PŘÍRODOVĚDECKÁ FAKULTA UNIVERZITY PALACKÉHO
KATEDRA INFORMATIKY

BAKALÁŘSKÁ PRÁCE

Algoritmy pro vyhledávání textu

2010

Jan Rajtr

Místopřísežně prohlašuji, že jsem celou práci včetně příloh vypracoval samostatně.

15. května 2010 Jan Rajtr

Anotace

Tato bakalářská práce se skládá z teoretické a programátorské části. Teoretická část se zabývá zkoumáním principů, na kterých jsou založeny algoritmy vyhledávání v textu. Programátorskou část tvoří aplikace demonstrující činnost a účinnost těchto algoritmů.

Rád bych poděkoval RNDr. Arnoštu Večerkovi za vedení této bakalářské práce a za přínosné konzultace. Díky patří také rodině za podporu a trpělivost.

Obsah

Seznam obrázků

1. Úvod

Vyhledávání textů je velmi důležitý subjekt v široké oblasti zpracování textu. Algoritmy pro vyhledávání jsou základní komponenty užité v implementacích praktických programů pro většinu operačních systémů. Navíc zdůrazňují programovací metody, které slouží jako paradigma v jiných polích informatiky, zejména návrhu systémů a softwaru. V neposlední řadě také hrají důležitou roli v teoretické informatice tím, že poskytují zajímavé problémy a výzvy k řešení.

I když lze data ukládat různými způsoby, text zůstává hlavní formou výměny informací. Toto se netýka jen literatury nebo lingvistiky, kde jsou data složena z mohutných spisů a slovníků. Platí to také pro informatiku, kde jsou velká množství dat uložena v lineárních souborech. A je to také případ molekulární biologie, protože biologické molekuly mohou být často přibližně popsány jako sekvence nukleotidů nebo aminokyselin. Množství dat v těchto oborech se zdvojnásobuje každým rokem. To je důvod pro to, aby algoritmy byly účinné, i když uvážíme, že rychlost a velikost dostupné paměti v počítačích se pravidelně zvětšují.

1.1. Požadavky na práci

Cílem práce je demonstrovat činnost algoritmů vyhledávání v textu a zjistit, jaká je jejich účinnost. Práce by měla obsahovat:

- Popis principů, na kterých jsou založeny algoritmy vyhledávání v textu.
- Implementaci jednotlivých algoritmů vyhledávání (Knuth-Morris-Pratt, Boyer-Moore a další).
- Aplikaci, která ukazuje činnost jednotlivých algoritmů.
- Srovnání, jak jsou tyto algoritmy účinné.

1.2. Použití algoritmů

Tyto algoritmy jsou používány v situacích, kdy hledáme řetězec v předem neznámém textu. Nebereme do úvahy možnost, že by text, ve kterém hledáme, byl dopředu indexován, či jinak zpracován. Předpokládáme také, že máme k dispozici spolehlivou funkci k porovnání znaků.

Typické použití je v textových editorech, kde je často používaným algoritmem Boyer-Moore.

Algoritmy jsou pojmenovány podle odborníků, kteří je vyvinuli.

2. Základní pojmy

Vyhledávání řetězců sestává z nalezení jednoho, nebo obecněji všech výskytů řetězce, kterému říkáme `vzor`, v řetězci, který nazýváme `text`.

2.1. Pojmy

Délku textu označujeme `n`, délku vzoru `m`.

Oba řetězce jsou sestaveny z konečné množiny znaků zvané abeceda.

Slovo `u` je předložka (prefix, předpona) slova `w` pokud existuje slovo `v` (i prázdné) takové, že `w = uv`.

Slovo `v` je přípona (suffix, koncovka) slova `w` pokud existuje slovo `u` (i prázdné) takové, že `w = uv`.

Slovo `z` je podřetězec (substring, podslovo nebo faktor) slova `w` pokud existují dvě slova `u` a `v` (i prázdná) taková, že `w = uzv`.

`Okno` je podřetězec textu délky `m`, a jedná se o tu část textu, ve které probíhá aktuální porovnávání se vzorem (v programu Vizualizace je okno barevně odlišeno).

`Pokus` je vlastní akt porovnávání okna se vzorem. Falešný pokus je takový, kdy dojde ke shodě u několika prvního znaku a případně u několika dalších znaků, ale vzor se s celým oknem neshoduje.

2.2. Princip vyhledávání

Algoritmy popsané dále v textu prochází text s pomocí okna, a hledají okno shodné se vzorem.

Nejdříve zarovnají konce okna a vzoru, poté porovnají znaky v okně se znakem vzoru, při neshodě posunou okno vpravo. Opakují tuto proceduru, dokud se pravý okraj okna nedostane za pravý konec textu. Tento mechanismus se nazývá mechanismus posouvání okna.

Pokud vzor není v textu nalezen, vrací algoritmus hodnotu -1, v případě nalezení vrací algoritmus pozici okna shodného se vzorem.

3. Popis jednotlivých algoritmů

Kódy všech algoritmů v jazyce C# lze nalézt v Programátorské dokumentaci od kapitoly 6.4 dále.

3.1. Algoritmus hrubé síly

3.1.1. Princip

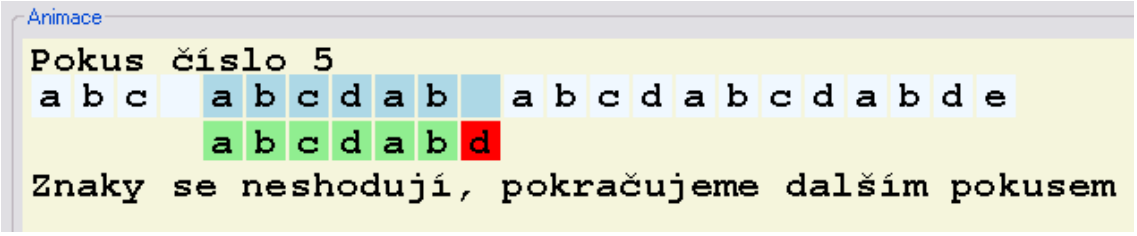
Algoritmus hrubé síly v jednom cyklu posouvá okno zleva doprava a v druhém (vnitřním) cyklu porovnává znaky v okně se vzorem, také zleva doprava.

