohou být konečné i nekonečné, zde se zaměříme na (malé) konečné množiny. Mezi nejběžnější operace, které s množinami provádíme, patří:

- zjištění, zda se daný prvek x nachází v množině A,
- přidání prvku x do množiny A,
- odebrání prvku x a množiny A.

Každá z těchto operací má jiné nároky na složitost nebo na paměť. Vhodná implementace zohlední, které operace budeme provádět často.

Množiny se implementují obtížně. Problémem je jednak fakt, že se prvky v množině nesmí opakovat a jednak neuspořádanost množin, neboť nemusí být stanoveno, kam v paměti který prvek uložit. Existuje několik klasických způsobů ukládání množin, které nyní uvedeme.

2.1.1 Charakteristická funkce podmnožiny

Jestliže známe celé *univerzum* $\mathcal{U} = \{u_1, u_2, \ldots, u_n\}$, ze kterého vybíráme prvky, a toto univerzum je relativně malé, tak podmnožinu $A \subseteq \mathcal{U}$ můžeme implementovat jako pole `a[ ]`, kde

$$\texttt{a[i]} = \begin{cases} 1 & \text{pro } u_i \in X \\ 0 & \text{jinak.} \end{cases}$$

Výhodou charakteristické funkce podmnožiny je, že se snadno hledají prvky množiny, sjednocení je funkcí OR, průnik funkcí AND.

Nevýhodou je, že charakteristické funkce podmnožiny lze sestavit jen pro poměrně malá univerza $\mathcal{U}$! každá podmnožina (i prázdná) totiž potřebuje alokovat nejméně $|\mathcal{U}|$ bitů.

2.1.2 Seznam prvků množiny

V dalším textu budeme předpkládat, že množina A má k prvků a množina B má l prvků. Množinu A můžeme také implementovat pomocí seznamu všech prvků. Seznam k prvků množiny A je uložen v poli `a[ ]` a můžeme psát

$$A = \{\texttt{a[1]}, \texttt{a[2]}, \ldots, \texttt{a[k]}\} \text{ pro pole } \texttt{a[ ]} \text{ délky } k.$$

Místo pole lze použít *dynamický spojový seznam prvků*, pak lze snadno přidávat a vynechávat prvky v seznamu.

Seznam prvků je vhodný i pro velká a předem neurčená univerza. Nevýhodou je, že vyhledání prvku množiny (jedna z nejčastěji požadovaných operací) je zdlouhavá – abychom ukázali, že prvek v množině není, musíme projít celý seznam.

Algoritmus 2.1 (Nachází se prvek `x` v množině uložené v `a[ ]`?).

```
for (i=0; i<k; i++)
   if (x==a[i]) {
      printf("x patří do a[ ]");  // nalezeno
      break;
   }
if (i<k)
   printf("x nepatří do a[ ]");   // nenalezeno
```

Složitost uvedeného jednoduchého algoritmu je $O(k)$.

Abychom do pole `c[ ]` uložili sjednocení množin uložených v polích `a[ ]` a `b[ ]`, můžeme využít následující kód.

Algoritmus 2.2 (Sjednocení dvou množin z `a[ ]`,`b[ ]` do pole `c[ ]`).

```
for (i=0; i<k; i++)
   c[i]=a[i];                  // z a[ ] všech k prvků
for (i=0,m=k; i<l; i++) {      // z b[ ] dalších až l prvků
   for (j=0; j<k; j++)
      if (b[i]==a[j]) break;   // je-li b[i] v a[ ]
   if (j<k) continue;          // přeskoč ho
   c[m++] = b[i];              // jinak ho přidej
}
```

Složitost uvedeného algoritmu je $O(kl)$.

Průnik množin uložených v polích `a[ ]` a `b[ ]`, můžeme určit následujícím algoritmem.

