EKONOMICKÁ UNIVERZITA V BRATISLAVE

FAKULTA HOSPODÁRSKEJ INFORMATIKY

Evidenčné číslo: 103004/D/2020/36097107837859588

Porovnanie exekučnej efektívnosti vyhľadávacieho algoritmu hrubá sila a knižničnej vyhľadávacej funkcie strstr v programe vytvorenom v jazyku C++

Diplomová práca

2020 Bc. Peter Belko

EKONOMICKÁ UNIVERZITA V BRATISLAVE

FAKULTA HOSPODÁRSKEJ INFORMATIKY

Porovnanie exekučnej efektívnosti vyhľadávacieho algoritmu hrubá sila a knižničnej vyhľadávacej funkcie strstr v programe vytvorenom v jazyku C++

Diplomová práca

Študijný program: Informačný manažment

Študijný odbor: Ekonómia a manažment

Školiace pracovisko: Katedra aplikovanej informatiky

Vedúci záverečnej práce: Ing. Igor Košťál, PhD.

Bratislava 2020

Bc. Peter Belko

Čestné vyhlásenie

Čestne vyhlasujem, že som záverečnú prácu vypracoval samostatne a že som uviedol všetku použitú literatúru súvisiacu so zameraním diplomovej práce.

Dátum: Podpis študenta:

Poďakovanie

Srdečne by som chcel poďakovať všetkým, ktorí mi pomáhali počas písania diplomovej práce. Predovšetkým ďakujem môjmu školiteľovi Ing. Igorovi Košťálovi, PhD. za ochotný prístup, pomoc pri písaní záverečnej práce a za jeho podporu počas štúdia.

36097107837859588

Ekonomická univerzita v Bratislave
Fakulta hospodárskej informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta:	Bc. Peter Belko
Študijný program:	informačný manažment (Jednoodborové štúdium, inžiniersky II. st., denná forma)
Študijný odbor:	ekonómia a manažment
Typ záverečnej práce:	Inžinierska záverečná práca
Jazyk záverečnej práce:	slovenský
Sekundárny jazyk:	anglický

Názov: Porovnanie exekučnej efektívnosti vyhľadávacieho algoritmu hrubá sila a knižničnej vyhľadávacej funkcie strstr v programe vytvorenom v jazyku C++

Anotácia: Študent v práci zanalyzuje rôzne druhy vyhľadávacích algoritmov pre hľadanie reťazcov v reťazcoch, ich princípy, použitie a ich implementáciu v programe vytvorenom v jazyku C++ s detailnejším zameraním sa na vyhľadávací algoritmus hrubá sila a knižničnú vyhľadávaciu funkciu strstr. V rámci diplomovej práce študent vytvorí program v jazyku C++, v ktorom implementuje vyhľadávací algoritmus hrubá sila a knižničnú vyhľadávaciu funkciu strstr. Vytvorený program bude schopný zmerať exekučnú efektívnosť vyhľadávacieho algoritmu hrubá sila a knižničnej vyhľadávacej funkcie strstr pri vyhľadávaní reťazca v niekoľko tisíc znakovom reťazci. Pomocou výstupov programu študent vykoná porovnanie exekučnej efektívnosti vyhľadávacieho algoritmu hrubá sila a knižničnej vyhľadávacej funkcie strstr pri vyhľadávaní reťazca vo veľkom reťazci a vyhodnotí toto porovnanie.

Vedúci:	Ing. Igor Košťál, PhD.
Katedra:	KAI FHI - Katedra aplikovanej informatiky FHI
Vedúci katedry:	Ing. Mgr. Peter Schmidt, PhD.
Dátum zadania:	03.11.2017

Dátum schválenia: 03.11.2017 Ing. Mgr. Peter Schmidt, PhD.

vedúci katedry

Abstrakt

BELKO, Peter: Porovnanie exekučnej efektívnosti vyhľadávacieho algoritmu hrubá sila a knižničnej vyhľadávacej funkcie strstr v programe vytvorenom v jazyku C++ – Ekonomická univerzita v Bratislave, Fakulta hospodárskej informatiky, Katedra aplikovanej informatiky. Vedúci záverečnej práce Ing. Igor Košťál, PhD. – Bratislava: FHI EU, 2020, 52 strán.

Cieľom diplomovej práce je vytvorenie programu v programovacom jazyku C++, ktorý porovnáva exekučnú efektívnosť vyhľadávacieho algoritmu hrubá sila s vyhľadávacou funkciou *strstr*. V teoretickej časti práce boli postupne opísané základné vyhľadávacie algoritmy, ich časová a pamäťová náročnosť. Pri každom algoritme sú taktiež spomenuté jeho najväčšie výhody a vizualizácia postupnosti krokov daného algoritmu. Praktická časť diplomovej práce sa zameriava na vytvorenie a opis programu, ktorý porovnáva exekučnú efektívnosť algoritmu hrubá sila a vyhľadávacej knižničnej funkcie *strstr* a vyhodnotenie výsledkov meraní časovej náročnosti oboch metód vyhľadávania. Pri porovnaniach oba algoritmy vyhľadávali v prehľadávanom texte ten istý vyhľadávaný vzor, ktorý definoval používateľ pri spustení programu, alebo bol vzor pri automatickom spustení náhodne generovaný programom. Používateľ si takisto zadefinoval pri automatickom spustení aj počet jednotlivých testovaní. V závere je podľa výsledkov priemerných časov z testovania algoritmu hrubá sila a knižničnej funkcie *strstr* zhodnotené, ktorý spôsob vyhľadávania je efektívnejší.

Kľúčové slova:

vyhľadávací algoritmus, exekučná efektívnosť, porovnávanie, funkcia *strstr*, hrubá sila

Abstract

BELKO, Peter: A comparison of an execution efficiency of the brute force searching algorithm and the strstr library searching function in a program created in C++ – University of Economics in Bratislave, Faculty of Economic Informatics, Department of Applied Informatics. Supervisor of final thesis Ing. Igor Košťál, PhD. – Bratislava: FHI EU, 2020, 52 pages.

The aim of the master thesis is to create a program in the programming language C++, which compares the execution efficiency of the "brute force" search algorithm with the search function *strstr*. In the theoretical part of the work, the basic search algorithms, their time and memory requirements were gradually described. With each algorithm there is also mentions of its biggest advantages and visualization of the sequence of steps of the algorithm. The practical part of the diploma thesis focuses on the creation and description of a program that compares the execution efficiency of the brute force algorithm and the search library function *strstr* and the evaluation of the results of time consuming measurements of both search methods. In the comparisons, both algorithms searched the search text for the same search pattern that the user defined at program startup, or the pattern was randomly generated at startup randomly by the program. The user also defined the number of individual tests during the automatic start. Finally, according to the results of average times from testing the brute force algorithm and the library function *strstr*, it is evaluated which search method is more efficient.

Keywords:

searching algorithm, execution efficiency, comparison, *strstr* function, bruteforce

Obsah

Úvod

Text je jednou z najbežnejších foriem vyjadrenia informácie. Pri práci s textom je ho často potrebné upravovať, aktualizovať. Vyhľadanie upravovanej časti textu v rozsiahlom celku môžu uľahčiť rôzne nástroje.

V prípade textu v digitálnej forme môže byť tento problém jednoduchšie vyriešený. Program namiesto človeka nájde hľadaný text a odkáže používateľa na túto konkrétnu časť textu. Avšak texty sú často také rozsiahle, že jednoduché prehľadávanie nemusí byť postačujúce, a preto programátor vytvárajúci vyhľadávací program musí použiť najefektívnejší vyhľadávací algoritmus.

V tejto práci sa zaoberáme základnými algoritmami na vyhľadávanie textu, ich rozdelením, ich výhodami a nevýhodami a ich praktickým využitím.

Takéto algoritmy sú pomerne komplexné, nie sú pre bežného človeka intuitívne, oproti iným algoritmom, ako napríklad triediace algoritmy, a práve preto sa môže v informačnom systéme vyskytnúť ich nesprávne použitie.

1 Súčasný stav riešenej problematiky doma a v zahraničí

V súčasnom stave je dostatočné množstvo kvalitnej literatúry, ktorá vysvetľuje problém hľadania reťazca v texte (angl. „string-searching algorithm“). Tak isto je to aj v prípade algoritmu hrubá sila (angl. „brute force“, často označovaný aj ako „naive“) a knižničnej funkcie *strstr* („str“ označuje slovo „string“, do slovenčiny prekladané ako „reťazec“), ktorá je využívaná vo viacerých programovacích jazykoch a každý skúsený programátor vie, čo sa skrýva pod týmto označením.

Ponúkaná literatúra [1] poskytuje exaktné vysvetlenie princípu fungovania algoritmu **hrubá sila**, ktorý je pre človeka jedným z najintuitívnejších riešení tohto problému. Nie je ťažké nájsť vhodnú literatúru, ktorá by tento algoritmus vzorovo implementovala vo väčšine programovacích jazykov. Tento algoritmus je bežne používaný komunitou programátorov, preto je naň často odkazované na rôznych fórach o programovaní. Tak ako jeho koncept, tak i jeho implementácia je pomerne jednoduchá, a preto je tento algoritmus často vyučovaný ako jeden zo základných algoritmov, v problematike hľadania podreťazca vo väčšom reťazci alebo pri obdobnom probléme, pre ktorý je možné použiť algoritmus hrubá sila. Algoritmus hrubá sila je síce jednoduchý a veľmi intuitívny, ale nevyužíva dostatočne potenciál výpočtovej techniky, a preto má v porovnaní s inými algoritmami často dlhší exekučný čas. [1]

Jednou z najdôležitejších informácií o funkcii ***strstr*** je, že táto funkcia môže byť implementovaná rôznymi algoritmami. Pri bežnom používaní programátor často nepotrebuje vedieť o konkrétnej implementácii tejto knižničnej funkcie a len ju používa podľa predpísanej deklarácie a dokumentácie. Namiesto podrobného skúmania o aký algoritmus ide, programátori často implementujú svoje špecifické riešenie a taktiež nazvú túto funkciu *strstr*. Toto označenie sa teda stalo akýmsi predpisom názvu funkcie tohto typu. Používanie zaužívaných označení je dobrá praktika programátora, ktorá zvyšuje čitateľnosť a zrozumiteľnosť kódu. Používanie knižničných funkcií je rýchle a pohodlné riešenie pre väčšinu programátorov.

V súčasnosti je teoretická časť vyhľadávacích algoritmov veľmi dobre zmapovaná a nie je možné prispieť veľkou zmenou v súčasnom chápaní danej problematiky. Najznámejšie algoritmy sú Knuth–Morris–Pratt algoritmus, Boyer–Moore algoritmus a takzvaný „Two-

way“ algoritmus. Tieto algoritmy sú spomínané v literatúre na Slovensku ako aj v zahraničí. [1]

Aj napriek tomu sú vyhľadávacie algoritmy v informačných systémoch a ich výučba veľmi potrebné. Bez nich by sme neboli schopní nájsť konkrétnu informáciu vo veľkom zhluku dát a mnohé operácie by tak boli omnoho neefektívnejšie. Pri týchto typoch problémov je potrebné poznať očakávané dáta, ako sa s nimi bude pracovať a podľa toho vybrať najvhodnejší algoritmus.

1.1 Hľadanie reťazca

Hľadanie reťazca je v informatike proces, pri ktorom je hľadaný výskyt určitého vzoru (angl. „pattern“) vo väčšej dátovej štruktúre. Je to špecifický problém z obšírnejšej skupiny vyhľadávacích algoritmov, do ktorej patria aj komplexnejšie problémy, ako napríklad optimalizačné algoritmy.