Při výskytu neshody znaků končí vnitřní cyklus, okno se posune o jeden znak doprava a znovu začíná porovnávání znaků od prvního znaku v okně, algoritmus tedy některé znaky může porovnávat vícekrát, což je neefektivní.

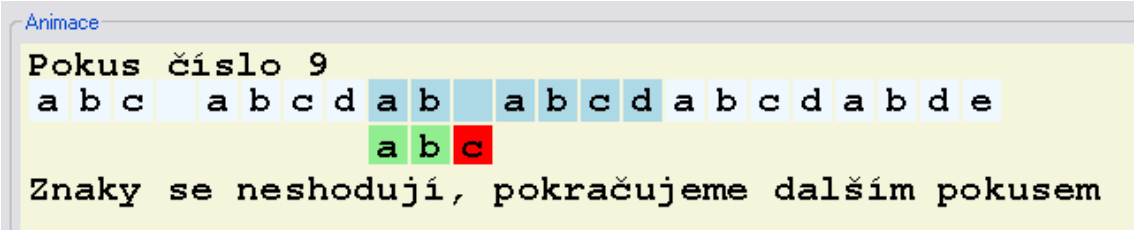
3.1.2. Časová složitost

Časová složitost algoritmu: vzor může začínat na `n-m+1` pozicích a s délkou vzoru `m` máme v nejhorším případě kvadratickou časovou složitost O(nm).

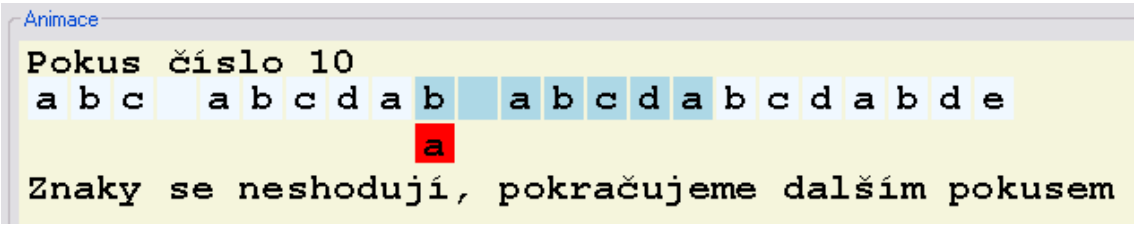
3.1.3. Příklad



Obrázek 1. Algoritmus hrubé síly – příklad, pokus č.5



Obrázek 2. Algoritmus hrubé síly – příklad, pokus č.9



Obrázek 3. Algoritmus hrubé síly – příklad, pokus č.10

V tomto případě byl znak `b`, kterým začíná okno pokusu číslo deset, porovnán třikrát, a z toho dvakrát při falešných pokusech číslo pět a devět.

3.2. Algoritmus Knuth-Morris-Pratt

3.2.1. Princip

Algoritmus Knuth-Morris-Pratt (dále jen KMP) je založen na myšlence, že v případě výskytu neshody při porovnávání znaků je možné posunout okno o více než jeden znak doprava, což vede k urychlení průběhu algoritmu.

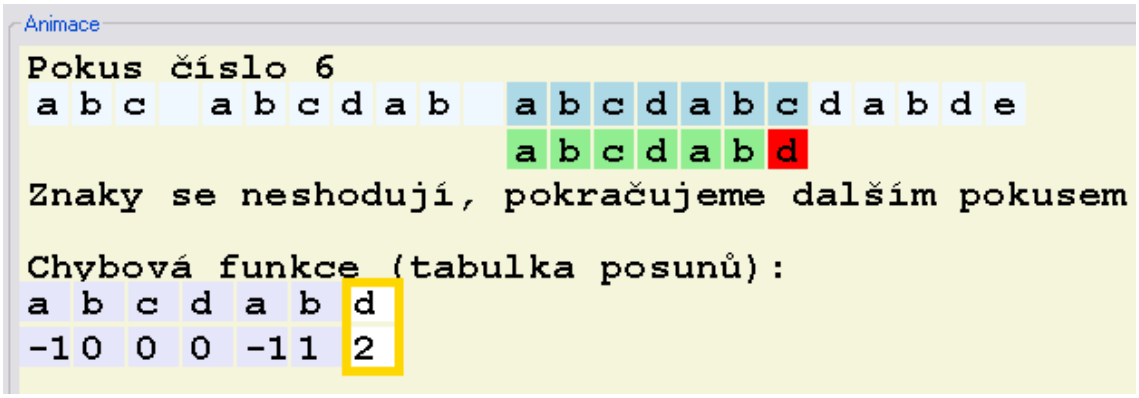
Princip spočívá v tom, že před začátkem vlastního vyhledávání nejdříve zpracujeme vzor tak, že při výskytu neshody můžeme posunout okno o více znaků, aniž bychom minuli možnost výskytu.

Významný fakt je ten, že zpracováváme vzor, ne text.

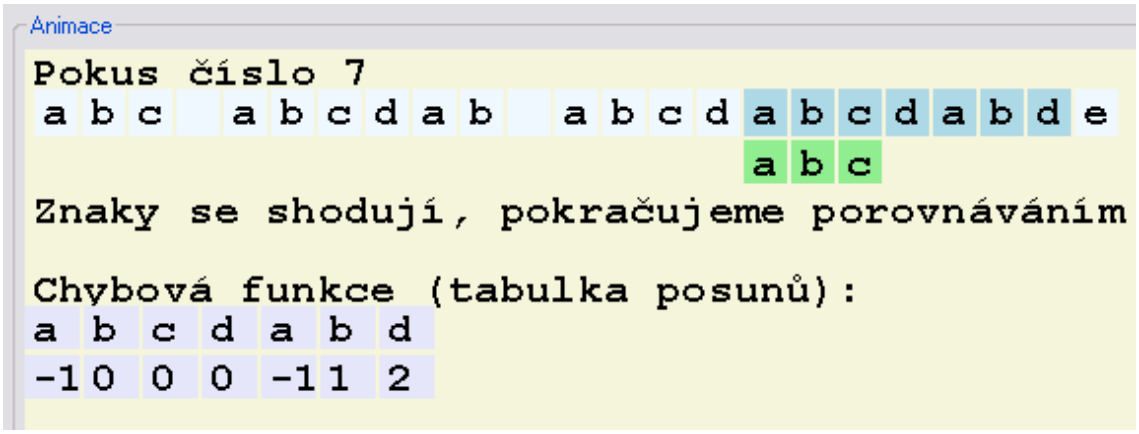
3.2.2. Posun okna podle hodnoty z tabulky

