VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

MOŽNOSTI VYUŽITÍ SUFIXOVÝCH STROMŮ

SUFFIX TREES CAPABILITIES

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **PAVEL CHLUBNA**
AUTHOR

VEDOUCÍ PRÁCE **Ing. IVANA BURGETOVÁ, Ph.D.**
SUPERVISOR

BRNO 2020

Ústav informačních systémů (UIFS) Akademický rok 2019/2020

Zadání bakalářské práce

Student: **Chlubna Pavel**
Program: Informační technologie
Název: **Možnosti využití sufixových stromů**
Suffix Trees Capabilities
Kategorie: Algoritmy a datové struktury
Zadání:

1. Seznamte se se sufixovými stromy.
2. Seznamte se s problémy, které lze pomocí sufixových stromů efektivně řešit.
3. Po dohodě s vedoucí vyberte vhodné problémy a detailně prostudujte řešení těchto problémů pomocí sufixových stromů a dalších technik.
4. Navrhněte aplikaci, která bude demonstrovat řešení vybraných problémů různými technikami.
5. Navrženou aplikaci implementujte.
6. Aplikaci otestujte a zhodnoťte dosažené výsledky

Literatura:

- D. Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology*. New York: Cambridge University Press, 1997, 534 s., ISBN 0-521-58519-8.
- E. Ukkonen. On-line construction of suffix trees. Algorithmica, 14(3):249-260, 1995.

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 až 4.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Burgetová Ivana, Ing., Ph.D.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2019
Datum odevzdání: 28. května 2020
Datum schválení: 21. října 2019

Abstrakt

Tato práce pojednává o struktuře sufixových stromů, jejich implementaci a problematice, kterou pomocí sufixových stromů řešíme. Práce se také zaměřuje na konstrukci sufixového stromu, pomocí Ukkonenova algoritmu. Kromě samotné implementace se práce také zabývá porovnáním jiných používaných algoritmů se sufixovými stromu, z hlediska časové náročnosti. Výstupem této práce je ucelená aplikace s grafickým rozhraním, která demonstruje využití sufixových stromů při řešení různých problematik a nabízí porovnání této struktury s jinými algoritmy.

Abstract

This thesis discusses structure of suffix trees, their implementation and problematics, we can solved with suffix trees. The thesis also focuses on construction of suffix trees, with usage of Ukkonen's algorithm. Apart of implementation of this structure this thesis deals with comparing other commonly used algorithms with suffix trees in term of time complexity. A result of this thesis is application with graphical interface, that shows usage of suffix trees in solving various problematics and shows comparison of this structure to other algorithms.

Klíčová slova

Sufixový strom, Ukkonenův algoritmus, dynamické programování, algoritmy pro vyhledávání v textu, porovnávání vzorů, časová náročnost

Keywords

Suffix tree, Ukkonen's algorithm, dynamic programming, string-searching algorithm, pattern matching, time complexity

Citace

CHLUBNA, Pavel. *Možnosti využití sufixových stromů*. Brno, 2020. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Ivana Burgetová, Ph.D.

Možnosti využití sufixových stromů

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Ivany Burgetové, Ph.D. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

. . . .
Pavel Chlubna
28. května 2020

Poděkování

Rád bych tímto poděkoval Ing. Ivaně Burgetové Ph.D za vedení této práce, cenné rady k problematice, pomoc při výběru tématu a také trpělivost, kterou se mnou měla.

Obsah

1 Úvod **3**

2 Sufixový strom **4**
2.1 Struktura sufixového stromu . . . 4
2.2 Konstrukce stromu . . . 5
2.2.1 Implicitní strom a ukončovací znak . . . 6
2.3 Naivní algoritmus . . . 7
2.4 Ukkonenův algoritmus . . . 8
2.4.1 Rozšiřování stromu . . . 9
2.4.2 Snížení prostorové náročnosti . . . 10
2.4.3 Urychlení konstrukce . . . 11
2.5 Generalizovaný sufixový strom . . . 16

3 Využití sufixového stromu **18**
3.1 Vyhledávání podřetězců . . . 18
3.1.1 Sufixový strom . . . 18
3.1.2 Posuvné okno a naivní algoritmus . . . 20
3.1.3 Knuth–Morris–Pratt algoritmus . . . 22
3.1.4 Rabin-Karp algoritmus . . . 24
3.2 Nejdelší opakující se podřetězec . . . 26
3.2.1 Sufixový strom . . . 26
3.2.2 Dynamické programování . . . 28
3.3 Nejdelší společný podřetězec . . . 29
3.3.1 Sufixový strom . . . 29
3.3.2 Dynamické programování . . . 31
3.4 Nejdelší palindromický podřetězec . . . 32
3.4.1 Sufixový strom . . . 32
3.4.2 Dynamické programování . . . 33