Algoritmus 2.3 (Průnik dvou množin z `a[ ]`,`b[ ]` do pole `c[ ]`).

```
for (i=0, m=0; i<k; i++) {
   for (j=0; j<l; j++)      // všechny dvojice z a[ ] a b[ ]
      if (a[i]==b[j])       // je společný?
         c[m++] = a[i];     // patří do průniku
}
```

Složitost uvedeného algoritmu je $O(kl)$.

2.1.3 Uspořádaný seznam prvků množiny

V jednorozměrných polí automaticky rozlišujeme první, druhý nebo poslední prvek, což neodpovídá neuspořádanosti množin. Nicméně při výčtu prvků množin obvykle intuitivně prvky seřazujeme podle nějakého klíče, například od nejmenšího po největší, což usnadní hledání prvků v množině. Toto uspořádání může využít i při uložení množiny do pole. Prvky v seznamu navíc (lineárně) uspořádáme podle předem daného klíče (dle velikosti v případě čísel, dle abecedy u jmen, a pod.)

Výhodné pak je, že při hledání prvků v množině můžeme použít *binární vyhledávání* prvků v poli metodou půlení intervalu v seznamu (viz příklad).

Algoritmus 2.4 (Binární vyhledání čísla x v *uspořádaném* poli `p[ ]`).

```
int a = 0; b = n-1;             // p[ ] má n prvků
while (a<b && p[a]!=k) {        // našli?
   c = floor((a+b)/2);
   if (p[c]<x)  a = c+1;        // ne, bude větší
   else         b = c;          // ne, bude menší
}
if (p[a]!=x)  printf("Číslo x není v seznamu.");
```

Pro určení, zda se prvek `x` v poli nachází, stačí jen $\lceil \log_2 n \rceil$ vyhledávacích kroků. Musíme však uvážit, že seřazení prvků pole podle nějakého klíče vyžaduje strojový čas, navíc při každé modifikaci pole je zpravidla nutné pole přeuspořádat.

Pokud by ale množina byla uložena v dynamickém spojovaném seznamu, tak binární vyhledávání použít nelze, protože pro každý prvek známe pouze předchozí a následující člen, ale neumíme bez procházení seznamu určit prostřední člen seznamu.

Algoritmus 2.5 (Sjednocení dvou seřazených množin `a[ ]`, `b[ ]`).

```
int i=0, j=0, k, l=0;
for (k=0; k<m+n; k++) {
   if (i>=m) {           // je-li a[ ] vyčerpáno (má m prvků)
      c[l++]=b[j++];
      continue;
   }
   if (j>=n) {           // je-li b[ ] vyčerpáno (má n prvků)
      c[l++]=a[i++];
      continue;
   }
   if (a[i]==b[j]) {     // průnik jen jednou
      j++;
      continue;
   }
   c[l++]=(a[i]<b[j]) ? a[i++] : b[j++];
}
```

Složitost uvedeného algoritmu je $O(m + n)$.

Analogicky je možno sestavit sjednocení dvou množin uložených v seřazených dynamických spojových seznamech.

Pro sestavení průniku dvou množin uložených v jednorozměrných polích lze využít následující kód.

Algoritmus 2.6 (Průnik dvou seřazených množin `a[ ]`, `b[ ]`).

```
int i=0, j=0, k, l=0;
for (k=0; k<m+n; k++) {
   if (i>=m) break;          // je-li a[ ] vyčerpáno (má m prvků)
   if (j>=n) break;          // je-li b[ ] vyčerpáno (má n prvků)
   if (a[i]<b[j]) {
      i++;                   // není v průniku
      continue;
   }
   if (a[i]= b[j])
      c[l++]=a[i++];         // je v průniku
   j++;
}
```

Složitost uvedeného algoritmu je $O(m + n)$.