Pred používaním výpočtovej techniky bol problém s organizovaním dát pomerne náročný a aj pokročilé riešenia v bežnej praxi, ako napríklad usporiadané zložky pacientov v kartotékach, boli nakoniec pri bežných činnostiach, ako napríklad vyhľadanie pacienta, pomerne prácne. To sa zlepšilo s príchodom výpočtovej techniky a skoro automaticky s tým vznikla potreba vyhľadávať informácie. Od počiatku priťahoval problém hľadania veľký záujem výskumníkov, snáď vďaka zložitosti efektívneho riešenia, napriek jednoduchému a intuitívnemu zadaniu. O tom svedčí aj to, že dodnes používame algoritmy, ktoré boli sformulované už počas 60.-80. rokov 20. storočia, ako napríklad veľmi používaný Knuth–Morris–Pratt algoritmus, ktorý bol publikovaný v roku 1970. [2]

Vyhľadávacie algoritmy sú prevažne prezentované v úvodných kurzoch informatiky a programovania, kedy množstvo algoritmov riešiacich rovnaký problém poskytuje dobrý úvod k rôznym fundamentálnym konceptom algoritmizácie a analýze zložitosti. [3]

Hlavným cieľom algoritmu je vyriešiť, kde sa v texte nachádza prvý výskyt hľadaného slova alebo znakového reťazca. Tento algoritmus môže byť implementovaný rôznymi spôsobmi. V niektorých prípadoch je návratová hodnota index prvého znaku slova v texte, v prípade jazyka C/C++ sa využíva taktiež návratová hodnota ukazovateľ na prvý znak. V prípade ak sa slovo nepodarí nájsť, funkcia zvyčajne vracia NULL alebo inú hodnotu, ktorá je definovaná v dokumentácii.

Ako vstupné údaje vyhľadávacích funkcii je povinné slovo, ktoré sa bude hľadať a text v ktorom sa bude toto slovo hľadať.

V súčasnosti musí informačný systém často obsahovať pokročilejšie možnosti pri vyhľadávaní, ako napríklad ignorovať, či je písmeno malé alebo veľké, nájsť podobné slová, hľadať podľa predpisu – vyhľadávanie regex, hľadať naprieč viacerými dátovými štruktúrami.

Regex je pomerne komplexné vyhľadávanie, pomocou ktorého je možné hľadať napríklad britské slovo „*color*“ a zároveň jeho americký ekvivalent, slovo „*colour*“, a to za pomoci regulárneho výrazu, ktorého predpis by vyzeral nasledovne:

```
colou?r
```

Znak „?“ vyjadruje, že predchádzajúci znak „u“ nie je povinný. Regex je silný nástroj používaný predovšetkým pri validácii dát, pri rozpoznávaní rukou napísaného textu, pri rôznych úpravách textu a taktiež je implementované v častiach systémov, ktoré hľadajú a zvýrazňujú text. [4]

1.2 Klasifikácia vyhľadávacích algoritmov

Vyhľadávacie algoritmy je možné klasifikovať podľa rôznych kritérií. Toto delenie napomáha lepšiemu pochopeniu konkrétneho algoritmu, a teda pomáha aj lepšiemu výberu z určitej skupiny pre vyriešenie konkrétneho problému. Ide nielen o výpočtovú zložitosť, využitie pamäte, ale aj o stabilitu programu, či spôsob, akým algoritmus vykonáva hľadanie.

Jedno zo základných rozdelení vyhľadávajúcich algoritmov je rozdelenie, či naraz pracuje s jedným vzorom alebo s viacerými vzormi naraz.

Nech m je dĺžka hľadaného slova – vzoru a nech n je dĺžka prehľadávaného textu, pričom k je veľkosť abecedy. Potom môžeme vyhľadávacie algoritmy rozdeliť nasledovne

Algoritmy s jedným vzorom

- Brute force
- Rabin-Karp
- Knuth–Morris–Pratt
- Boyer–Moore string-search algorithm

- Bitap
- Two-way
- BNDM (skr. z „Backward Nondeterministic Dawg Matching algorithm“)

Algoritmy s konečným počtom vzorov

- Aho–Corasick
- Commentz-Walter
- Set-BOM

Algoritmy a predspracovanie [5]

	Text nie je predpripravený	**Text je predpripravený**
Vzor nie je predpripravený	*Základné algoritmy*	*Indexové metódy*
Vzor je predpripravený	*Pokročilé vyhľadávacie mechanizmy*	*Špecifické metódy*

Ďalšia klasifikácia môže rozdeliť algoritmy podľa ich vyhľadávacej stratégie: [6]

- Najskôr vyhľadaj prefix (Knuth-Morris-Pratt, Shift-And, Aho-Corasick)
- Najskôr vyhľadaj suffix (Boyer-Moore, Commentz-Walter)
- Najskôr vyhľadaj najlepší faktor (BNDM, BOM, Set-BOM)
- Iné stratégie (Naive, Rabin-Karp)

Výpočtová zložitosť vyhľadávajúcich algoritmov

Výpočtová zložitosť je pojem z teórie algoritmov, ktorý vyjadruje nakoľko je výpočet podľa zvoleného algoritmu zložitý. Jeden z najdôležitejších faktorov algoritmu je jeho výpočtová zložitosť, ktorú opisuje teória zložitosti. Najbežnejšie používaný ukazovateľ efektívnosti algoritmu je časová výpočtová zložitosť, ktorá opisuje závislosť počtu vykonaných operácií od veľkosti vstupných údajov.

Výpočtová zložitosť má dve základné miery:

- časová zložitosť
- pamäťová zložitosť

Optimalizácia algoritmu sa týka minimalizácie jednej alebo obidvoch mier zložitosti.

Na vyjadrenie zložitosti algoritmov je využívaný tzv. „Big O Notation“, ktorý vyjadruje vzájomné prepojenie medzi veľkosťou vstupných parametrov a následnou náročnosťou algoritmu. [7]

Časová zložitosť vyhľadávajúcich algoritmov

Pri hľadaní efektívneho riešenia je potrebné mať nástroj, ktorým je možné zmerať rýchlosť jednotlivých algoritmov, prípadne ich medzi sebou porovnať. Na tieto účely sú využívané znalosti z oblasti zložitosti algoritmov.

Napríklad časová náročnosť O(1) predstavuje jeden krok algoritmu, ktorého trvanie je konštantné a je vlastne definované technickými parametrami. Ak by takýchto operácií, ktoré trvajú konštantný čas, bolo potrebné vykonať *n*-krát, potom je časová náročnosť O(n). Mnohé algoritmy dosahujú časovú náročnosť $O(n^2)$, ktorú by dosahoval aj nasledujúci algoritmus.

```
void printAllPossibleOrderedPairs(int arr[], int size)
{
    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            printf("%d = %d\n", arr[i], arr[j]);
        }
     }
}
```

Zdrojový kód č. 1: Ukážka algoritmu [Zdroj: Vlastné spracovanie]

Podobne potom existujú algoritmy, ktoré majú kvadratickú, logaritmickú, polynomickú, exponenciálnu či faktoriálovú.

Mnohé pokročilé informačné systémy vykonajú denne nespočetné množstvo operácií, ktoré v konečnom dôsledku potrebujú elektrickú energiu. Z ekonomického hľadiska je preto neprípustné vykonávať nepotrebné operácie. Optimalizácia algoritmov je preto veľmi potrebná.

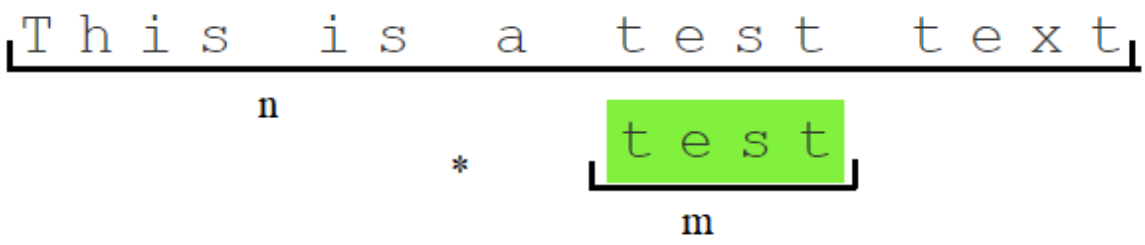
V niektorých systémoch od efektívnosti algoritmov závisí vyriešenie problému. Niektoré systémy, ako napríklad zabezpečenie heslom, sa spoliehajú na to, že ak by bol pri pokuse o prelomenie použitý algoritmus hrubá sila, trvalo by to niekoľko rokov.

V nasledujúcej tabuľka trvá jedna operácia jednu nanosekundu.

	Veľkosť vstupných dát						
	10	20	50	100	1 000	1 000 000	1 000 000 000
log log n	2 ns	3 ns	3 ns	3 ns	4 ns	5 ns	5 ns
log n	4 ns	5 ns	6 ns	7 ns	10 ns	20 ns	30 ns
$\sqrt{n}$	4 ns	8 ns	8 ns	10 ns	32 ns	1 µs	32 µs
n	10 ns	20 ns	50 ns	100 ns	1 µs	1 ms	1 s
n log n	34 ns	87 ns	283 ns	665 ns	10 µs	20 ms	30 s
$\mathbf{n^2}$	100 ns	400 ns	3 µs	10 µs	1 ms	16 min 40 s	32 rokov
$\mathbf{n^3}$	1 µs	8 µs	125 µs	1 ms	1 s	32 rokov	
$\mathbf{n^4}$	10 µs	160 µs	6 ms	100 ms	16 min 40 s	31 688 088 rokov	
$\mathbf{2^n}$	1 µs	1 ms	13 dní	40×10^{12} rokov			
$\mathbf{3^n}$	59 µs	4 s	22 760 000 rokov				
n!	4 ms	77 rokov					
$\mathbf{n^n}$	10 s	$3{,}32\times10^{9}$ rokov					
$\mathbf{2^{2^n}}$	$5{,}7\times10^{291}$ rokov						

Tabuľka č. 1: Časové zložitosti v reálnom čase [Zdroj: Vlastné spracovanie]

V prípade algoritmu hrubá sila je časová zložitosť O(*n***m*), pretože pozostáva z dvoch cyklov. Vonkajší cyklus sa pohybuje po texte a vnútorný cyklus sa posúva v hľadanom slove.



Obrázok č. 1: Znázornenie časovej zložitosti [Zdroj: Vlastné spracovanie]

Prehľadná tabuľka časovej zložitosti iných algoritmov:

Algoritmus	Časová zložitosť
Hrubá sila	Θ(nm)
Rabin-Karp	Θ(n + m)
Knuth–Morris–Pratt	Θ(n)
Boyer–Moore	O(mn)
Bitap	O(mn)
Two-way	O(n+m)
BNDM	O(n)
BOM	O(mn)
FM-index	O(m)

Tabuľka č. 2: Časová zložitosť algoritmov [Zdroj: Vlastné spracovanie]

Niektoré algoritmy dosahujú menšiu časovú zložitosť vďaka predpripraveným pomocným tabuľkám, ktoré ale využívajú extra pamäť.

Pamäťová zložitosť vyhľadávajúcich algoritmov

Pamäťová zložitosť je veľmi podobná časovej zložitosti. V niektorých prípadoch je potrebné optimalizovať ukladací priestor namiesto optimalizácie časovej zložitosti.