4 Návrh aplikace **35**
4.1 Požadavky na aplikaci . . . 35
4.2 Grafické rozhraní . . . 36
4.3 Testování algoritmů . . . 36

5 Implementace sufixového stromu **38**
5.1 Využité technologie . . . 38
5.2 Vytvoření datové struktury . . . 38
5.3 Implementace Ukkonenova algoritmu . . . 40

5.4 Vyhledávání ve stromě 42
5.5 Grafické rozhraní 43

6 Měření 46
6.1 Vstupní data 46
6.2 Ukkonenův algoritmus 46
6.3 Vyhledávání podřetězce 48
6.4 Nejdelší opakující se podřetězec 51

7 Závěr 54

Literatura 55

A Výsledky měření 57

B Obsah CD 63

Kapitola 1

Úvod

Vyhledávání v datech a textu je dnes součástí většiny aplikací, ať už se jedná o klasické vyhledávání podřetězce v rozsáhlém textu, nebo hledání podobností ve více řetězcích v bioinformatice. Jelikož se dnes stále zvětšuje objem dat, která je potřeba zpracovat, musíme hledat nové postupy, jak tuto práci provést efektivně z hlediska časové i prostorové náročnosti.

Jednou z těchto efektivních možností je využití sufixových stromů, které nám umožňují řešit širokou škálu problémů nad řetězci. Tato stromová struktura v sobě uchovává všechny sufixy vstupního textu a jejich indexy. Kapitola 2 detailně pojednává o podobě této struktury, kterou navrhl Peter Weiner v roce 1972 , o tom, z čeho se skládá a jaké všechny informace nese. Následně se také zaměřuje na to, jakým způsobem se pracuje se sufixovým stromem a jak se v něm vyhledává. Důležitou součástí této struktury je také konstrukce tohoto stromu, protože chceme-li mít efektivní strukturu, musíme se snažit provést sestavení tohoto stromu v co nejlepším možném čase. Peter Weiner navrhl kromě samotné struktury také algoritmus pro jeho konstrukci, ten se ovšem v praxi moc nevyužívá. Dalším algoritmem k této struktuře přispěl Edward M. McCreight, který navrhl algoritmus pro konstrukci stromu s lineární časovou náročností. Následujícím významným milníkem této struktury bylo navržení algoritmu Esko Ukkonenem, který navrhl efektivní algoritmus s lineární časovou náročnosti. V následující kapitole se podíváme na tento algoritmus a rozebereme si ho po jednotlivých krocích.

V kapitole 3 se detailně podíváme na problematiku, která se pomocí sufixových stromů řeší a kde je řešení této problematiky v praxi důležité. Vyhledávání vzoru v textu je asi nejčastější problematika s řetězci, která se v aplikacích řeší, a pro tuto problematiku máme velké množství algoritmů. Seznámíme se s postupy, které se touto problematikou zabývají, jako je například Knuth–Morris–Pratt algoritmus, nebo Rabin–Karp algoritmus. Kromě této základní problematiky nám však sufixové stromy umožňují řešit mnohem více pokročilých problémů, které řešíme v bioinformatice, nebo při kompresi dat. Jednou z těchto problematik je například hledání nejdelšího opakujícího se podřetězce, pomocí které hledáme opakující se sekvence v DNA různých organismů, jelikož některé choroby mohou být způsobeny právě těmito opakujícími se sekvencemi. Jako příklad u komprese dat si můžeme uvést Burrows–Wheelerovu transformaci, která pro hledání opakujících se dat využívá právě sufixové stromy. Kromě sufixových stromů se také tato problematika dá řešit pomocí dynamického programování, na které se také zaměříme.

Kapitola 5 pojednává o implementaci výsledné aplikace, která byla výstupem této práce a její úlohou je demonstrovat využití sufixových stromů, a také o jejím grafickém rozhraní. Dále také uvádí jak proběhla implementace struktury sufixového stromu a jakým způsobem se pro realizaci výstupní aplikace implementoval Ukkonenův algoritmus. Kromě seznámení se s postupy v kapitole 3 je také navzájem porovnáme v kapitole 6 a budeme sledovat v jakém čase byly tyto algoritmy schopny provést danou úlohu vzhledem k velikosti vstupních dat.

Kapitola 2

Sufixový strom

Sufixový strom (Suffix tree) je stromová struktura, která je zvláštním druhem komprimované Trie. Můžeme pomocí něj velice efektivně pracovat s textovými řetězci, jelikož strom v sobě uchovává všechny sufixy vstupního textu a jejich pozice v tomto textu. Protože jednotlivé hrany mohou reprezentovat více než jeden znak, jedná se o komprimovanou Trii [4].