Kapitola 3

Generování výběrů

Průvodce studiem

Je snadné spočítat, že ve sportce existuje $\binom{49}{6} = 13\ 983\ 816$ *možností, jak vylosovat šest čísel z 49. Jak sestavit algoritmus, který všechny tyto šestice systematicky vypíše, každou právě jedenkrát? V této kapitole ukážeme, jak vygenerovat (nebo projít) některé základní typy výběrů.*

Výhodou výpočetní techniky je její rychlost, která umožňuje prověřit velké množství případů v krátké době. V informatice mezi důležitou třídu úloh patří hledání řešení výrokové formule, která je sestavena z výroků spojených logickými operátory negace, konjunkce a disjunkce. Při hledání řešení hrubou silou musíme prověřit velké množství variací hodnot výrokových formulí. Ukážeme, jak tyto variace systematicky a efektivně probrat.

Cíle

Po prostudování této kapitoly budete schopni:

- efektivně vygenerovat základní typy výběrů s pevně zvoleným počtem prvků
- efektivně vygenerovat základní typy výběrů s libovolným počtem prvků

3.1 Výběry s předepsaným počtem prvků

Nejprve se budeme věnovat generování výběrů s předepsaným počtem prvků, tj. výběrům k prvků z nějaké n-prvkové množiny, přičemž k bude (malé) pevně zvolené číslo. Například při sestavování výběrů sportky nás mohou zajímat pouze vylosované šestice. Ukážeme algoritmy, které pomocí pevně předepsaného počtu vnořených cyklů proberou všechny různé výběry zvoleného typu:

- všechna různá zobrazení (variace s opakováním),
- všechna různá seřazení (variace bez opakování),

- všechny různé podmnožiny předepsané velikosti: kombinace bez opakování.

3.1.1 Jednoduché procházení dvojic (trojic, atd. . .)

Pro jednoduchost začneme pocházením uspořádaných a neuspořádaných dvojic: dvouprvkových variací a kombinací s i bez opakování.

Všechny uspořádané dvojice indexů (i, j) projdeme dvojicí vnořených cyklů s řídícími proměnnými `i,j`

Algoritmus 3.1 (Dvouprvkové variace s opakováním z n prvků).
```
for (i=0; i<n; i++)                // dvakrát vnořený cyklus
   for (j=0; j<n; j++) {
      // zpracujeme uspořádanou dvojici (i,j)
    }
```

Složitost uvedeného algoritmu je $O(n^2)$. Řádek „zpracujeme uspořádanou dvojici (i,j)“ nahradíme například výpisem nebo voláním funkce, která bude pro každou uspořádanou dvojici zavolána právě jednou.

Všechny uspořádané dvojice *různých* indexů (i, j) pro $i \neq j$ projdeme například následujícím jednoduchým kódem.

Algoritmus 3.2 (Dvouprvkové variace bez opakování z n prvků).
```
for (i=0; i<n; i++)                // dvakrát vnořený cyklus
   for (j=0; j<n; j++) {
      if (i<>j)
         // zpracujeme uspořádanou dvojici (i,j)
    }
```

Avšak pro tří- nebo čtyř-prvkové variace bez opakování by takový postup byl zbytečně komplikovaný, neboť bychom mnohokrát testovali, které hodnoty se opakují. Později ukážeme elegantnější řešení pro sestavení všech $n!$ permutací n-prvkové množiny.

Všechny neuspořádané dvojice indexů $\{i, j\}$ projdeme podobně. Všimneme si, že neuspořádané výběry můžeme vždy chápat jako výběr, kdy každý další prvek není menší, než prvek předchozí. Při procházení dvouprvkových kombinací je řídící proměnná `j` vnořeného cyklu vždy větší než řídící proměnná vnějšího cyklu `i`.

Algoritmus 3.3 (Dvouprvkové kombinace z n prvků).
```
for (i=0; i<n-1; i++)              // jen "nad diagonálou"
   for (j=i+1; j<n; j++) {
      // zpracujeme neuspořádanou dvojici {i,j}
   }
```

Řádek „zpracujeme neuspořádanou dvojici (i,j)“ opět nahradíme kódem, který bude pro každou neuspořádanou dvojici zavolán právě jednou.