Zpracováním vzoru dostaneme tabulku posunů, která každé pozici vzoru přiřadí číslo, které při neshodě v daném znaku vzoru určí pozici následujícího okna vlevo od aktuální pozice neshody v textu.

3.2.3. Příklad



Obrázek 4. Algoritmus KMP – neshoda



Obrázek 5. Algoritmus KMP – posun

3.2.4. Předpony vzoru v tabulce

V tabulce je uchován údaj o předponách vzoru, v příkladu je vidět, že pokud by algoritmus posunul v Obrázku 5 okno na místo neshody nebo až za ně, přeskočil by výskyt vzoru, který je vidět v okně na Obrázku 5. Algoritmus však posune okno podle tabulky o dvě místa vlevo od neshody v textu, čímž zachytí předponu `ab` a vzor nalezne porovnáním zbývajících znaků.

Program Vizualizace při neshodě znaků zvýrazňuje v tabulce posunů neshodný znak a hodnotu posunu, viz Obrázek 4

3.2.5. Konstrukce tabulky posunů

Při vyplňování tabulky jsou první dvě hodnoty vždy -1 a 0, tedy při neshodě na prvním znaku se okno přesouvá doleva o (-1), tedy doprava o jeden znak a při neshodě na druhém znaku se začátek okna posouvá na tento znak v textu.

Poté algoritmus zjišťuje shodu vzoru se sebou samým – hledá uvnitř vzoru jeho předpony. Pro tento účel si algoritmus v této časti programu udržuje proměnnou, ve které si uchovává velikost nalezené předpony, podle které přiřazuje příslušné hodnoty posunu.

Algoritmus při tvoření tabulky projde cyklem jedenkrát vzor - časová náročnost je O(m).

3.2.6. Časová složitost

Výhodnou vlastností tohoto algoritmu je posouvání okna. K opakovanému porovnání znaku v textu může dojít pouze v tom případě, když dojde k neshodě s hodnotou posunu nula a znak v textu není shodný s prvním znakem vzoru, tento znak je porovnán dvakrát. V případě posunu se shodnou předponou (viz příklad) se tato předpona samozřejmě znovu nekontroluje a pokračuje se znakem za předponou. Časová složitost KMP je tedy lineární O(2n+m).

3.3. Algoritmus Boyer-Moore

3.3.1. Popis

Algoritmus Boyer-Moore (dále BM) stejně jako KMP nejdříve zpracovává vzor. Výsledkem zpracování vzoru jsou v tomto případě dvě tabulky, které nazýváme tabulka posunů a tabulka koncovek.

Zvláštností tohoto algoritmu je, že při porovnávání znaků vzoru v okně postupuje zprava doleva, tedy jakoby od konce.

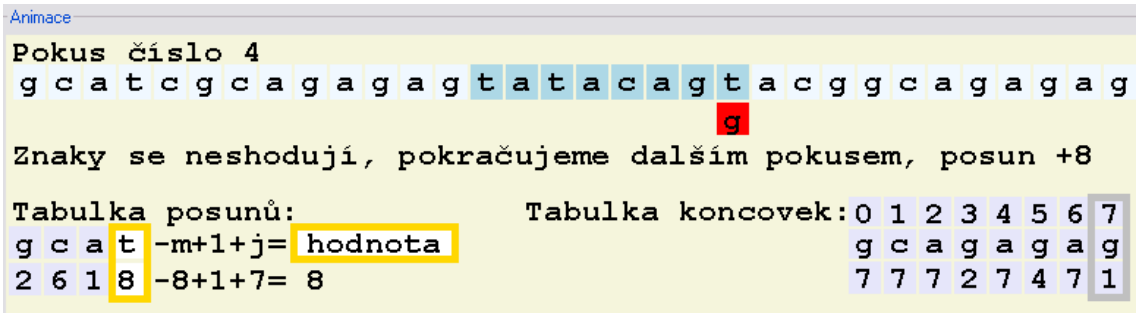
Jeho efektivita vzrůstá se zvětšující se délkou vzoru, při neshodě se znakem v textu, který ve vzoru není, posunujeme okno o celou délku vzoru.

3.3.2. Tabulka posunů

Tabulka posunů se řídí neshodným znakem v textu.

Hodnota v tabulce je vzdálenost znaku od nejpravějšího znaku vzoru, pro znaky abecedy, které ve vzoru nejsou, je hodnota v tabulce délka vzoru (na obrázku 6 má znak `t` v tabulce hodnotu 8, protože se nevyskytuje ve vzoru `gcagagag`).