S konceptem sufixových stromů přišel Peter Weiner, když jej v roce 1973 prezentoval v článku Linear pattern matching algorithms [22], ačkoliv jej ve své práci nazýval ***Position tree***. V tomto roce také doktor Weiner přišel s prvním algoritmem pro konstrukci stromu s lineární časovou náročností. Po třech letech k tomuto tématu přispěl **Edward M. McCreight**, který v roce 1976 publikoval práci, ve které představil prostorově efektivnější algoritmus pro konstrukci stromu [18]. Asi nejznámější práci spojenou s konstrukcí vydal roku 1995 doktor **Esko Juhani Ukkonen**, jehož algoritmus si ponechává výhody McCreightova algoritmu, ale je snazší na implementaci [21].

Pomocí sufixových stromů můžeme v lineárním čase řešit velké množství problému spojených s vyhledáváním v textu. Klasické použití pro tuto strukturu je řešení problémů s podřetězci, jako například nejdelší opakující se podřetězec nebo nejdelší společný podřetězec dvou a více textů. Právě proto je tato struktura, kromě vyhledávání v textu, využívána také při hledání podobností vzorků DNA v bioinformatice nebo při hledání opakujících se vzorků při kompresi dat.

Algoritmy používající sufixové stromy nejdříve musí provést předzpracování (tzv. *preprocessing*), jehož časová náročnost se odvíjí od délky textu, ve kterém bude probíhat vyhledávání, což může být časově náročné, pokud jsou data objemná. Nejvíce užitečné jsou proto tyto algoritmy v případě, kdy vyhledáváme velké množství různých podřetězců ve stejném textu, jelikož není potřeba toto předzpracování opakovat do té doby, dokud se nezmění text. Další výhodou, kterou nám tato operace přináší je časová náročnost samotných vyhledávacích operací, která se již neodvíjí od velikosti textu, ale od délky vyhledávaného podřetězce. Právě tím se liší od jiných algoritmů určených pro vyhledávání, jako jsou například Rabin–Karp algoritmus nebo Knuth–Morris–Pratt algoritmus, jejichž časová náročnost je při každém vyhledávání závislá na délce textu, ve kterém se vyhledává.

2.1 Struktura sufixového stromu

Sufixový strom vytváříme pro libovolný řetězec, ve kterém probíhá vyhledávání. Tento řetězec označujeme jako S a jeho délku jako m. Předpokládáme, že abeceda Σ, ze které se skládá text S, je známá a konečná. Definice je pro takovýto strom následující.

Definice 2.1.1. Jako T označme sufixový strom pro m-znakový řetězec S, který je kořenový jednosměrně orientovaný strom, mající přesně m listů, očíslovaných 0 až $m-1$. Každý interní uzel, jiný než kořenový, má alespoň dva potomky a každá hrana je označena neprázdným podřetězcem z S. Z jednoho uzlu nemohou vycházet dvě hrany s označením začínajícím stejným znakem. Klíčovou vlastností sufixového stromu je, že po zřetězení všech popisů hran po cestě z kořene k listu i, vznikne sufix řetězce S, který leží právě na pozici i, taktéž označený jako $S[i..m-1]$ [12].

Každý listový uzel nacházející se ve stromu nese hodnotu v podobě celého čísla, která je v rozsahu 0 až $m-1$. Tato hodnota označuje pozici unikátního sufixu, který tento listový uzel reprezentuje, jak to ukazuje obrázek 2.2. Jelikož kořenový a interní uzly žádný sufix nereprezentují, jejich hodnota je nastavována na záporné hodnoty, což pomáhá s rozlišováním od uzlů listových. Hrany spojující jednotlivé uzly jsou označeny podřetězcem z S, takže každá hrana reprezentuje část původního textu [19].
Pokud ve stromě vyhledáváme, cestujeme po těchto hranách a porovnáváme jejich označení s vyhledávaným vzorem. Pokud úspěšně porovnáme všechny znaky, vyhledávaný vzor se nachází na všech listech (indexech v textu), které se nachází pod místem, kde jsme ukončili porovnávání. Pokud jsme porovnávání skončili neúspěchem, požadovaný vzor se v textu nenachází.
V mnoha problémech řešených pomocí sufixového stromu počítáme hloubku, ve které se nacházíme. Tato hloubka se nepočítá podle uzlů, které jsme po cestě prošli, jako je tomu například u binárních stromů, ale počítá se podle délky hran, po kterých jsme šli. Hloubka je tedy délka textu, kterým jsou označeny hrany, které jsme doposud prošli.