Při procházení dvouprvkových kombinací s opakováním dovolíme, aby řídící proměnná `j` vnořeného cyklu byla rovna řídící proměnné vnějšího cyklu `i`.

Algoritmus 3.4 (Dvouprvkové kombinace s opakováním z n prvků).

```
for (i=0; i<n; i++)                // ne "pod diagonálou"
   for (j=i; j<n; j++) {
      // zpracujeme dvojici {i,j}
   }
```

Pro systematické procházení trojic potřebujeme tři vnořené cykly:

Algoritmus 3.5 (Tříprvkové variace s opakováním z n prvků).

```
for (i=0; i<n; i++)                // tři vnořené cykly
    for (j=0; j<n; j++)
       for (k=0; k<n; k++) {
       // zpracujeme uspořádanou trojici (i,j,k)
    }
```

Podobně pro k-prvkové variace nebo kombinace využijeme k vnořených cyklů.

Příklad 3.1. Pro sestavení všech možných šestic losovaných ve sportce můžeme využít následující kód. Všimněte si, že nejmenší tažené číslo nemůže být větší než 44, což jsme využili pro zrychlení kódu, který by jinak procházel i nepřípustné hodnoty prvních řídících proměnných, přičemž ke zpracování takových šestic by nedošlo, protože by nebyla splněna podmínka nejvíce vnořených cyklů.

Algoritmus 3.6 (Šestiprvkové kombinace z 49 prvků $1, 2, \ldots, 49$).

```
for (i=1; i<=44; i++)                // šest vnořených cyklů
    for (j=i+1; j<=45; j++)
        for (k=j+1; k<=46; k++)
            for (l=k+1; l<=47; l++)
                for (m=l+1; m<=48; m++)
                    for (n=m+1; n<=49; m++) {
                        // zpracujeme šestici {i,j,k,l,m,n}
    }
```

3.2 Permutace n prvků

Mějme množinu A s n prvky, které jsou uloženy do jednorozměrného pole `a[ ]`. Všech $n!$ permutací prvků množiny A můžeme systematicky projít pomocí následujícího rekurzivního algoritmu, který navrhl B. R. Heap v roce 1963.

Algoritmus 3.7 (Heapův algoritmus).
```
int i, a[ ];
permutace(n, a[ ]) {
   if (n==1)
      // zpracujeme permutaci v a[ ]
   else
      for (i=0; i < n-1; i++) {
         permutace(n-1, a[ ]);
         if (n je sudé)
            swap(a[i], a[n-1]);
         else
            swap(a[0], a[n-1]);
      }
      permutace(n-1, a[ ]);
}
```

Řádek „`// zpracujeme permutaci v a[ ]`“ můžeme nahradit kódem, který bude zavolán právě jednou pro každou z $n!$ různých permutací prvků, které budou postupně uloženy v poli `a[ ]`. Funkce `swap(x,y)` jednoduše prohodí obsah proměnných `x` a `y`.

Algoritmus 3.8 (Prohození obsahu proměnných).
```
swap(x,y) {
int temp;
   temp=x, x=y, y=temp;
}
```

Permutace s opakováním mají předepsaný počet opakování jednotlivých prvků. Jejich systematickému procházení se v tomto textu nevěnujeme.

3.3 Výběry s libovolným počtem prvků

Nevýhodou postupů popsaných v sekci 3.1 je, že *musíme dopředu znát velikost výběru*, tj. počet vnořených cyklů je pevně určen. Pro systematické procházení trojic nebo čtveřic potřebujeme jiný kód, než pro procházení dvoji atd. Následující kód takové omezení nemá. Místo k vnořených cyklů s pevným počtem opakování pracuje s řídící proměnnou `i`, jejíž hodnota se v průběhu algoritmu mění v závislosti na parametrech `n` a `k`.