Výsledná hodnota posunu se počítá dle vzorce T – m + 1 + j, T je hodnota z tabulky posunů, m je délka vzoru, j je pozice znaku ve vzoru



Obrázek 6. Algoritmus Boyer-Moore – tabulka posunů

Popis obrázku 6: vzor je `gcagagag`, neshodný znak v textu je `t`, v tabulce posunů má přiřazeno 8, podle vzorce T - m + 1 + j je to tedy 8 – 8 + 1 + 7 = výsledná hodnota je 8 (zvýrazněno ve vizualizaci, protože 8 je větší než 1 v tabulce koncovek)

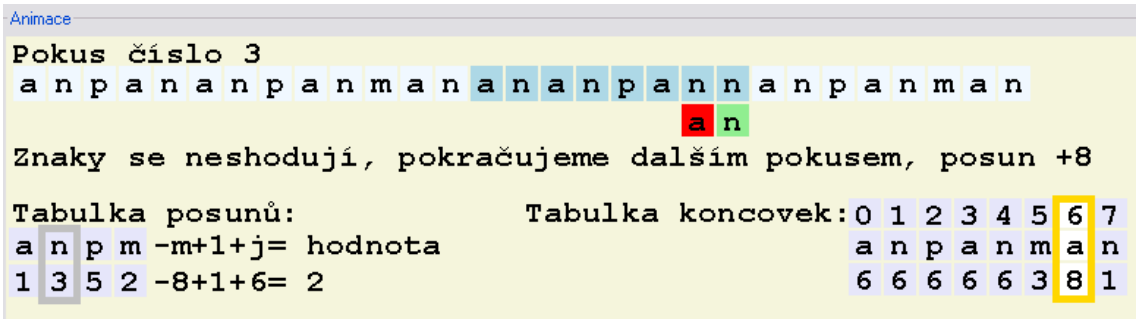
3.3.3. Tabulka koncovek

Hodnota v tabulce koncovek je určena neshodnou pozicí znaku ve vzoru. Pro každou pozici algoritmus bere do úvahy koncovku od dané pozice, a určí počet znaků posunu tak, aby koncovka odpovídala předponě vzoru.

j	vzor	posun
0	anpanman	6
1	npanman	6
2	panman	6
3	anman	6
4	nman	6
5	man	3
6	an	8
7	n	1

Obrázek 7. Algoritmus Boyer-Moore – tabulka koncovek

Znak označený červeně na obrázku 7 je označení pozice neshody. Posunuje se pravý konec okna doprava od pravého konce aktuálního okna.



Obrázek 8. Algoritmus Boyer-Moore – tabulka koncovek ve Vizualizaci

Tabulka koncovek ve Vizualizaci Neshoda na pozici 6 v tabulce koncovek vzoru má přiřazenu hodnotu 8 (ve Vizualizaci zvýrazněno, je větší než 2)

3.3.4. Posuv okna podle tabulek

Při neshodě znaku posune algoritmus okno podle tabulek. Tabulky jsou nazvané tabulka posunů a tabulka koncovek, přičemž hodnota z tabulky posunů je ještě následně upravena podle vzorce, tabulka koncovek je použita přímo.

Po neshodě znaku se bere do úvahy větší z obou hodnot, protože u obou tabulek se jedná o dolní odhady – o kolik nejméně je možné okno posunout. Výsledný posun je posun pravého konce okna doprava od pravého konce aktuálního okna o danou hodnotu.

V programu Vizualizace je větší z obou hodnot zvýrazněna zlatě, druhá hodnota je označena stříbrně.

3.3.5. Časová složitost

Složitost tohoto algoritmu může být sublineární - porovnává znaky od konce okna a v případě, že narazí na znak, který ve vzoru není, tak se přesune doprava o délku vzoru - některé znaky nemusí vůbec porovnávat.

Nejhorší případ si vyžádá 3n porovnání[1][3], složitost je tedy lineární O(n).

3.4. Algoritmus Karp-Rabin

3.4.1. Princip

Algoritmus Karp-Rabin pro porovnávání používá hashovací funkci a porovnává hashe jednotlivých oken a vzoru.

3.4.2. Hashování

Při shodě hashí potom porovnává jednotlivé znaky pro případ kolize - shodné hashe pro různé řetězce.

Protože počítání hashe pro každé okno by bylo velmi časově náročné, tak se celá hash počítá pouze v prvním okně a v dalších se dopočítává z hashe předchozího okna.

Hash řetězce počítá algoritmus jako součet hashí znaků řetězce modulo velikost hashovací tabulky, hash znaku = hodnota znaku * báze^j

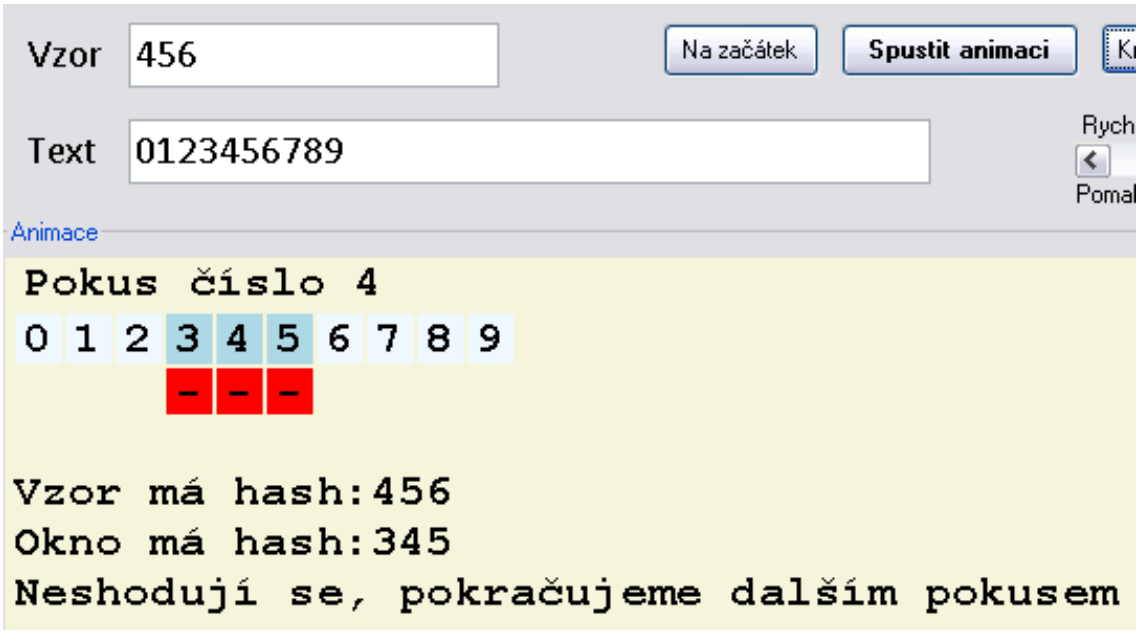
Báze hashe je počet znaků abecedy, nebo to může být velké prvočíslo, exponent `j` je pozice v okně zprava, a hodnota znaku je typicky ASCII hodnota daného znaku.