Nasledovná funkcia využíva konštantnú veľkosť pamäte, O(1), ktorá nezáleží od veľkosti vstupných parametrov.

```
function sum(array) {
	var total = 0;
	for (let i = 0; i < array.length; i++) {
		console.log("${total}");
	}
}
```

Zdrojový kód č. 2: Ukážka algoritmu [Zdroj: Vlastné spracovanie]

Ďalšia funkcia využíva pamäť, ktorá sa lineárne zväčšuje s veľkosťou vstupných parametrov. Takýto algoritmus má teda pamäťovú zložitosť O(n).

```
function sayHiNTimes(n) {
	const hiArray = [];
	for (let i = 0; i < n; i++) {
		hiArray[i] = 'hi';
	}
	return hiArray;
}
```

Zdrojový kód č. 3: Ukážka algoritmu [Zdroj: Vlastné spracovanie]

Skúmanie pamäťovej zložitosti bolo potrebné najmä v minulosti, kedy bola pamäť veľmi obmedzená. Dodnes je táto téma dôležitá pri špeciálnych zariadeniach, ako napríklad vesmírne roboty, počítače malých rozmerov, ktoré zo špecifických dôvodov nedisponujú veľkým množstvom voľnej pamäte.

Algoritmus hrubá sila nevyužíva žiadnu doplnkovú pamäť, čo je jedna z jeho najväčších výhod.

Prehľadná tabuľka pamäťovej zložitosti iných algoritmov:

Algoritmus	Pamäťová zložitosť
Hrubá sila	-
Rabin-Karp	O(1)
Knuth–Morris–Pratt	Θ(m)
Boyer–Moore	Θ(k)
Two-way	O(1)
FM-index	O(n)

Tabuľka č. 3: Pamäťová zložitosť algoritmov [Zdroj: Vlastné spracovanie]

2 Prehľad vyhľadávacích algoritmov

V nasledujúcej časti budú predstavené základné vyhľadávacie algoritmy, spôsob ich fungovania, ich časová náročnosť a taktiež pamäťová náročnosť. K algoritmom je priložený pseudokód a vizualizácia priebehu triedenia pre ľahšie pochopenie daného algoritmu.

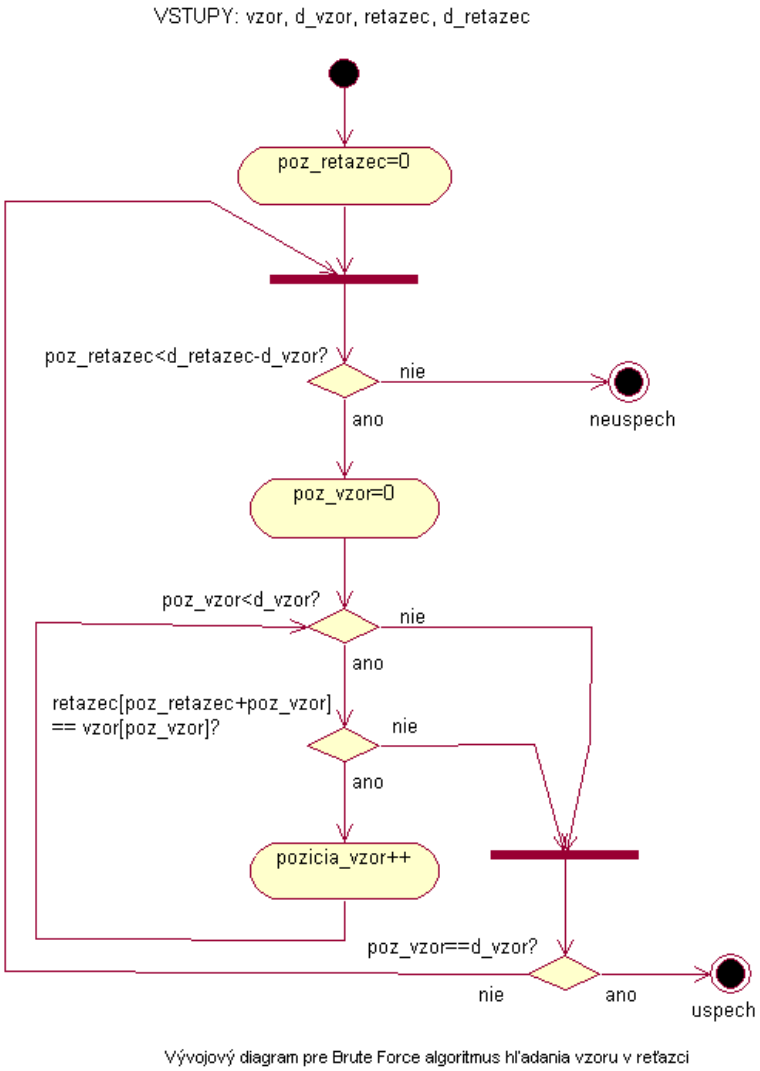
2.1 Hrubá sila

Algoritmus hrubá sila odkazuje na riešenie problému, ktoré neobsahuje žiadne zjednodušenia na zvýšenie efektívnosti, ale namiesto toho sa spolieha na výpočtovú silu počítača a algoritmus kontroluje postupne všetky možnosti, až kým nie je nájdené riešenie.

Algoritmus hrubá sila je veľmi všeobecné riešenie problému a vyskytuje sa v programovaní veľmi bežne aj pri riešení rôznych iných problémov. Ide o jeden z najrozšírenejších algoritmov, ktorý je v mnohých prípadoch pre človeka najintuitívnejší. Aj v prípade riešenia vyhľadávania sa môže vyskytovať v rôznych obmenených podobách, ktoré ale fungujú na základnom princípe a to, že je postupne prechádzané cez každý symbol v texte a ten je porovnávaný so symbolmi vo vzore.

```
Algorithm BruteForce(T[0...n-1], P[0...m-1]) {
        for i ← 0 to n-m do
                j ← 0
                while j < m and P[j] = T[i+j] do
                        j++
                if j=m then return i
        return -1
}
```

Zdrojový kód č. 4: Pseudokód algoritmu hrubá sila [Zdroj: Vlastné spracovanie]



Vývojový diagram pre Brute Force algoritmus hľadania vzoru v reťazci

Obrázok č. 2: Vývojový diagram algoritmu hrubá sila [Zdroj: 8]

Obrázok č. 3: Vizualizácia algoritmu hrubá sila [Zdroj: Vlastné spracovanie]

Hlavnou výhodou algoritmu hrubá sila je jeho ľahká implementácia. Vďaka tejto vlastnosti často slúži ako skvelý nástroj na výučbu algoritmizácie a optimalizácie algoritmov.

2.2 Rabin-Karp

Algoritmus hrubá sila postupne porovnáva každý symbol textu s každým symbolom vo vzore. Podobne ako pri algoritme hrubá sila, aj algoritmus Rabin-Karp postupuje krok po kroku, avšak namiesto porovnávania konkrétnych symbolov porovnáva hash hodnotu (t.j. vygenerovaná číselná interpretácia) konkrétneho výseku textu s hash hodnotou hľadaného vzoru. Ak algoritmus nájde zhodu v týchto hodnotách, tak vtedy skontroluje konkrétne symboly.

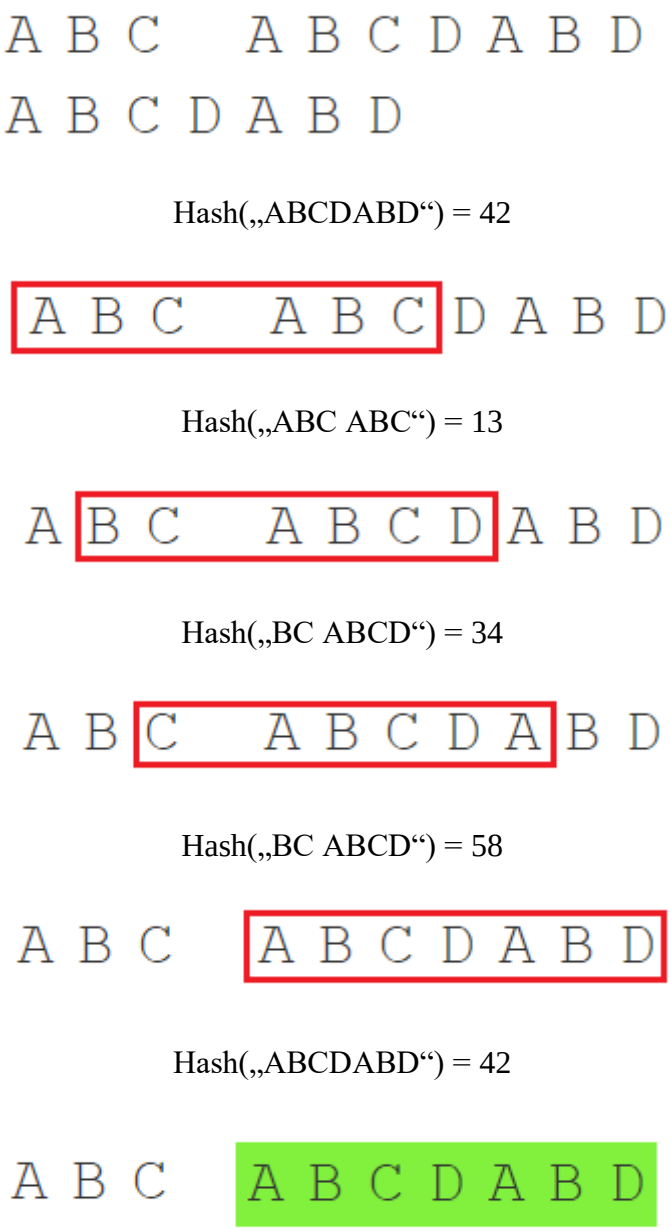
Takže algoritmus Rabin-Karp musí vypočítať hash hodnoty pre nasledovné reťazce:

- Vzor
- Všetky výseky textu, ktoré majú dĺžku ako vzor

Aby bol tento algoritmus dostatočne efektívny, musia byť hash hodnoty počítané s nasledovnými vlastnosťami. Hash hodnoty pri posune musia byť efektívne počítané z predchádzajúcej hash hodnoty tak, aby časová zložitosť výpočtu hash hodnoty dosahovala hodnotu O(1). [9]

```
Algorithm RabinKarp(T[0...n-1], P[0...m-1])  {
        n ← length[T]
        m ← length[P]
        h ← d m-1 mod q
        p ← 0
        t0 ← 0
        for i ← 1 to m do
                p  ← (dp + P[i]) mod q
                t0  ← (dt0 + T[i]) mod q
        for s ← 0 to n-m do
                if p = ts
                        then if P[1..m] = T[s+1.. s+m]
                                return s
                if s < n – m
                        then ts+1 ← (d(ts - T[s+1]h) + T[s+m+1]) mod q
        return -1
}
```

Zdrojový kód č. 5: Pseudokód algoritmu Rabin-Karp [Zdroj: 9]



Obrázok č. 4: Vizualizácia algoritmu Rabin-Karp [Zdroj: Vlastné spracovanie]

Výhodou algoritmu Rabin-Karp je, že sa dá ľahko upraviť aj na riešenie iných problémov. Je hlavne používaný pri pracovaní s dvojrozmerným poľom, porovnávaní otlačkov prstov, DNA porovnávaní a podobne.

2.3 Knuth-Moris-Pratt

Hlavnou charakteristikou algoritmu Knuth-Moris-Pratt (skrat. KMP) je, že vždy keď sa nájde nezhoda medzi symbolom vo vzore a v texte, algoritmus použije predpripravenú tabuľku prefixov vzoru, aby vedel, o koľko znakov sa môže posunúť tak, aby znova nekontroloval už raz kontrolované symboly v texte. Tým optimalizuje zbytočné opakované porovnávanie, ktoré vykonáva napríklad aj algoritmus hrubá sila, ktorý sa po nezhode symbolov medzi textom a vzorom posunie v texte iba o jednu pozíciu.