3.3.1 Procházení všech k-prvkových variací

Víme, že všechny uspořádané k-tice z n prvků odpovídají všem různým n^k zobrazením k-prvkové do n-prvkové množiny. Pro názornost tak postupně uložíme každou

k-prvkovou variaci s oakováním do jednorozměrného pole `map[ ]` s `k` prvky vybranými z množiny prvků `0, 1, ...,n-1`.

$$\texttt{map} : \{0, 1, \ldots, k-1\} \ \rightarrow \ \{0, 1, \ldots, n-1\}$$

Každou k-prvkovou variaci s opakováním z n prvků jednoduše zpracujeme na řádku „`// zpracujeme zobrazení (map[0], ..., map[k-1])`“.

Algoritmus 3.9 (`k`-prvkové variace s opakováním z `n` prvků).

```
int i, map[k];
map[i=0]=-1;
while (i>=0) {
   if (++map[i]>=n)           // zvyš o 1
      { i--; continue; }
   if (++i<k)                 // 'nuluj' další prvky
      { map[i]=-1; continue; }
   // zpracujeme zobrazení (map[0], ..., map[k-1])
   i--;
}
```

Všimněte si, že nemusíme sestavovat k vnořených cyklů! Pro každou hodnotu vkládanou do pole testujeme:

- zda už překročila rozsah `n`, pak se vracíme na předchozí prvek pole,
- zda už byl zpracován poslední prvek pole (obraz `k`-tého prvku), jinak přistoupíme ke zpracování následujícího prvku pole.

3.3.2 Procházení všech k-prvkových variací bez opakování

Procházení všech k-prvkových variací (bez opakování) z n prvků odpovídá seřazení k různých prvků n-prvkové množiny. Pro názornost tak postupně uložíme každou k-prvkovou variaci bez oakování do jednorozměrného pole `arrange[ ]` s `k` prvky vybranými z množiny prvků `0, 1, ...,n-1`.

$$\texttt{arrange} : \{0, 1, \ldots, k-1\} \ \rightarrow \ \{0, 1, \ldots, n-1\}$$

Každou variaci zpracujeme na řádku „`// zpracujeme zobrazení (arrange[0], ..., arrange[k-1])`“.

Algoritmus 3.10 (k-prvkové variace bez opakování z n prvků).

```
int i, j, arrange[k];
arrange[i=0]=-1;
while (i>=0) {
   if (++arrange[i]>=n)       // zvyš o 1
      { i--; continue; }
   for (j=0; j<i; j++)        // opakuje se?
      if (arrange[i]==arrange[j])
         break;
   if (j<i)  continue;        // opakované vynech
   if (++i<k)                 // 'nuluj' další prvky
      { arrange[i]=-1; continue; }
   // zpracujeme variaci (arrange[0],...,arrange[k-1])
   i--;
}
```

Pro každou hodnotu vkládanou do pole testujeme:

- zda už překročila rozsah `n`, pak se vracíme na předchozí prvek pole,
- zda se prvek v poli už nevyskytuje, jinak výběr přeskočíme,
- zda už byl zařazen poslední prvek, jinak přistoupíme ke zpracování dalšího prvku pole.

3.3.3 Procházení všech k-prvkových kombinací (bez opakování) z n prvků

Všechny k-prvkové kombinace (bez opakování) z n prvků odpovídají výběru k-prvkové podmnožiny z n prvkové množiny. Je šikovné si uvědomit, že prvky výběru můžeme vždy jednoznačně seřadit, například od nejmenšího po největší. Budeme-li sestavovat výběry, kde každý další prvek je větší než předchozí, dostaneme právě hledané kombinace – každou kombinaci *právě jednou*. Nemusíme proto ani kontrolovat, zda se nově přidané prvky v poli již nevyskytly, protože víme, že každý prvek je *větší* než předchozí prvky. Kombinace uložíme do pole

`select[ ]`

s `k` prvky. Každou kombinaci pak zpracujeme na řádku „`//zpracujeme zobrazení (select[0], ..., select[k-1])`“.