3.4.3. Výpočet hashe následujícího okna

Obecný princip výpočtu hodnoty hashe následujícího okna ve třech krocích:

1. Odečteme od hashe násobek nejlevější hodnoty okna s hodnotou báze umocněnou délkou vzoru - 1 (tuto hodnotu máme předem spočítanou, protože se používá pořád, v našem příkladu na obrázku 9 je to deset na druhou = 100)
2. Vynásobíme hash bází
3. Přičteme hodnotu v pravé + 1 pozici od okna, čímž jsme se posunuli o 1 pozici doprava a můžeme porovnávat se vzorem.

3.4.4. Příklad



Obrázek 9. Algoritmus Karp-Rabin

Program Vizualizace sestavuje abecedu ze znaků obsažených ve vzoru a textu a přiřazuje jim hodnotu pro výpočet hashe podle pořadí v becedě.

Na obrázku 9 máme situaci, kdy počet znaků abecedy je deset: jsou to znaky nula až devět, báze hashe je deset a máme tedy desítkovou soustavu.

Okno v obrázku 9 v tomto případě je řetězec 345 a hash tomu odpovídá. Stejně tak vidíme, že i hash vzoru odpovídá příslušnému řetězci.

3.4.5. Časová složitost

Příprava (vytvoření hashe pro vzor a první okno provedené zároveň) vyžaduje m kroků, hledání vyžaduje v nejhorším případě O(nm), většinou je však složitost lineární O(n), bereme-li v úvahu porovnání znaků a hashí.

Bohužel máme stále k každém kroku vytvoření nové hashe z hashe starého okna, které zahrnuje násobení.

4. Srovnání výkonnosti

Pro srovnání výkonnosti jsem používal zejména část Graf programu Srovnání.

Provedl jsem nespočet testů pro různé typy textů, ale pořadí algoritmů bylo vždy stejné (pro přiměřeně velký textový soubor).

Nejhůře si vedla nativní metoda IndexOf objektu String frameworku .NET. Tuto jsem zařadil do testů ze dvou důvodů. Za prvé jsem chtěl zjistit, jaký algoritmus byl v této metodě použit a doufal jsem, že pokud by významně odpovídaly časy této metody a jednoho z mnou implementovaných algoritmů, mohl bych s určitou mírou přesnosti toto určit. Druhý důvod byl pragmatičtější – potřeboval jsem spolehlivou kontrolu funkčnosti mých algoritmů, a vždy jsem jejich výsledky porovnával s výsledky této metody.

Nicméně se ukázalo, že v testu rychlosti si tato metoda vedla nejhůře. Protože se mi nepodařilo zjistit, jakým způsobem je tato metoda implementována, budu hádat, že během své činnosti provádí nějaké bezpečnostní kontroly, které ji zpomalují.

Z algoritmů, které jsem implementoval já, si nejhůře vedl algoritmus Karp-Rabin, který při každém posunu okna používá operace násobení a modulo pro výpočet hashe. Ve výsledcích byl ještě horší než algoritmus hrubé síly.

Algorimus hrubé síly byl obvykle uprostřed, nicméně často se zdatně dotahoval na algoritmus KMP, který používá jednu tabulku posunů a jeho implementace je již složitější.

Algoritmus KMP je rychlejší než algoritmus hrubé síly, avšak rozdíly mezi ve výkonu těmito dvěma algoritmy jsou jen velmi malé.

Nejlépe si vedl algoritmus Boyer-Moore, který stabilně vítězil ve všech testech velmi výrazným rozdílem.

5. Uživatelská dokumentace

V této části je popsán způsob ovládání obou programů.

5.0.6. Instalace a spouštění programů

Oba programy je možné jednoduše spustit, není je třeba instalovat a lze je spustit z libovolného záznamového média.

5.0.7. Systémové požadavky

Programy byly vyvíjeny a testovány na operačním systému Windows XP CZ s frameworkem .NET 3.5. Instalační program frameworku je přiložen na CD v adresáři .NET.

5.1. Program Vizualizace

Program slouží k demonstraci činnosti algoritmů, lze jej nalézt v adresáři Vizualizace.

5.1.1. Zadávání vzoru a textu

Obrázek 10. Program Vizualizace – zadání textu a vzoru

Vzor i text lze zapisovat do formuláře. Kvůli přehlednosti je velikost text a vzoru v tomto programu omezena. Po změně vzoru nebo textu ve formuláři se objeví tato výchozí obrazovka s nápisem Nové vyhledávání algoritmem, která se také objeví při změně algoritmu

5.1.2. Změna algoritmu

Obrázek 11. Program Vizualizace – výběr algoritmu

Pokud při změně algoritmu podržíte klávesu **Ctrl**, budou pole vzor a text předvyplněny vhodnými řetězci.

5.1.3. Ostatní ovládací prvky

Po zadání textu, vzoru a algoritmu je možné spustit animaci pomocí tlačítka Spustit animaci.

Rychlost animace lze pomocí stejnojmenného ovládacího prvku.

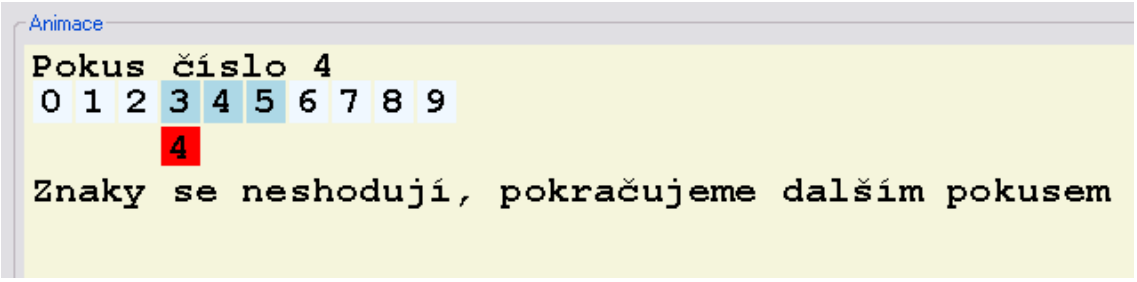
Průběh animace je možno pozastavit tlačítkem Pauza.