Tabuľka, ktorú je potrebné vytvoriť pred samotným porovnávaním je v podstate veľmi triviálna. Táto tabuľka je často označovaná ako LPS (skrat. z angl. „longest prefix suffix“) alebo „Π table“, pričom v programe je reprezentovaná ako pole celých čísiel. Nakoľko táto tabuľka odpovedá hľadaným prefixom a sufixom vzoru, aj jej dĺžka je rovnaká ako dĺžka vzoru. Na začiatku inicializujeme *lps[0]* a premennú len na hodnotu 0. Ak sú hodnoty na pozíciách *vzor[len]* a *vzor[i]* rovnaké, navýšime hodnotu *len* o 1 a túto hodnotu uložíme na pozíciu *lps[i]*. Ak sa hodnoty na pozíciách *vzor[len]* a *vzor[i]* nezhodujú a hodnota premennej *len* nie je 0, aktualizujeme hodnotu premennej *len* na hodnotu *lps[len-1]*.

j	0	1	2	3	4	5
vzor [j]	a	b	a	b	a	c
lps [j]	0	0	1	2	3	0

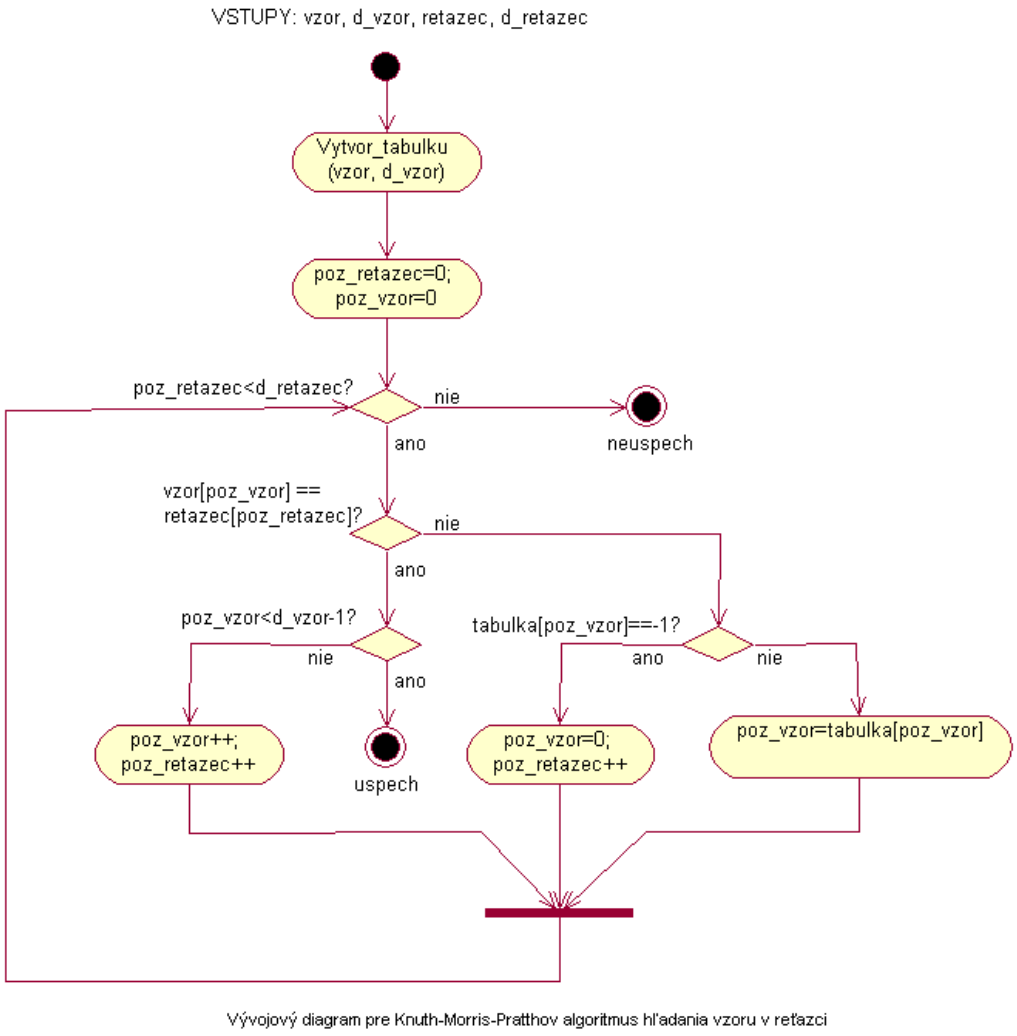
Tabuľka č. 4: Príklad LPS tabuľky [Zdroj: Vlastné spracovanie]

```
Algorithm LPSTable(P[0...m-1]) {
        i ← 1
        j ← 0
        while i ≤ m-1 do
                if P[j] = T[j] then
                        lps[i] ← j + 1
                        i ← i + 1
                        j ← j + 1
                else if j > 0 then
                        j ← lps[j-1]
                else
                        lps[i] ← 0
                        i ← i + 1
}
```

Zdrojový kód č. 6: Pseudokód algoritmu na vytváranie tabuľky LPS [Zdroj: 10]

```
Algorithm KMP(T[0...n-1], P[0...m-1]) {
        lps ← LPSTable(P)
        i ← 0
        j ← 0
        while i < n do
                if P[j] = T[i] then
                        if j = m - 1 then
                                return i - m - 1
                        i ← i + 1
                        j ← j + 1
                else if j > 0 then
                        j ← lps[j-1]
                else
                        i ← i + 1
        return -1
}
```

Zdrojový kód č. 7: Pseudokód algoritmu KMP [Zdroj: 10]



Obrázok č. 5: Vývojový diagram algoritmu KMP [Zdroj: 11]

A B A B C A B A B D

A B A B D

i	lps
0	-1
1	0
2	0
3	1
4	2

A B A B C A B A B D

A B A B D

i	lps
0	-1
1	0
2	0
3	1
4	2

Nezhoda na pozícii 4, lps(4) = 2, takže sa vzor posunie o (4) - (2) = 2.

A B A B C A B A B D

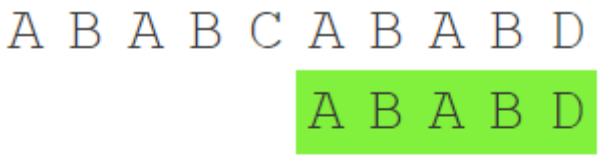
A B A B D

i	lps
0	-1
1	0
2	0
3	1
4	2

Nezhoda na pozícii 2, lps(2) = 0, takže sa vzor posunie o (2) - (0) = 2.

i	lps
0	-1
1	0
2	0
3	1
4	2

Nezhoda na pozícii 0, lps(0) = -1, takže sa vzor posunie o (0) - (-1) = 1.



i	lps
0	-1
1	0
2	0
3	1
4	2

Obrázok č. 6: Vizualizácia algoritmu KMP [Zdroj: Vlastné spracovanie]

Najväčšou výhodou algoritmu Knuth-Moris-Pratt je schopnosť preskočiť veľké množstvo znakov bez zbytočného opakovaného kontrolovania. Tento algoritmus sa na určitú dobu stal štandardom na vyhľadávanie vzoru v texte. Aj vďaka jeho pomerne ľahkej implementácii je dodnes využívaný v mnohých systémoch, napríklad pri analýze DNA reťazca, ktorý sa skladá zo 4 symbolov (A,C,G,T). [2]

2.4 Boyer-Moore

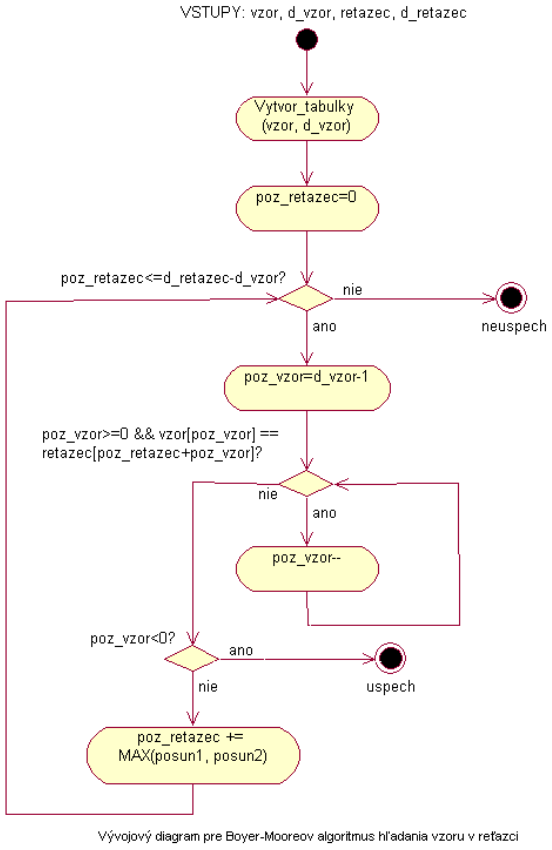
Najväčší rozdiel medzi algoritmom Boyer-Moore a predchádzajúcimi algoritmami je, že namiesto klasického prístupu porovnávania textu a vzoru zľava doprava, Boyer-Moore

porovnáva symboly sprava doľava, teda prvé porovnávanie začína na poslednom symbole vo vzore a v texte na pozícii určenou dĺžkou hľadaného vzoru. Takýmto spôsobom sa vie posúvať textom omnoho efektívnejšie.

Taktiež ako KMP algoritmus, aj Boyer-Moore využíva predpripravenú tabuľku, ktorá analyzuje hľadaný vzor, a ktorú využíva na určenie, o koľko sa môže hľadaný vzor posunúť. Boyer-Moore algoritmus ale ešte využíva druhú predpripravenú tabuľku, v ktorej sa nachádzajú unikátne symboly vzoru a ich posledný výskyt v rámci vzoru. [12]

```
Algorithm BoyerMoore(T[0...n-1], P[0...m-1) {
        st ← Vytvor suffix tabuľku
        cht ← Vytvor tabuľku charakterov
        i ← 0
        while s ≤ n - m do
                j ← m
                while j > 0 and P[j] = T[s+j] do
                        j ← j - 1
                if j = 0 then
                        s ← s + st[0]
                else
                        s ← s + max(st[j], j-cht[T[s+j]]
}
```

Zdrojový kód č. 8: Pseudokód algoritmu Boyer-Moore [Zdroj: Vlastné spracovanie]



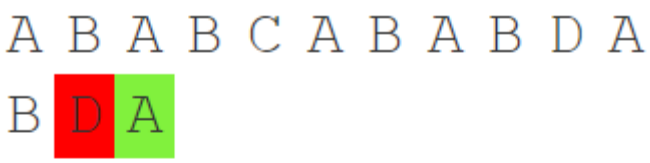
Obrázok č. 7: Vývojový diagram algoritmu Boyer-Moore [Zdroj: 13]

A B A B C A B A B D A

B D A

i	Suffix Location
0	-3
1	-2
2	1

char	Last Occurrence
B	0
D	1
A	2



i	Suffix Location
0	-3
1	-2
2	1

char	Last Occurrence
B	0
D	1
A	2

discrepancy_index = 1

last_occurrence(B) = 0

good_suffix(1) = -2

Good Suffix umožňuje skok (1) - (-2) = 3. Last Occurrence umožňuje skok (1) - (0) = 1. Algoritmus vyberá Good Suffix.

i	Suffix Location
0	-3
1	-2
2	1

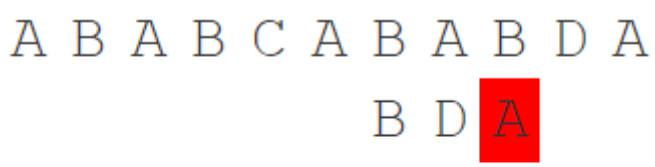
char	Last Occurrence
B	0
D	1
A	2

discrepancy_index = 1

last_occurrence(C) = -1

good_suffix(1) = -2

Good Suffix umožňuje skok (1) - (-2) = 3. Last Occurrence umožňuje skok (1) - (-1) = 2. Algoritmus vyberá Good Suffix.



i	Suffix Location
0	-3
1	-2
2	1

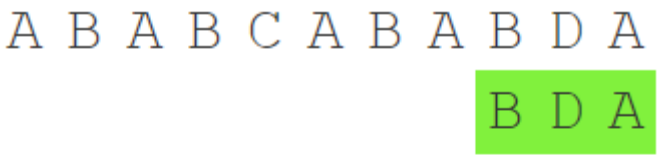
char	Last Occurrence
B	0
D	1
A	2

discrepancy_index = 2

last_occurrence(B) = 0

good_suffix(2) = 1

Good Suffix umožňuje skok (2) - (1) = 1. Last Occurrence umožňuje skok (2) - (0) = 2. Algoritmus vyberá Good Suffix.



i	Suffix Location
0	-3
1	-2
2	1

char	Last Occurrence
B	0
D	1
A	2

Obrázok č. 8: Vizualizácia algoritmu Boyer-Moore [Zdroj: Vlastné spracovanie]

Algoritmus Boyer-Moore sa v praxi ukázal ako veľmi efektívny. Z jeho vlastností vyplýva, že čím dlhší je hľadaný vzor, tým efektívnejší je celý algoritmus. Jeho nevýhodou je, že vyžaduje viac pamäte než ostatné algoritmy. Priemerná efektívnosť algoritmu je lineárna, t.j. O(n). V najlepšom prípade je vďaka svojim vlastnostiam lepšia ako lineárna, a to O(n/m).