Algoritmus 3.11 (k-prvkové kombinace (bez opakování) z n prvků).

```
int i, select[k];
select[i=0]=-1;
while (i>=0) {
   if (++select[i]>=n)            // zvyš o 1
     { i--; continue; }
   if (++i<k) {
     select[i]=select[i-1];     // jen seřazené!
     continue;
   }
   // zpracujeme kombinaci (select[0],...,select[k-1])
   i--;
}
```

Příklad 3.2. Pro sestavení všech možných šestic losovaných ve sportce můžeme využít také následující kód, který nepotřebuje šest vnořených cyklů.

Algoritmus 3.12 (Šestiprvkové kombinace z 49 prvků $1, 2, \ldots, 49$).

```
int n=49, k=6;
int i, select[k];
select[i=0]=-1;
while (i>=0) {
   if (++select[i]>=n)            // zvyš o 1
     { i--; continue; }
   if (++i<k) {
     select[i]=select[i-1];     // jen seřazené!
     continue;
   }
   // zpracujeme šestici (select[0],...,select[k-1])
   i--;
}
```

Kapitola 4

Generátory náhodných čísel

Průvodce studiem

Použití náhodných čísel v počítači považujeme za samozřejmé. Málokdo si však uvědomuje, že „náhodná“ čísla zpravidla ve skutečnosti náhodná nejsou. Vždyť po inicializaci náhodného generátoru na pevně zvolenou hodnotu, dostáváme vždy stejnou posloupnost hodnot! Přitom náhodná čísla jsou z bezpečnostního a praktického hlediska nezastupitelná. Následuje několik poznámek o generátorech náhodných čísel.

Cíle

Po prostudování této kapitoly budete schopni:

- vysvětlit význam náhodných čísel,
- vyjmenovat základní typy generátorů náhodných čísel.

4.1 Využití generátorů náhodných čísel

Generátorem náhodných čísel rozumíme náhodnou posloupnost bitů v počítači.

Kde všude potřebujeme používat náhodná čísla/bity? Například:

- Při vytváření náhodných (velkých) privátních klíčů (v SSL certifikátech). Použití náhodného hesla při šifrování SSL přenosu. Heslo (klíč) musí být skutečně náhodné, jinak by mohlo být uhodnuto nebo vypočteno!).
- Kolize paketů na Ethernetu řešena náhodně dlouhým čekáním před příštím vysíláním.
- Využití při *pravděpodobnostních algoritmech*, využívají náhodného promíchání vstupního souboru dat pro výrazné statistické urychlení běhu výpočtu.
- Při statistické analýze algoritmů a reálných procesů metodou Monte-Carlo, při modelování chaotických fyzikálních dějů, kdy místo prozkoumání všech

možných řešení volíme dostatečně velký soubor náhodných vzorků, kterými aproximujeme přesné řešení, atd.

4.2 Typy náhodných generátorů

Primitivní pseudonáhodné generátory

Využívají aritmetické vzorce typu

$$x := (A \cdot x + B) \mod C.$$

Hodnotu x opakovaně iterujeme a výsledné hodnoty (bity) x vytvářejí hledanou náhodnou posloupnost. Nevýhodou tohoto postupu je, že bity mají *velkou závislost* na předchozích iteracích. Jestliže je znám algoritmus výpočtu a máme dostatečně velký vzorek předchozích dat, tak hrozí *snadná předvídatelnost* dalších „náhodných“ čísel.