Zastavenou animaci lze znovu oživit tlačítkem Pokračovat, nebo je také možné postupovat po jednotlivých obrázcích tlačítky Krok vzad a Krok vpřed, pro které fungují také klávesové zkratky ALT + kurzorová šipka vpravo nebo vlevo.

Pomocí klávesy Na začátek se dostaneme na začátek animace.

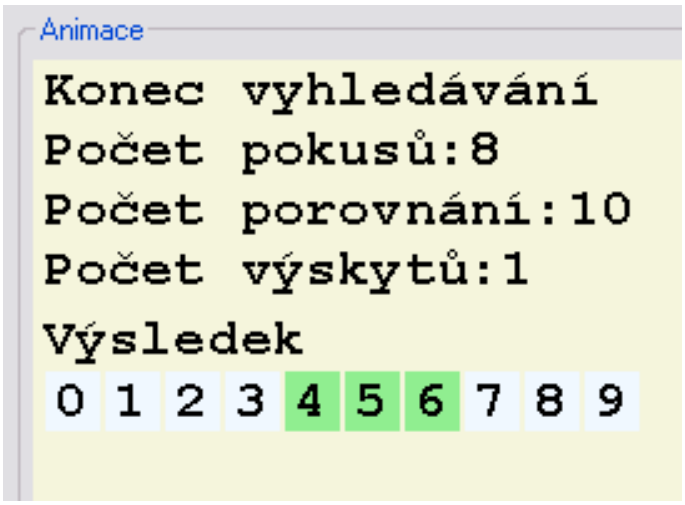
5.1.4. Průběh animace

Za běhu animace je barevně odlišeno okno – ta část textu, kterou aktuálně algoritmus porovnává se vzorem.



Obrázek 12. Program Vizualizace - okno

Na konci animace vidíme počet pokusů, počet porovnání znaků a počet výskytů vzoru v textu. V textu jsou barevně odlišeny výskyty vzoru.

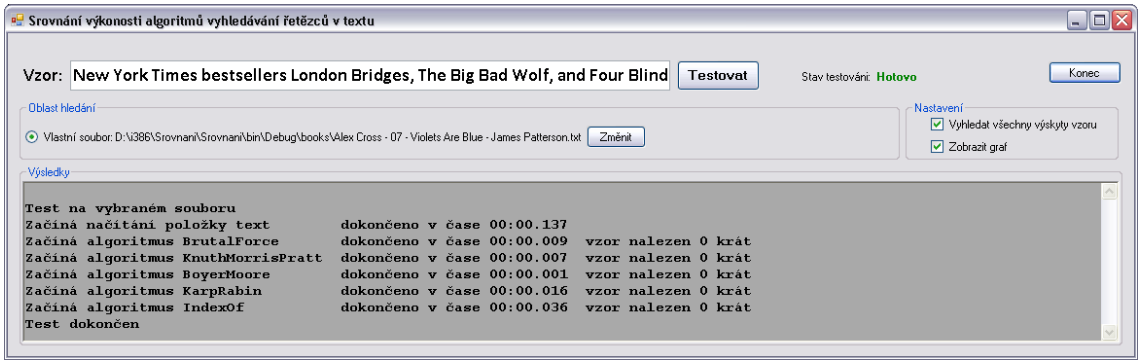


Obrázek 13. Program Vizualizace - konec animace

5.2. Program Srovnání

Program se nachází v adresáři Srovnání.

Tento program testuje výkonnost jednotlivých algoritmů, pro porovnání je zde i metoda IndexOf objektu String frameworku .NET.



Obrázek 14. Program Srovnání

5.2.1. Ovládací prvky

Vzor se zapisuje do formuláře bez diakritiky, text je načten ze zvoleného textového souboru.

K testování jsou připraveny e-booky v adresáři Knihy.