2.5 Two-way string-matching algorithm

Tento algoritmus je kombinácia Knuth-Morris-Pratt algoritmu a algoritmu Boyer–Moore. Bol predstavený v roku 1991 Maximom Crochemorom a Dominiquem Perrinom a okamžite sa stal využívaný v mnohých systémoch.

Najväčšou výhodou tohto algoritmu je, že napríklad na rozdiel od KMP algoritmu potrebuje na pomocnú tabuľku iba konštantnú pamäť a má vynikajúcu efektívnosť v bežnej praxi.

Aj napriek jeho zložitejšej implementácií oproti iným algoritmom, sa tento algoritmus skoro automaticky stal štandardom na vyhľadávanie vzoru v texte a dnes sa využíva napríklad aj v knižničnej funkcii *strstr* jazyka C.

Časová zložitosť pripravenia tabuľky je lineárna a lineárne je aj samotné vyhľadávanie. Spolu je teda časová zložitosť celého algoritmu O(n+m). [14] [15]

```
function MATCH (x, t):
  n ← LENGTH(x);
  (l1, p1) ← MAXIMAL-SUFFIX(x, ≤);
  (l2, p2) ← MAXIMAL-SUFFIX(x, ⊆);
  if (l1 ≥ l2) then {
    l ← l1; p ← p1;
  } else {
    l ← l2; p ← p2;
  } end if
  if (l < n/2 and x[1] ··· x[l] is a suffix of x[l + 1] ··· x[l + p]) then {
    P := ∅; pos ← 0; s ← 0;
    while (pos + n ≤ |t|) do {
      i ← max(l, s) + 1;
      while (i ≤ n and x[i] = t[pos + i]) do i ← i + 1;
      if (i ≤ n) then {
        pos ← pos + max(i − l, s − p + 1);
        s ← 0;
      } else {
            j ← l;
            while (j > s and x[j] = t[pos + j]) do j ← j − 1;
            if (j ≤ s) then add pos to P;
            pos ← pos + p;
            s ← n − p;
      } end if
    } end while
    return (P);
  } else {
    q := max(l, n − l) + 1;
    P := ∅; pos ← 0;
    while (pos + n ≤ |t|) do {
      i ← l + 1;
      while (i ≤ n and x[i] = t[pos + i]) do i ← i + 1;
      if (i ≤ n) then {
        pos ← pos + i − l;
      } else {
        j ← l;
        while (j > 0 and x[j] = t[pos + j]) do j ← j − 1;
        if (j = 0) then add pos to P;
        pos ← pos + q,
      } end if
    } end while
    return (P);
  }end if
end function.
```

Zdrojový kód č. 9: Pseudokód Two-way string matching algoritmu [Zdroj: 14]

2.6 Iné

Samozrejme, existuje veľké množstvo iných algoritmov, ktoré sa líšia vo svojich špecifikáciách, v ich implementačných vlastnostiach a v ich časovej, či pamäťovej zložitosti. Problém hľadania vzoru v texte sa neustále vyvíja, pretože text je najčastejšia forma ukladania informácií a má svoje špecifické vlastnosti, ako napríklad text, ktorý obsahuje DNA údaje, obsahuje iba určité symboly, slová majú špecifickú dĺžku a podobne. Preto je dôležité uvedomiť si pri výbere algoritmu jeho výhody a nevýhody.

Dodnes vznikajú takéto algoritmy, či už navrhnutím na mieru, alebo kombináciou vlastností starších, dobre známych algoritmov.

3 Vyhľadávacia funkcia strstr

Funkcia *strstr* je knižničná funkcia jazyka C, ktorá lokalizuje prvý výskyt hľadaného reťazca v texte. Jej predpis určuje dva potrebné parametre, a to ukazovateľ na text a ukazovateľ na hľadané slovo. Funkcia v prípade úspešného nájdenia slova v texte vracia ukazovateľ na prvý výskyt tohto slova. V prípade, že hľadanie nie je úspešné, funkcia vracia prázdny ukazovateľ (*NULL*). Táto funkcia je deklarovaná v hlavičkovom súbore *string.h*.

```
char *strstr(const char *pattern, const char *text);
```

Zdrojový kód č. 10: Deklarácia funkcie strstr [Zdroj: 16]

Súčasná implementácia využíva Two-way string-matching algoritmus, ale ešte pred samotným vyhľadávaním funkcia analyzuje prehľadávaný text a vzor, a tým ošetruje špeciálne prípady, ktoré môžu celkové hľadanie urýchliť.

Funkcia najskôr zisťuje, či nie je hľadaný prázdny reťazec. V takom prípade ukončí funkciu a vráti ukazovateľ na prvú pozíciu v prehľadávanom texte. Ďalej funkcia upraví prehľadávaný text tak, že nájde prvú pozíciu prvého symbolu hľadeného slova v hľadanom texte a následné vytvorí kópiu prehľadávaného textu bez začiatku, kde sa nenachádza daný symbol a s týmto textom ďalej pracuje. Vďaka tomuto lineárnemu prehľadávaniu algoritmus funkcie nemusí prehľadávať časť textu, v ktorom sa takýto symbol ani nenachádza, a teda je samozrejmé, že sa v ňom nenachádza ani hľadané slovo. Ďalšia kontrola zisťuje či je dĺžka prehľadávaného textu väčšia ako hľadané slovo. Posledná kontrola pred samotným Two-way algoritmom zistí, či sa takto upravený prehľadávaný text nezhoduje s hľadaným slovom

pomocou funkcie *memcmp*. Ak podmienka splní požiadavky, nie je potrebné, aby bol inicializovaný Two-way algoritmus a celá funkcia je tak efektívnejšia.

```
/* Return the first occurrence of NEEDLE in HAYSTACK.  Return HAYSTACK
   if NEEDLE is empty, otherwise NULL if NEEDLE is not found in
   HAYSTACK.  */
char *
STRSTR (const char *haystack, const char *needle)
{
        size_t needle_len; /* Length of NEEDLE.  */
        size_t haystack_len; /* Known minimum length of HAYSTACK.  */

  /* Handle empty NEEDLE special case.  */
        if (needle[0] == '\0')
                return (char *) haystack;

  /* Skip until we find the first matching char from NEEDLE.  */
        haystack = strchr (haystack, needle[0]);
        if (haystack == NULL || needle[1] == '\0')
                return (char *) haystack;

  /* Ensure HAYSTACK length is at least as long as NEEDLE length.
     Since a match may occur early on in a huge HAYSTACK, use strnlen
     and read ahead a few cachelines for improved performance.  */
        needle_len = strlen (needle);
        haystack_len = __strnlen (haystack, needle_len + 256);
        if (haystack_len < needle_len)
                return NULL;

  /* Check whether we have a match.  This improves performance since we avoid
     the initialization overhead of the two-way algorithm.  */
  if (memcmp (haystack, needle, needle_len) == 0)
        return (char *) haystack;

  /* Perform the search.  Abstract memory is considered to be an array
     of 'unsigned char' values, not an array of 'char' values.  See
     ISO C 99 section 6.2.6.1.  */
  if (needle_len < LONG_NEEDLE_THRESHOLD)
        return two_way_short_needle ((const unsigned char *) haystack,
                                     haystack_len,
                                     (const unsigned char *) needle, needle_len);
  return two_way_long_needle ((const unsigned char *) haystack, haystack_len,
                              (const unsigned char *) needle, needle_len);
}
```

Zdrojový kód č. 11: Implementácia GNU C knižničnej funkcie strstr [Zdroj: 16]

Funkcia *strstr* je medzi programátormi veľmi známa. Toto pomenovanie funkcie je používané aj pri tvorbe vlastnej implementácie a vďaka tomu je zaručená dobrá čitateľnosť a zrozumiteľnosť zdrojového kódu.

4 Metodika práce a metódy skúmania

Metodika práce bola realizovaná vytvorením programu v programovacom jazyku C++ na porovnanie exekučnej efektívnosti vyhľadávacieho algoritmu hrubá sila a knižničnej vyhľadávacej funkcie *strstr*.

Testovanie bolo vykonané na notebooku ASUS X555LJ s nasledujúcimi parametrami:

- Procesor: Intel(R) Core(TM) i3-5010U CPU @ 2.10GHz 2.10GHz
- RAM: 8.00 GB
- Typ systému: 64-bitový operačný systém, Windows 10

Projekt bol vytvorený v prostredí programu Visual Studio 2017, v ktorom bol vytvorený projekt MFC podľa odporúčanej šablóny. Tento projekt bol kompilovaný za pomoci vstavaného kompilátoru programu Visual Studio 2017.