Pseudonáhodné generátory s vnějším vstupem

Tyto generátory využívají jednak podobné vzorce jako předchozí typ, ale navíc přidávají iniciační klíč – vstup nebo vstupy z *vnějších fyzických procesů* jako jsou prodlevy stisků kláves, zpoždění operací disku, statistiku síťových paketů, atd. Takový generátor může opět být problematický, neboť závisí na vnějších podmínkách, které teoreticky lze při znalosti algoritmu generování ovlivnit. Navíc s vyhodnocováním vnějších vstupů je spojena „velká cena“ každého bitu, což je nešikovné, pokud je náhodné číslo používáno často.

Hardwarové náhodné generátory

Založeny na různých *kvantových šumech* (třeba přechodové stavy polovodičů jako teplotní šum, rádiový šum, radioaktivní rozpad a pod.). Problematický je převod šumu na posloupnost uniformních bitů, kdy 1 má stejnou pravděpodobnost výskytu jako 0. Navíc princip je obvykle založen na důvěře v neurčitost popsanou zákony kvantové mechaniky. Nicméně kvantová mechanika je pouze teoretickým modelem, přičemž nelze vyloučit možnost, že jiná teorie přijde s deterministickým výkladem principu fyzikálních dějů.

Kapitola 5

Kombinatorická exploze

Průvodce studiem

Tato kapitola je věnována fenoménu kombinatorické exploze. Čtenáře seznámíme s tímto pojmem a upozorníme na související úskalí současné výpočetní techniky.

Cíle

Po prostudování této kapitoly budete schopni:

- vysvětlit pojem kombinatorické exploze.
- kriticky posoudit vhodnost řešení „hrubou silou“ procházením velkého počtu možností.

5.1 Co je kombinatorická exploze

Při programátorském řešení problémů spjatých s problém, při jejichž řešení se uplatní pojmy diskrétní matematiky, se často používají algoritmy typu:

Projdi všechny možnosti a podle nich rozhodni.

Bohužel tak často narazíme na fenomén zvaný *exponenciální kombinatorická exploze*, kdy *počet prohledávaných možností roste mnohem rychleji, že velikost vstupu.* Je-li růst exponenciální, tak i při zvětšení vstupu o velikost 1 se potřebný čas výpočtu zmnohonásobí.

Příklad 5.1. Počet různých permutací konečné množiny je dán vztahem $n!$. Jestliže se velikost vstupu zvětší o 1, tak počet prováděných operací bude n-násobný.

Příklad 5.2. Dáme-li na první políčko šachovnice 1 zrnko rýže, na druhé dvě, na třetí čtyři a na každé další dvojnásobek počtu zrnek na předchozím políčku, tak na celou šachovnici by bylo potřeba více zrnek rýže, než je její celosvětová produkce.

Stává se, že výpočet pro vstup o velikosti 10 zvládne stroj s procesorem 386, ale vstup o velikosti 15 *nelze spočítat* ani na nejvýkonnějších superpočítačích.

Mějte tento fenomén na paměti, až budete programovat procházení možností „hrubou silou“! Vhodnou volbou algoritmu, datové struktury či omezujících podmínek, které zohledňují konkrétní úlohu, můžeme dosáhnout výrazného zrychlení výpočtu.

5.2 Až budou lepší počítače?

Třebaže se podle Moorova zákona hustota tranzistorů každé dva roky přibližně zdvojnásobí, nemůžeme čekat, že jejich výkon poroste libovolně. Vždyť hmota je složena z atomů, které mají konečnou velikost a jakmile hustota tranzistorů dosáhne stavu, kdy velikost jednoho tranzistoru bude srovnatelná s velikostí atomu, už nebude možno hustotu zvyšovat.

Nemůžeme proto čekat, že úlohy, jejichž řešení je mimo dosah dnešní výpočetní techniky bude možné za pár let snadno vyřešit. Uvědomme si, že počet atomů na Zemi (v celém obejmu zeměkoule) se odhaduje na 10^{50}. Pokud bychom chtěli například projít nebo dokonce vypsat všechny možné relace dvacetiprvkové množiny, tak to nebude možné nikdy, neboť takových relací existuje 2^{20^2}, tj. více než $2 \cdot 10^{120}$. Zkrátka není možné relace vypsat, ani projít, protože jejich počet překračuje odhadovaný počet atomů ve Vesmíru!