5.2.2. Nastavení

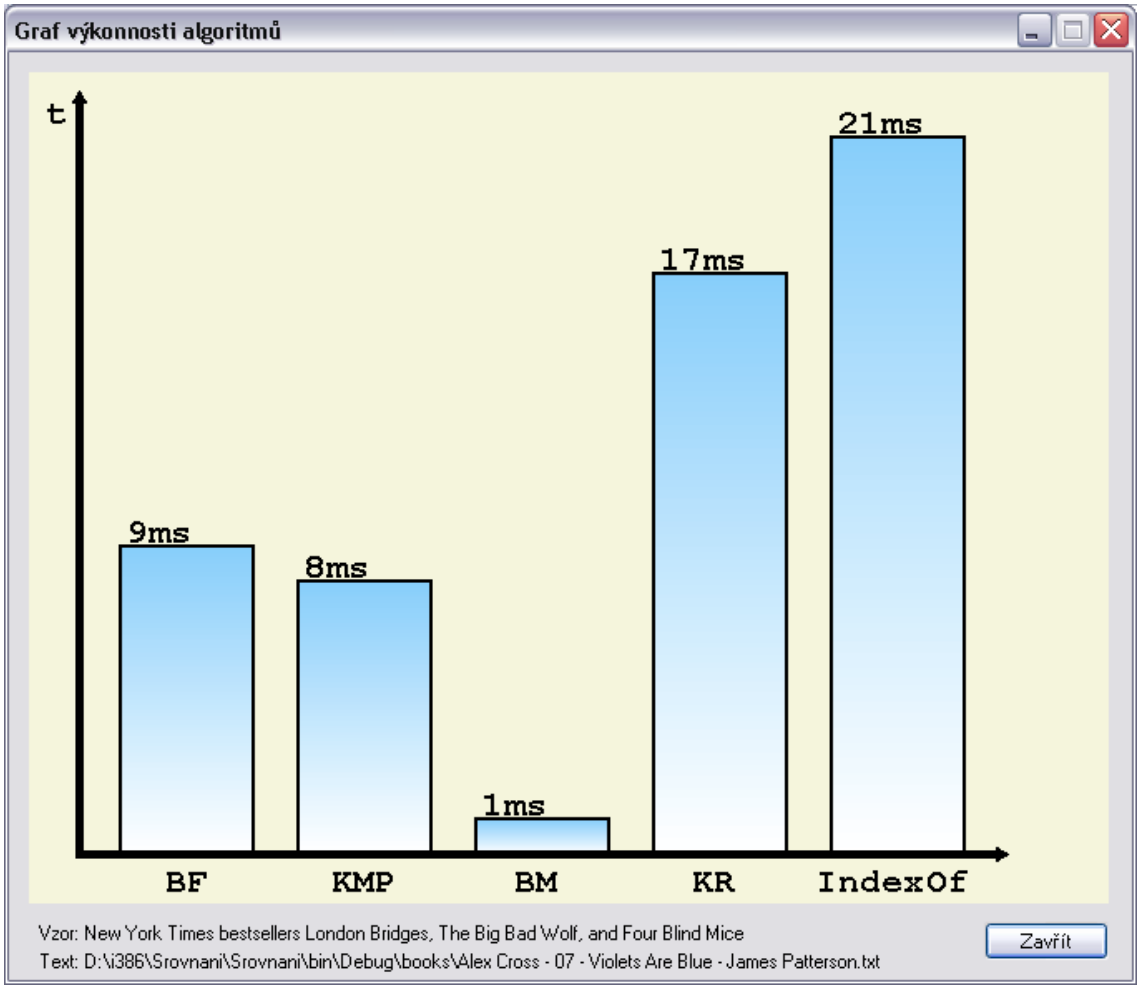
V nastavení lze zvolit vícenásobné hledání a možnost zobrazení grafu výsledků.

5.2.3. Výsledky

Ve výsledcích je kromě časů také pozice vzoru v textu, resp. počet opakování vzoru v textu při zvoleném vícenásobném vyhledávání umožní kontrolu vůči nativnímu IndexOf.

5.2.4. Grafy

Tato část slouží ke grafickému znázornění výkonu jednotlivých algoritmů.



Obrázek 15. Program Srovnání - graf

Graf ukazuje časy hledání jednotlivých algoritmů – delší čas znamená horší výkon algoritmu.

Dole je vzor a zdroj textu, vhodné pro porovnání výkonu při různých vzorech.

Zkratky algoritmů jsou stejné jako v celém textu: BF brutal force – algoritmus hrubé síly, KMP Knuth-Morris-Pratt, BM Boyer-Moore a KR Karp-Rabin.

6. Programátorská dokumentace

6.1. Použité prostředky

Obě aplikace jsou napsány pomocí MS Visual C# 2008 Express Edition

6.2. Architektura programu Vizualizace

Animace programu Vizualizace je pole obrázků typu Metafile, které je vytvořeno po změně řetězce vzoru, textu nebo algoritmu a stisknutí tlačítka Animace nebo tlačítka Krok vpřed.

Jednotlivé obrázky jsou potom zobrazovány při animaci pomocí časovače, nebo podle přání uživatele když tiskne krokovací tlačítka.

Posuvník Rychlost animace mění hodnotu intervalu časovače a tím rychlost animace.

6.3. Architektura programu Srovnání

Na programu Srovnání je zajímavý způsob volání algoritmů: vytvořil jsem si delegáta `delegate int Algoritmus(string t, string v, int o)` a podle něj jsem naprogramoval všechny algoritmy.

Potom jsem vytvořil pole funkcí těchto algoritmů jako delegátů a po převzetí vzoru a načtení textu ze souboru procházím tímto polem, volám v cyklu jednotlivé algoritmy a měřím délku jejich běhu pomocí objektu Stopwatch.

Zobrazení grafu je řešen pomocí přepsání `overide` metody `OnPaint`, kde přímo kreslím do části formuláře.

6.4. Kódy jednotlivých algoritmů

Následně uvedené kódy algoritmů lze najít také v souboru Algoritmy.cs v adresáři doc a jsou použity programem Srovnání.

Algoritmy použité v program Vizualizace byly upraveny tak, aby je bylo možno lépe využít při zobrazení funkčnosti algoritmů - zejména ve dvou případech. Algoritmus Karp Rabin ve Vizualizaci nejdříve zjistí, kolik znaků má abeceda textu a vzoru a podle toho počítá hashe. Algoritmus Boyer Moore přiřazuje v tabulce posunů znakům, které nejsou ve vzoru hodnotu m (délku vzoru), avšak Vizualizace zobrazuje pouze ty z nich, které jsou v textu, a algoritmus na ně může reálně narazit. Toto si můžeme dovolit, protože ve Vizualizaci je délka textu podstatně kratší než v běžné praxi.

6.5. Algoritmus hrubé síly

```
static int BrutalForce(string text, string vzor, int odkud)
{
    int m = vzor.Length;
    int n = text.Length;
    bool pokracujeme = true;
    for (int i = odkud; i < n - m + 1; i++)
    {
        for (int j = 0; j < m && pokracujeme; j++)
        {
            if (text[i + j] != vzor[j]) pokracujeme = false;
        }
        if (pokracujeme) return i;
        pokracujeme = true;
    }
    return -1;
}
```

6.6. Algoritmus Knuth-Morris-Pratt

```
public static int KMP(string text, string vzor, int odkud)
        {
            init_shifts(vzor); //vytvoření tabulky posunů
            int n = text.Length;
            int m = vzor.Length;
            int i = 0, j = odkud;
//j je začátek aktuální shodné pozice v textu,
//i je pozice ve vzoru
            while (i + j < n)
            {
                if (vzor[i] == text[i + j])
                {
                    i++;
                    if (i == m) return j; //platný pokus
                }
                else
                {
                    j = j + i - shift[i];
                    if (shift[i] > -1) i = shift[i];
//posun s platnou předponou - už se nekontroluje
                    else i = 0;
                }
            }
            return -1;
        }
```