5 Cieľ práce

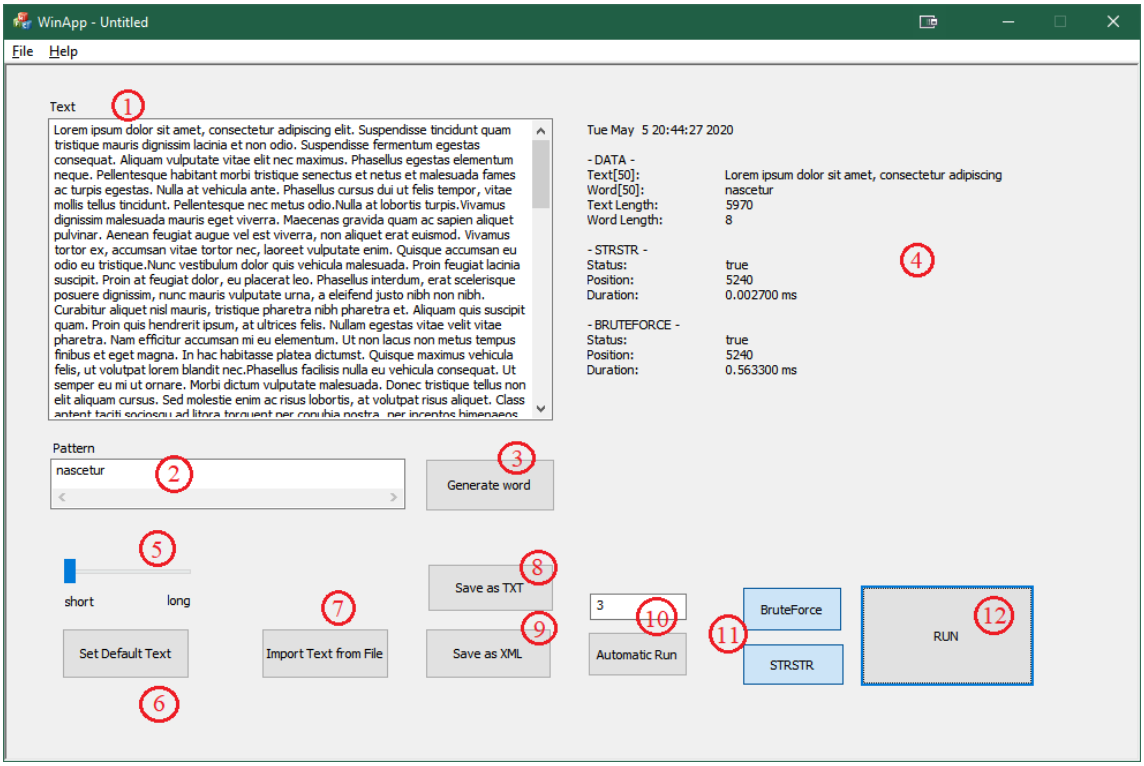
Diplomová práca je rozdelená do viacerých častí, pričom praktická časť diplomovej práce sa zameriava na opis vytvoreného programu v jazyku C++, ktorý porovnáva exekučný čas vyhľadávacieho algoritmu hrubá sila a knižničnej vyhľadávacej funkcie *strstr*, ktoré vyhľadávajú zvolený vzor v zadanom texte. Pri spustení vyhľadávania je spustená funkcia *strstr* aj funkcia vyhľadávacieho algoritmu hrubá sila s identickým vyhľadávaným vzorom a prehľadávaným textom, a teda je zaručené, že majú rovnaké podmienky. Počas porovnávania je meraný exekučný čas jednotlivých funkcií a na základe tohto merania vieme porovnať efektívnosť týchto dvoch vyhľadávacích algoritmov.

Ďalšie časti diplomovej práce budú zamerané na opis programu a výsledky merania exekučných časov.

6 Opis programu

Navrhnutý program na porovnávanie exekučnej efektívnosti vyhľadávacieho algoritmu hrubá sila a vyhľadávacej funkcie *strstr* je vytvorený v programovacom jazyku C++ za pomoci programu Visual Studio 2017. Tento program ponúka na výber preddefinované šablóny zdrojového kódu podľa platformy a cieľa programovaného projektu. Tie pomáhajú pri tvorbe programu a urýchľujú vytvorenie spustiteľného zdrojového kódu. V tomto projekte je využitá MFC šablóna, ktorá poskytuje využívanie WIN32 API pre programy s grafickým rozhraním v operačnom systéme Microsoft Windows. Po vytvorení projektu sú vygenerované potrebné triedy, ako napríklad *CFrameWnd, CDocument, CView, CWinApp,* ktoré po skompilovaní vytvoria prázdny, ale funkčný program. MFC taktiež podporuje tvorbu programov za pomoci dialógových okien, ktoré pomáhajú programátorom s vizuálnym návrhom pomocou systému „drag and drop“, ktorý je využitý aj v našom programe. MFC projekt poskytuje vlastné dátové typy, ako napríklad *CString*.

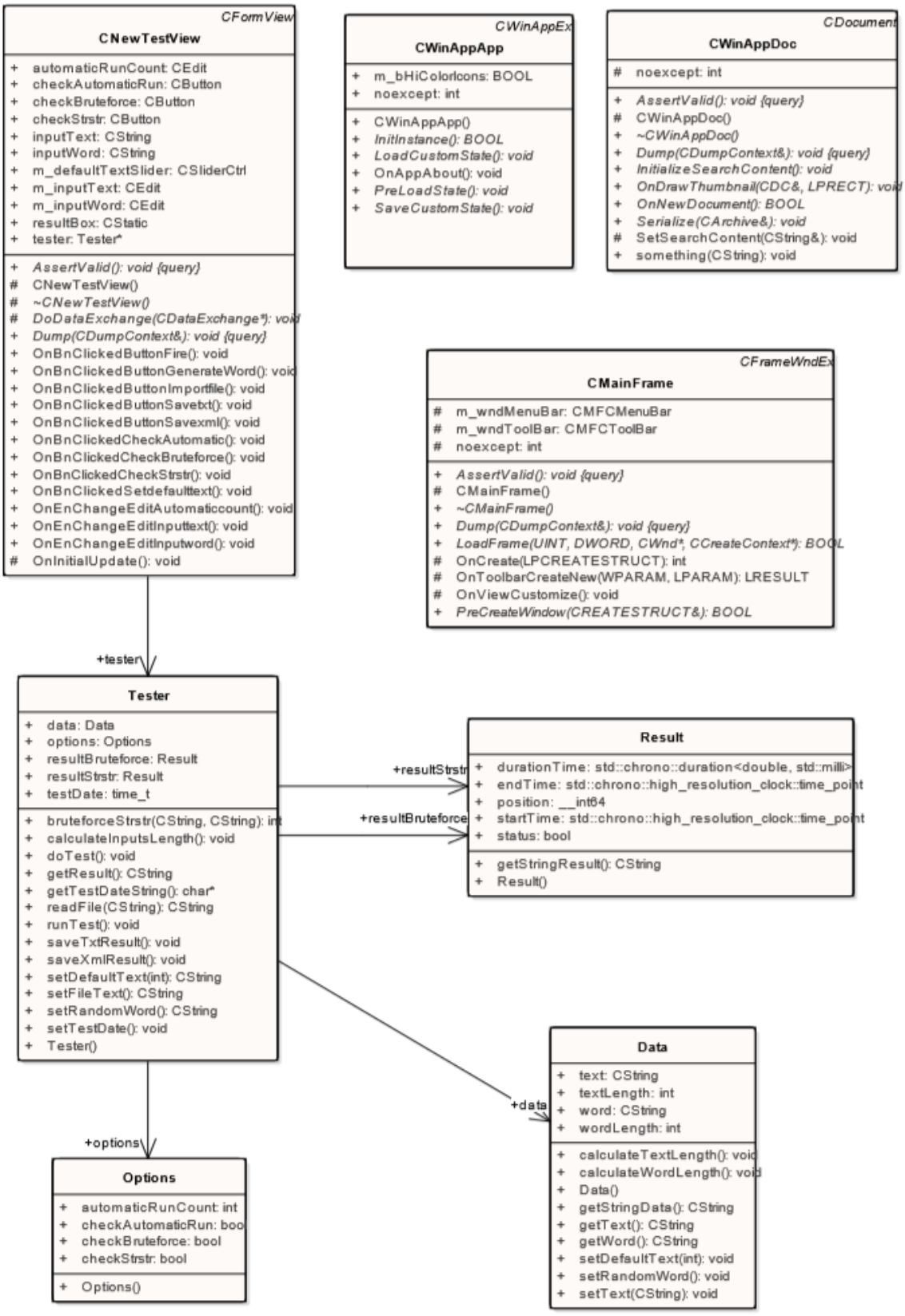
Používateľské rozhranie navrhnutého programu pozostáva z viacerých ovládacích prvkov, prevažne tlačidiel a textových polí, pomocou ktorých používateľ spúšťa udalosti na konfigurovanie a spustenie testovania.



Obrázok č. 9: Mapa funkcií programu [Zdroj: Vlastné spracovanie]

1. **Text**: Pole, do ktorého používateľ vkladá text, v ktoré bude prehľadávané
2. **Vzor**: Pole, do ktorého používateľ vkladá slovo, ktoré bude hľadané v texte
3. **Generuj vzor**: Tlačidlo na vygenerovanie náhodného vzoru z textu
4. **Výsledok**: Slúži na zobrazenie výsledkov testu
5. **Nastavenie dĺžky základného textu**: Slúži na ovládanie, aký dlhý základný text má byť vygenerovaný do poľa s textom
6. **Nastav základný text**: Tlačidlo na vygenerovanie základného textu podľa zvolenej dĺžky
7. **Import textu zo súboru**: Tlačidlo na importovanie textu z textového súboru
8. **Uložiť ako txt súbor**: Tlačidlo na uloženie výsledkov testu do textového súboru result.txt
9. **Uložiť ako xml súbor**: Tlačidlo na uloženie výsledkov testu do xml súboru result.xml
10. **Automatický beh**: Tlačidlo na spustenie viacerých náhodných testov naraz
11. **Algoritmy**: Tlačidlá na výber, ktorý algoritmus má byť testovaný
12. **Štart**: Tlačidlo na spustenie testu

V projekte sú taktiež vytvorené vlastné triedy, ktoré zapuzdrujú dáta a manipulujú s nimi. Takto rozdelený zdrojový kód je veľmi prehľadný a ľahko upravovateľný. Hlavná trieda *Tester* slúži na ovládanie vykonávaných testov. S touto triedou komunikuje trieda *CNewTestView*, ktorá obsahuje metódy udalostí, ako napríklad *OnBnClickedButtonGenerateWord*, ktorá je spustená po stlačení tlačidla na generovanie náhodného slova. Členská funkcia *setRandomWord* triedy *Tester* následne vygeneruje náhodné slovo z textu a to sa nastaví v aplikácii do textového poľa *vzor*. Obdobne funguje aj zvyšná funkcionalita triedy *Tester*. Trieda *Tester* taktiež využíva triedu *Data*, ktorá slúži na zapuzdrenie prehľadávaného textu a vyhľadávaného vzoru. Ďalej je v projekte trieda *Options*, ktorá uchováva používateľské nastavenia testu a trieda *Result*, ktorá obsahuje členské premenné, v ktorých sa nachádzajú výsledky testovania.



Obrázok č. 10: Diagram tried programu [Zdroj: Vlastné spracovanie]

Nasledujúce členské funkcie z triedy *CNewTestView* sú spúšťané po interakcii používateľa s programom. Napríklad funkcia *OnBnClickedCheckBruteforce* je prepojená s označovacím tlačidlom *BruteForce* a nastavuje, či má byť pri teste spustený algoritmus hrubá sila.

```
void CNewTestView::OnBnClickedCheckBruteforce()
{
	if (checkBruteforce.GetCheck())
		tester->options.checkBruteforce = true;
	else
		tester->options.checkBruteforce = false;
}
```

Zdrojový kód č. 12: Funkcia *OnBnClickedCheckBruteforce* [Zdroj: Vlastné spracovanie]

Nasledujúca funkcia je spustená, ak používateľ stlačí tlačidlo na uloženie výsledku testu do súboru XML, ktorý sa nachádza v priečinku pri spustiteľnom programe.

```
void CNewTestView::OnBnClickedButtonSavexml()
{
	tester->saveXmlResult();
	AfxMessageBox(L"Result has been saved to result.xml");
}
```

Zdrojový kód č. 13: Funkcia *OnBnClickedButtonSavexml* [Zdroj: Vlastné spracovanie]

Ako bolo spomínané, po stlačení tlačidla na generovanie náhodného vzoru je pomocou funkcie *setRandomWord* triedy *Tester* toto slovo vygenerované a uložené do premennej triedy *Data* a to je následne zobrazené v grafickom rozhraní.

```
void CNewTestView::OnBnClickedButtonGenerateWord()
{
	m_inputWord.SetWindowText(tester->setRandomWord());
}
```

Zdrojový kód č. 14: Funkcia *OnBnClickedButtonGenerateWord*

[Zdroj: Vlastné spracovanie]

Pri spustení automatického testovania je dôležité generovanie rôznych vzorov z prehľadávaného textu. Nasledujúca funkcia najskôr rozdelí text na jednotlivé slová a potom vyberie jedno pomocou knižničnej funkcie jazyka C *rand.*

```
void Data::setRandomWord()
{
	CArray<CString, CString> arrWords;
	CString field;
	int index = 0;
	int randomPosition;

	// posledný argument znak na základe ktorého oddeľuje slová
	while (AfxExtractSubString(field, text, index, _T(' ')))
	{
		arrWords.Add(field);
		++index;
	}
	randomPosition = rand() % index;
	word = arrWords[randomPosition];
}
```