Autorská práva

Při sestavování textu jsem se snažil důsledně dodržovat autorská práva. Většinu obrázků grafů jsem připravil sám v METAFONTU nebo v METAPOSTU. Oba programy jsou volně šiřitelné součásti řady distribucí operačního systému Linux.

Všechny fotografie, které jsem stáhnul z internetu (většinou z Wikipedie), buď mají na Wikipedii výslovně uvedeno, že jsou jejich autory volně dány k dispozici, nebo se jedná o reprodukce děl staších než 70 let. Novější fotografie budov, plaket a míst je možno použít ve smyslu § 33 odst. 1 autorského zákona, který říká „*Do práva autorského nezasahuje ten, kdo kresbou, malbou nebo grafikou, fotografií nebo filmem nebo jinak zaznamená nebo vyjádří dílo, které je trvale umístěno na náměstí, ulici, v parku, na veřejných cestách nebo na jiném veřejném prostranství; do autorského práva nezasahuje ani ten, kdo takto vyjádřené, zachycené nebo zaznamenané dílo dále užije. Je-li to možné, je nutno uvést jméno autora, nejde-li o dílo anonymní, nebo jméno osoby, pod jejímž jménem se dílo uvádí na veřejnost, a dále název díla a umístění.*“

Následuje seznam fotografií a obrázků i s uvedením zdrojů, ze kterých jsem čerpal. Seznam v době vzniku této pracovní verze textu není úplný.

Rejstřík

Literatura

[1] J. Černý, *Základní grafové algoritmy*, MFF UK, [online] (2010) [cit. 22.7.2012]. http://kam.mff.cuni.cz/~kuba/ka/

[2] J. Demel, Grafy, SNTL, Praha, (1989).

[3] D. Fronček, *Úvod do teorie grafů*, Slezská univerzita Opava, (1999), ISBN 80-7248-044-8.

[4] P. Hliněný, *Diskrétní matematika*, VŠB–TU Ostrava, skriptum (2006).

[5] P. Kovář, *Teorie grafů*, VŠB–TU Ostrava, skriptum (2012).

[6] P. Kovář, *Úvod do teorie grafů*, VŠB–TU Ostrava, skriptum (2012).

[7] M. Kubesa, *Základy diskrétní matematiky*, VŠB–TU Ostrava, skriptum (2012).

[8] J. Matoušek, J. Nešetřil, *Kapitoly z diskrétní matematiky*, Karolinum Praha, (2000), ISBN 80-246-0084-6.

[9] K.H. Rosen, *Discrete Mathematics and Its Applications – 6th ed.*, McGraw-Hill, New York NY, (2007), ISBN-10 0-07-288008-2.

[10] D.B. West, *Introduction to graph theory – 2nd ed.*, Prentice-Hall, Upper Saddle River NJ, (2001), ISBN 0-13-014400-2.

Přehled použitých symbolů

$\emptyset$	prázdná množina
$x \in A$	prvek x množiny A
$A \subseteq B$	podmnožina A množiny B
$A \subset B$	vlastní podmnožina A množiny B
$A \cap B$	průnik množin A a B
$A \cup B$	sjednocení množin A a B
$A \setminus B$	rozdíl množin A a B
$A \triangle B$	symetrická diference množin A a B
$A \times B$	kartézský součin množin A a B
$\simeq$	relace ekvivalence
$A \wedge B$	logická konjunkce („A a současně B“)
$A \vee B$	logická disjunkce („A nebo B“)
$A \Rightarrow B$	implikace („jestliže platí A, tak platí B“)
$A \Leftrightarrow B$	ekvivalence („A platí právě, když B“)