6.6.1. KMP – vytvoření tabulky posunů

```
static void init_shifts(string v)
        {
            int m = v.Length;
            int pos = 2,
            int cnd = 0;
//cnd je index (od nuly) vzoru dalšího znaku současného
// uchazeče o předponu
            shift = new int[m];
            shift[0] = -1;
            if (m > 1) shift[1] = 0;
            while (pos < m)
            {
                if (v[pos - 1] == v[cnd])
                //předpona pokračuje
                {
                    shift[pos] = cnd + 1;
                    pos++;
                    cnd++;
                }
                else
                {
                    if (cnd > 0) cnd = shift[cnd];
                //nepokračuje, vracíme se v tabulce
                    else
                    //žádný uchazeč o předponu, cnd je nula
                    {
                        shift[pos] = 0;
                        if (v[pos] == v[0]) shift[pos] = -1;
                        pos++;
                    }
                }
            }
        }
```

6.7. Algoritmus Boyer-Moore

```
public static int BM(string text, string vzor, int odkud)
{
    int m = vzor.Length;
    int n = text.Length;
    int[] posledni = Posledni(vzor);
    int[] dk = DobreKoncovky(vzor);
    int a, b;
    int j = odkud, i;
    while (j <= n - m)
    {
        for (i = m - 1; i >= 0 && vzor[i] == text[i + j]; --i);
        if (i < 0)
        {
            return j;   //platný pokus
        }
        else
        {
            a = dk[i];
            b = posledni[text[i + j]] - m + 1 + i;
            if (a > b) j += a;
            else j += b; //posun o větší hodnotu
        }
    }
    return -1;
}
```

6.7.1. BM – vytvoření tabulky posunů

```
static int[] Posledni(string v) //tabulka posunů
        {
            int m = v.Length;
            int[] posledni = new int[256];
            for (int i = 0; i < 256; i++)
            {
                posledni[i] = m;
                //tabulka ASCII s výchozí hodnotou
            }
            for (int i = 0; i < m; i++)
            {
                posledni[v[i]] = m - i - 1;
                        //vzdálenost od nejpravějšího znaku
            }
            return posledni;
        }
```

6.7.2. BM – vytvoření tabulky koncovek

```
static int[] Koncovky(string v)//pomocná tabulka
        {
            int m = v.Length;
            int[] koncovky = new int[m];
            koncovky[m - 1] = m;
            int g = m - 1;
            int f = 0;
            for (int i = m - 2; i >= 0; i--)
            {
                if (i > g && koncovky[i + m - 1 - f] < i - g)
                    koncovky[i] = koncovky[i + m - 1 - f];
                else
                {
                    if (i < g)
                        g = i;
                    f = i;
                    while (g >= 0 && v[g] == v[g + m - 1 - f])
                        g--;
                    koncovky[i] = f - g;
                }
            }
            return koncovky;
        }

static int[] DobreKoncovky(string v) //vytváří tabulku koncovek
        {
            int m = v.Length;
            int[] dobrekoncovky = new int[m];
            int[] koncovky = Koncovky(v);
            for (int i = 0; i < m; i++) dobrekoncovky[i] = m;
            for (int i = m - 1; i >= 0; i--)
                if (koncovky[i] == i + 1)
                    for (int j = 0; j < m - 1 - i; j++)
                        if (dobrekoncovky[j] == m)
                            dobrekoncovky[j] = m - 1 - i;
            for (int i = 0; i <= m - 2; i++)
                dobrekoncovky[m - 1 - koncovky[i]] = m - 1 - i;
            return dobrekoncovky;
        }
```

6.8. Algoritmus Karp-Rabin

```
static int KarpRabin(string text, string vzor, int odkud)
{
    int q = 659;  //velikost hash tabulky, prvočíslo
    int c = 251;        //báze hashe, prvočíslo
    int m = vzor.Length;
    int n = text.Length;
    int d = 1;
    //nejvyšší řád: báze umocněná délkou vzoru -1
    for (int i = 0; i < m - 1; i++) d = (d * c) % q;
    int hText = 0;
    int hVzor = 0;
                //vytvoření hashe vzoru a prvního okna
    for (int i = 0; i < m; i++)
    {
        hText = (hText * c + (int)text[i+odkud]) % q;
        hVzor = (hVzor * c + (int)vzor[i]) % q;
    }
    int v = odkud;
    bool mismatch;
    if (hText == hVzor) return odkud;
    do
    {
      do
      {
        v++;
        if (v > n - m) return -1;
        //konec, nenašli jsme nic
        hText =
              (hText + (c * q)-((int)text[v-1]*d)) % q;
        hText = (hText * c + (int)text[v + m - 1]) % q;
      }while (hText != hVzor);
    mismatch = false;
    for (int i = 0; i < m && !mismatch; i++)
      {
        if (vzor[i] != text[i + v]) mismatch = true;
      }
    } while (mismatch);
    return v;
}
```

Závěr

Cílem této bylo implementovat nejznámější algoritmy pro vyhledávání textu a vytvořit aplikace pro demonstraci jejich činnosti a účinosti.

Porovnáním učinnosti algoritmů jsme zjistili, že v praxi je nejrychlejší algoritmus Boyer-Moore, který je nicméně poněkud náročný na implementaci.

V případech, kdy tento typ vyhledávání není kritickou částí programu, bych zvážil, zda by nejjednodušší algoritmus hrubé síly nebyl dostačující, a k implementaci algoritmu Boyer-Moore bych přistoupil, až pokud by tato měla patřičný přínos.

Reference

[1] Richard Cole (1991) *Tight bounds on the complexity of the Boyer-Moore algorithm.* Elektronická publikace, 1991.

[2] Stehen, Graham A: String Searching Algorithms. World Scientific Publishing Co., 2000. ISBN: 981-02-1829-X

[3] *Wikipedia* Internetové stránky

[4] *EXACT STRING MATCHING ALGORITHMS* Internetové stránky http://www-igm.univ-mlv.fr/∼lecroq/string/