Zdrojový kód č. 15: Funkcia *setRandomWord* [Zdroj: Vlastné spracovanie]

Spustenie testu je možné v dvoch režimoch, jednotlivé testovanie, ktoré je opísané neskôr v tejto práci, a automatické spustenie testov. Pri automatickom teste je potrebné nastaviť koľkokrát sa má test spustiť a program potom sám generuje hľadaný vzor a výsledok testovania uloží do súborov *result.txt* a *result.xml*.

```
void Tester::runTest()
{
	if (options.checkAutomaticRun) {
		for (int i = 0; i < options.automaticRunCount; i++) {
			data.setRandomWord();
			doTest();
			saveXmlResult();
			saveTxtResult();
		}
	}
	else {
		doTest();
	}

}
```

Zdrojový kód č. 16: Funkcia *runTest* [Zdroj: Vlastné spracovanie]

Samotné jednotlivé testovanie najskôr skontroluje, ktoré algoritmy majú byť spustené. Potom pomocou knižničných funkcií *std::chrono::high_resolution_clock::now* spustí meranie exekučného času jednotlivých algoritmov. Výsledok hľadania uloží do premennej *position*. Funkcia algoritmu hrubá sila je implementovaná tak, že ak sa vzor v danom texte

nenájde, vracia hodnotu *-1*. Funkcia *strstr* v tomto prípade vracia prázdny ukazovateľ na dátový typ *char*. Rozdiel v tejto implementácii nespôsobuje žiadne zníženie efektívnosti. Ak sa algoritmom podarí nájsť hľadaný vzor, funkcia nastaví premennú *status* na hodnotu *true.*

```
void Tester::doTest()
{
	resultStrstr.status = false;
	resultBruteforce.status = false;

	data.calculateTextLength();
	data.calculateWordLength();

	if (options.checkBruteforce){
		resultBruteforce.startTime = std::chrono::high_resolution_clock::now();
		resultBruteforce.position = bruteforceStrstr(data.text, data.word);
		resultBruteforce.endTime = std::chrono::high_resolution_clock::now();
		resultBruteforce.durationTime = resultBruteforce.endTime -
resultBruteforce.startTime;
		if (resultBruteforce.position != -1) {
			//AfxMessageBox(_T("Inside BruteForce"));
			resultBruteforce.status = true;
		}
	}
	if (options.checkStrstr) {
		const char* char_text;
		const char* char_word;
		const char* char_position;
		USES_CONVERSION;

		char_text = T2A((LPCTSTR)data.text);
		char_word = T2A((LPCTSTR)data.word);

		resultStrstr.startTime = std::chrono::high_resolution_clock::now();
		char_position = strstr(char_text, char_word);
		resultStrstr.endTime = std::chrono::high_resolution_clock::now();
		resultStrstr.durationTime = resultStrstr.endTime - resultStrstr.startTime;
		if (char_position != NULL) {
			//AfxMessageBox(_T("Insidee STRSTR"));
			resultStrstr.position = char_position - char_text;
			resultStrstr.status = true;
		}
		else {
			resultStrstr.position = -1;
		}
	}
}
```

Zdrojový kód č. 17: Funkcia *doTest* [Zdroj: Vlastné spracovanie]

Algoritmus hrubá sila je v tomto projekte implementovaný s využitím dátového typu MFC *CString*, ktorý v podstate len zapuzdruje klasický ukazovateľ na dátový typ *char*. Tento rozdiel nijako výrazne nevplýva na efektívnosť algoritmu. Algoritmus postupne prechádza jednotlivé symboly, ktoré porovnáva s vyhľadávaným vzorom. Návratový typ funkcie je dátový typ *int*, a teda pozícia indexu v prehľadávanom texte.

```
int Tester::bruteforceStrstr(CString text, CString word) {

	int wordLength = word.GetLength();
	int textLength = text.GetLength();

	for (int i = 0; i <= textLength - wordLength; i++) {
		int j;
		for (j = 0; j < wordLength; j++)
			if (text[i + j] != word[j])
				break;
		if (j == wordLength)
			return i;
	}
	return -1;
}
```

Zdrojový kód č. 18: Funkcia *bruteforceStrstr* [Zdroj: Vlastné spracovanie]

7 Výsledky práce

Výsledky testovania sú zobrazené používateľovi ihneď po testovaní v časti *Výsledok*. Taktiež ich je možné uložiť do textového alebo XML súboru a pracovať s nimi neskôr. Následný obrázok ilustruje súbor *result.txt*, v ktorom je zaznamenaný dátum spustenia testu, ukážka prehľadávaného textu, ukážka vyhľadávaného vzoru a ich dĺžka. Po týchto údajoch sa nachádzajú samotné výsledky testu, a to exekučný čas testu, či bolo hľadanie úspešné a samotná pozícia vzoru v texte. Výsledky každého nového testu sú pripísané na koniec tohto súboru.

```
---------------------------------
Tue May  5 23:54:05 2020
- DATA -
Text[50]:          Lorem ipsum dolor sit amet, consectetur adipiscing
Word[50]:          facilisis
Text Length:       29844
Word Length:       9
- STRSTR -
Status:            true
Position:          1382
Duration:          0.001100 ms
- BRUTEFORCE -
Status:            true
Position:          1382
Duration:          0.143100 ms
---------------------------------
Tue May  5 23:54:05 2020
- DATA -
Text[50]:          Lorem ipsum dolor sit amet, consectetur adipiscing
Word[50]:          nisi
Text Length:       29844
Word Length:       4
- STRSTR -
Status:            true
Position:          2776
Duration:          0.001700 ms
- BRUTEFORCE -
Status:            true
Position:          2776
Duration:          0.299100 ms
---------------------------------
```

Obrázok č. 11: Výstup programu do textového súboru [Zdroj: Vlastné spracovanie]

V prípade ak sa používateľ rozhodne uložiť výsledky do súboru XML, program využíva verejne dostupnú knižnicu TinyXML [17], pomocou ktorej pripíše výsledky nového testu do súboru *result.xml*. Štruktúra tohto súboru je znázornená na nasledujúcom obrázku.

```
<?xml version="1.0" ?>
<testList>
  <test>
    <date>Tue May 5 23:54:05 2020&#x0A;</date>
    <data>
      <text>Lorem ipsum dolor sit amet, consectetur adipiscing</text>
      <word>facilisis</word>
      <textLenght>29844</textLenght>
      <wordLenght>9</wordLenght>
    </data>
    <results>
      <result>
        <type>strstr</type>
        <status>true</status>
        <position>1382</position>
        <duration>0.001100</duration>
      </result>
      <result>
        <type>bruteforce</type>
        <status>true</status>
        <position>1382</position>
        <duration>0.143100</duration>
      </result>
    </results>
  </test>
</testList>
```

Obrázok č. 12: Výstup programu do xml súboru [Zdroj: Vlastné spracovanie]

Na získanie údajov na porovnanie algoritmu hrubá sila a vyhľadávacej funkcie *strstr* boli spustené viaceré variácie testov. Všetky z nich ukázali to, čo sa predpokladalo, a to, že knižničná funkcia *strstr* je omnoho efektívnejšia a algoritmus hrubá sila má niekoľkokrát horší exekučný čas. Je potrebné upozorniť, že všetky grafy sú logaritmické.

Prvý test pozostával celkovo z 5000 spustení, kedy meniaci sa parameter bol počet symbolov prehľadávaného textu. Tento text bol nastavený vďaka funkcii programu *Nastav základný text*, ktorý môže dosahovať 5 rozdielnych dĺžok (5974 symbolov, 11867 symbolov, 17865 symbolov, 23855 symbolov a 29848 symbolov). Pri každej dĺžke textu bol nastavený automatický test na 1000 opakovaní. Program pri jednotlivých textoch menil vyhľadávaný vzor, ktorý bol vygenerovaný z celého rozsahu prehľadávaného textu.

Po spracovaní údajov je možné vidieť ako je knižničná funkcia *strstr* približne 150-krát efektívnejšia. Priemerný čas vyhľadávania je v prípade knižničnej funkcie *strstr* 1,612 nanosekundy a v prípade algoritmu hrubá sila 253,263 nanosekundy. S rastúcim počtom symbolov je možné pozorovať mierne zhoršenie exekučného času pri oboch funkciách, na ktoré sa pozrieme v ďalšom teste.

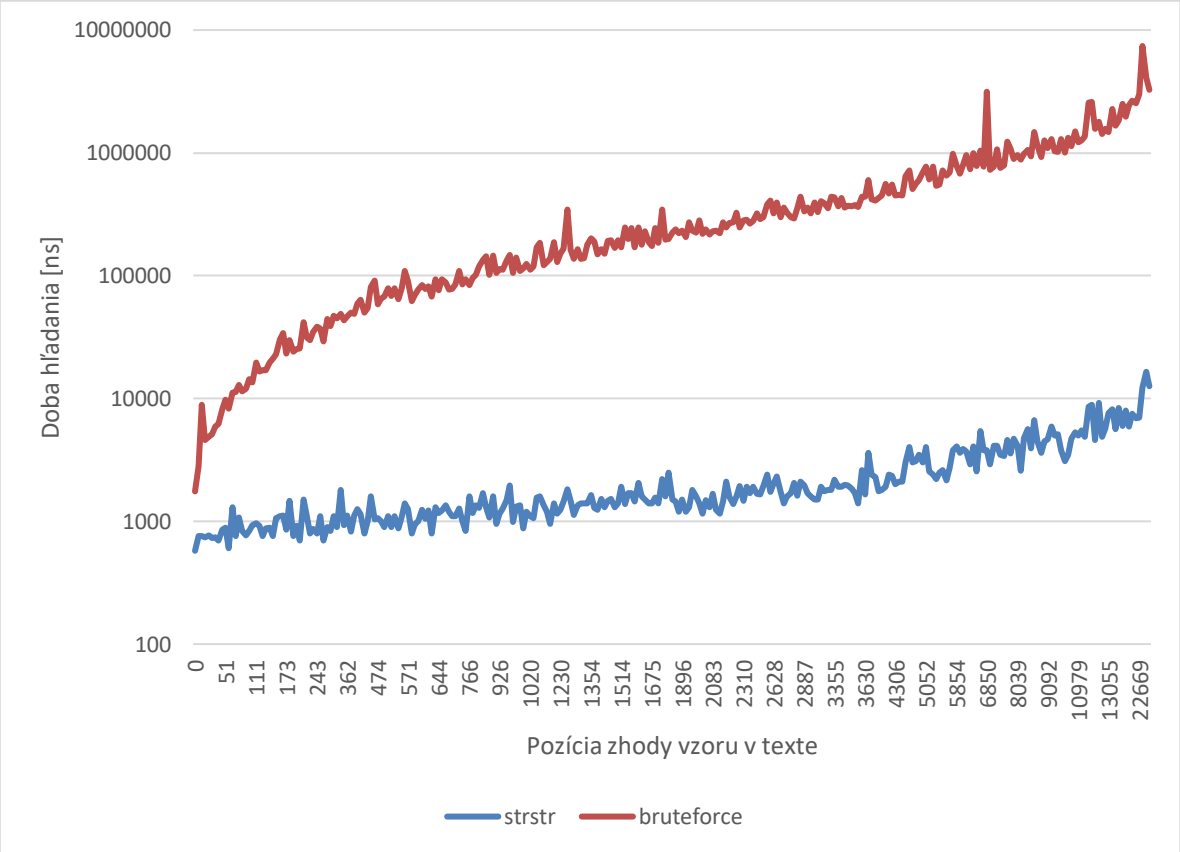
Graf č. 1: Porovnanie priemerných exekučných časov [Zdroj: Vlastné spracovanie]

	BRUTEFORCE	STRSTR
5974 SYMBOLOV	0,176576 ms	0,001281 ms
11867 SYMBOLOV	0,229945 ms	0,001574 ms
17865 SYMBOLOV	0,286233 ms	0,001778 ms
23855 SYMBOLOV	0,297837 ms	0,001755 ms
29848 SYMBOLOV	0,275728 ms	0,001671 ms

Tabuľka č. 5: Porovnanie priemerných časov [Zdroj: Vlastné spracovanie]

Ďalšie testovanie pozostávalo z vyjadrenia ako sa zhoršuje exekučný čas funkcií podľa pozície vzoru v hľadanom texte. Na analýzu tejto skutočnosti boli použité výsledky z testovania, kde bola dĺžka prehľadávaného textu 29848 symbolov.

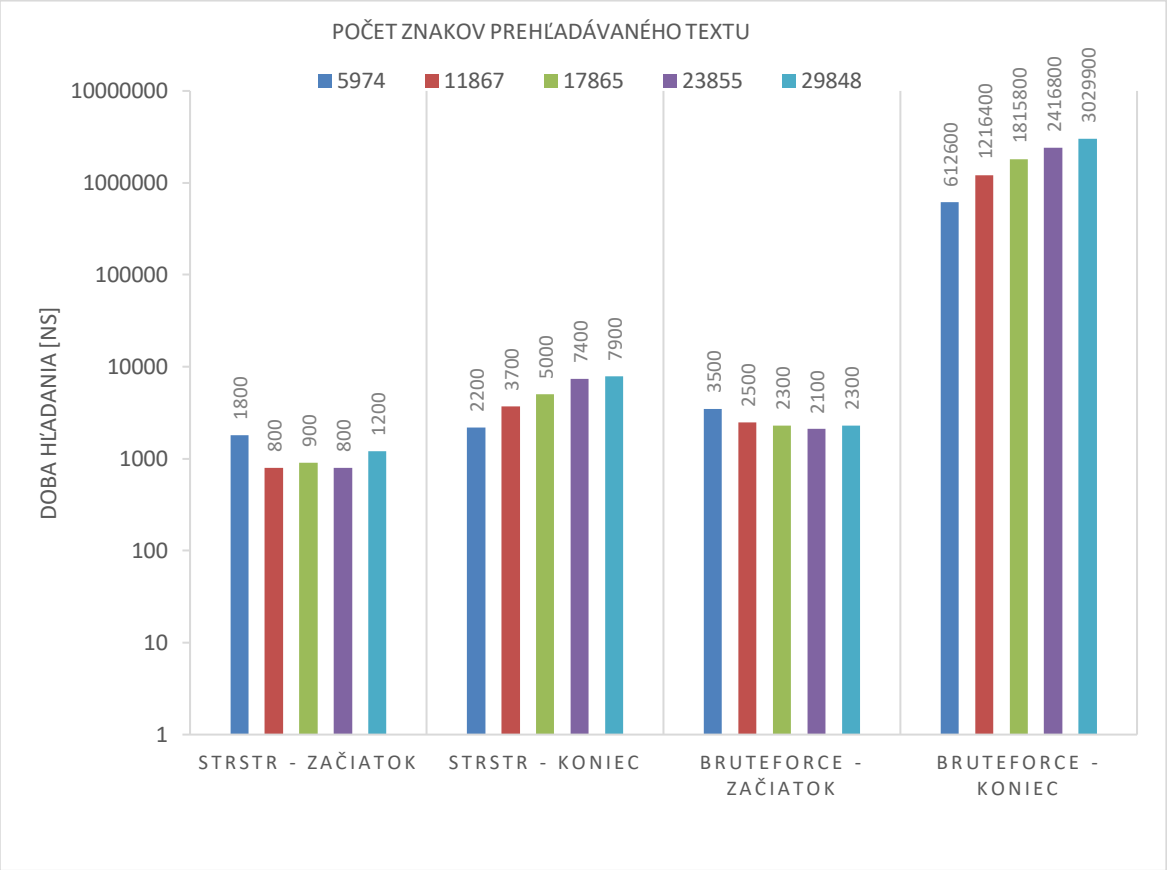
Z výsledkov vyplýva, že knižničná funkcia *strstr* vo všetkých prípadoch dosahuje lepšiu exekučnú efektívnosť a má menší exekučný čas. Ďalej je z grafu možné odčítať, že pri oboch funkciách rastie exekučný čas podľa pozície vyhľadávaného vzoru. V prípade knižničnej funkcie *strstr* je ale tento nárast oveľa pomalší oproti funkcii hrubá sila. Ak sa vzor nachádza napríklad na pozícii 10979, funkcia hrubá sila mala exekučný čas 1,2148 ms a *strstr* iba 0,005 ms.



Graf č. 2: Porovnanie exekučného času odvíjajúceho sa od pozície hľadaného vzoru [Zdroj: Vlastné spracovanie]

Posledný test pozostával celkovo z 10 jednotlivých manuálnych spustení. Cieľom testovania bolo ukázať rozdiel exekučného času vyhľadávania, keď sa vzor nachádzal na začiatku alebo na konci prehľadávaného textu, pri zvyšujúcom sa počte znakov prehľadávaného textu.

Z výsledkov je možné pozorovať, že rozdiel exekučného času, keď sa vzor nachádzal na začiatku nie je medzi algoritmom hrubá sila a funkciou *strstr* až taký výrazný. Na rozdiel od toho, ak sa vzor nachádzal na konci prehľadávaného textu, exekučný čas algoritmu hrubá sila dosahuje neporovnateľne horšiu efektívnosť. Z nasledujúceho grafu je taktiež možné vidieť, ako sa zhoršuje efektívnosť oboch funkcií s rastúcim počtom symbolov. V prípade algoritmu hrubá sila je tento rozdiel ale podstatne výraznejší, kde je napríklad exekučný čas pri počte 29848 symbolov oproti 5974 symbolov približne 5-krát dlhší.



Graf č. 3: Porovnanie exekučného času podľa pozície hľadaného vzoru [Zdroj: Vlastné spracovanie]

	STRSTR - ZAČIATOK	BRUTEFORCE - ZAČIATOK	STRSTR - KONIEC	BRUTEFORCE - KONIEC
5974 SYMBOLOV	0,0018 ms	0,0035 ms	0,0022 ms	0,6126 ms
11867 SYMBOLOV	0,0008 ms	0,0025 ms	0,0037 ms	1,2164 ms
17865 SYMBOLOV	0,0009 ms	0,0023 ms	0,005 ms	1,8158 ms
23855 SYMBOLOV	0,0008 ms	0,0021 ms	0,0074 ms	2,4168 ms
29848 SYMBOLOV	0,0012 ms	0,0023 ms	0,0079 ms	3,0299 ms

Tabuľka č. 6: Porovnanie exekučného času podľa pozície hľadaného vzoru [Zdroj: Vlastné spracovanie]

Záver

Cieľom tejto práce bol všeobecný prehľad algoritmov vyhľadávajúcich reťazec vo väčšom texte, ktoré pracujú na rôznom princípe. Pri algoritmoch bola opísaná ich časová zložitosť, pamäťová zložitosť a tak isto boli spomenuté ich najväčšie výhody. Práca bola bližšie zameraná na vyhľadávací algoritmus hrubá sila a knižničnú funkciu *strstr* a porovnanie ich exekučnej efektívnosti pri hľadaní v niekoľko tisíc znakovom reťazci v programe napísanom v programovacom jazyku C++. Algoritmy boli porovnávané najmä na základe priemerného exekučného času, potrebného na vyhľadanie vzoru, ktorý bol náhodne generovaný. Z jednotlivých výsledkov testovania je zrejmé, že knižničná funkcia *strstr* je omnoho efektívnejšia. Ďalej sa podarilo znázorniť, ako sa zhoršuje efektívnosť algoritmu hrubá sila so zväčšujúcim sa prehľadávaným textom.

Program spĺňa všetky ciele, ktoré boli vytýčené na začiatku tejto práce. Ako jeho ďalšie rozšírenie je možné ľahko pridať iné vyhľadávacie algoritmy. Taktiež by bolo vhodné doplniť do programu vizualizáciu vyhľadávania pre lepšiu znázornenie rozdielov medzi použitými vyhľadávacími algoritmami.

Zoznam použitej literatúry

[1] GALIL, A. A. Z. 1997. *Pattern Matching Algorithms*. Oxford University Press. New York, Oxford. 1997 ISBN 0-19-511367-5

[2] KNUTH, D. E. MORRIS Jr. J. H., PRATT, V. R. 1974. *Fast Pattern Matching in Strings*. Society for Industrial and Applied Mathematics. 2006. ISSN 1095-7111

[3] http://www2.fiit.stuba.sk/~pospichal/soltis/uvod.htm 2020

[4] GOYVAERTS, J. 2006. *Regular Expressions: The Complete Tutorial*. 2007

[5] MOKADEM, R. LITWIN, W. 2007. *Fast nGramBased String Search Over Data Encoded Using Algebraic Signatures*. 33rd International Conference on Very Large Data Bases

[6] NAVARRO, G. RAFFINOT, M. 2008. *Flexible Pattern Matching Strings: Practical On-Line Search Algorithms for Texts and Biological Sequences*, ISBN 978-0-521-03993-2

[7] https://www.csd.uwo.ca/courses/CS1027b/notes/CS1027-017-Analysis-W12.pdf 2020

[8] http://www2.fiit.stuba.sk/~pospichal/soltis/algo511.htm 2020

[9] GARG, N. SINGHAL, S. 2018 *Analysis and Design of Algorithms: A Beginner's Hope*. BPB Publication. 2018. ISBN 978-93-8655-189-4

[10] https://www.cs.purdue.edu/homes/ayg/CS251/slides/chap11.pdf 2020

[11] http://www2.fiit.stuba.sk/~pospichal/soltis/algo512.htm 2020

[12] BAEZA-YATES, R. 1989. *Algorithms for string searching*. University of Waterloo.

[13] http://www2.fiit.stuba.sk/~pospichal/soltis/algo513.htm 2020

[14] CROCHEMORE, M. PERRIN, D. *Two-Way String-Matching*. Université Paris. 1991

[15] http://www-igm.univ-mlv.fr/~lecroq/string/node26.html 2020

[16] https://github.com/lattera/glibc 2020

[17] http://grinninglizard.com/tinyxml/ 2020

Prílohy

Príloha č.1

Zoznam obrázkov

Zoznam tabuliek

Zoznam grafov

Zoznam zdrojového kódu

Zdrojový kód č. 18: Funkcia *bruteforceStrstr* [Zdroj: Vlastné spracovanie]

Príloha č.2

Zoznam textových súborov

Textový súbor č. 1 - result.txt

Textový súbor č. 2 - result.xml

Textový súbor č. 3 - result_1.txt

Textový súbor č. 4 - result_1.xml

Textový súbor č. 5 - result_2.txt

Textový súbor č. 6 - result_2.xml

Textový súbor č. 7 - result_3.txt

Textový súbor č. 8 - result_3.xml

Textový súbor č. 9 - result_4.txt

Textový súbor č. 10 - result_4.xml

Textový súbor č. 11 - result_5.txt

Textový súbor č. 12 - result_5.xml

Textový súbor č. 13 - result_zaciatok_koniec.txt

Textový súbor č. 14 - result_zaciatok_koniec.xml

Príloha č. 3

Zdrojový kód programu

Zdrojový kód programu sa nachádza na priloženom CD v priečinku Zdrojový kód. Okrem zdrojového kódu programu sa na priloženom CD nachádza aj samotný program.

-