2.2 Konstrukce stromu

Jak již bylo řečeno výše, nedílnou součástí práce se sufixovými stromy je předzpracování, což v případě této struktury je samotná konstrukce stromu. Strom skládáme z textu, ve kterém budeme vyhledávat, a tato operace je časově i prostorově závislá na délce tohoto textu. Ačkoliv se jedná o náročné předzpracování, není nutné jej opakovat, dokud nenastanou změny v textu. V této kapitole si řekneme, na jakém principu funguje rozšiřování stromu a ukážeme si některé algoritmy, které k tomu slouží. Hlavními zdroji pro teorií, ale i pro následné programování byly materiály [18, 21, 8, 6] a především pak kniha [12].

Pro konstrukci stromu existuje několik algoritmů, jako je například už zmíněný Ukkonenův nebo McCreightův algoritmus. V této sekci popíši princip naivního algoritmu, který je jednoduchý na implementaci, ale je časově neefektivní a také Ukkonenův algoritmus, který je složitější, zato je časově efektivnější.

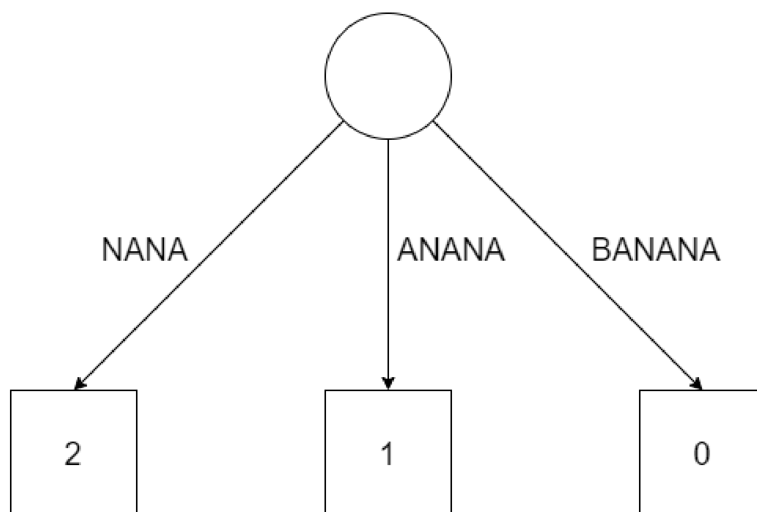
Prvním krokem konstrukce je vždy vytvoření kořenového uzlu, který žádný sufix nereprezentuje, s prázdným seznamem potomků. Poté v jednotlivých iteracích provádíme rozšiřování stromu o jednotlivé sufixy. Iterace probíhají od $i = 0$, až po $i = m-1$, kdy vkládáme do stromu sufix $i..m-1$. Začínáme od nejdelšího sufixu $0..m-1$, což je vlastně celý vstupní text, a postupně přidáváme kratší sufixy, až skončíme na posledním sufixu $m-1..m-1$ [12].

Při přidávání do stromu porovnáváme znak po znaku přidávaného sufixu s hranami stromu. Jakmile se při porovnávání stane, že se znak popisu hrany liší od znaku vkládaného řetězce, vytvoříme nový interní uzel a vložíme jej do hrany před místo, kde se tyto dva znaky lišily, a nově vytvoříme také list a k němu hranu, kterou označíme zbylými znaky přidávaného sufixu. Takto postupujeme, dokud ve stromu nejsou obsaženy všechny sufixy.

2.2.1 Implicitní strom a ukončovací znak

Při konstrukci stromu narážíme na problém s tzv. implicitním výskytem sufixu uvnitř stromu. Je to situace, kdy cesta reprezentující některý sufix nekončí na listovém uzlu, ale někde uvnitř hrany. Když k tomu dojde, jeden nebo více sufixů nejsou reprezentovány listovým uzlem a nemáme tedy ani index jejich výskytu v textu, čímž tento strom nerespektuje předchozí definici 2.1.1. Strom ve kterém k této situace dojde je tzv. **implicitní sufixový strom**.

K této situaci dojde v případě, kdy sufix textu S je identický s prefixem jiného sufixu z S. Obrázek 2.1 nám ukáže, jak vypadá implicitní sufixový strom, který byl sestaven pro text S = "BANANA".



Obrázek 2.1: Implicitní sufixový strom pro S = "BANANA"

V tomto stromu jsou sice vyjádřeny všechny sufixy, ale některé z nich nejsou reprezentovány explicitně. Jak už bylo řečeno, k této situaci došlo, protože některý sufix byl identický s prefixem jiného sufixu. V tomto slově k tomu došlo i vícekrát, například sufix "NA" je shodný s prefixem sufixu "NANA", což způsobilo, že první ze zmíněných není explicitně obsažen ve stromu. Stejné platí i pro sufixy "A" a "ANA".

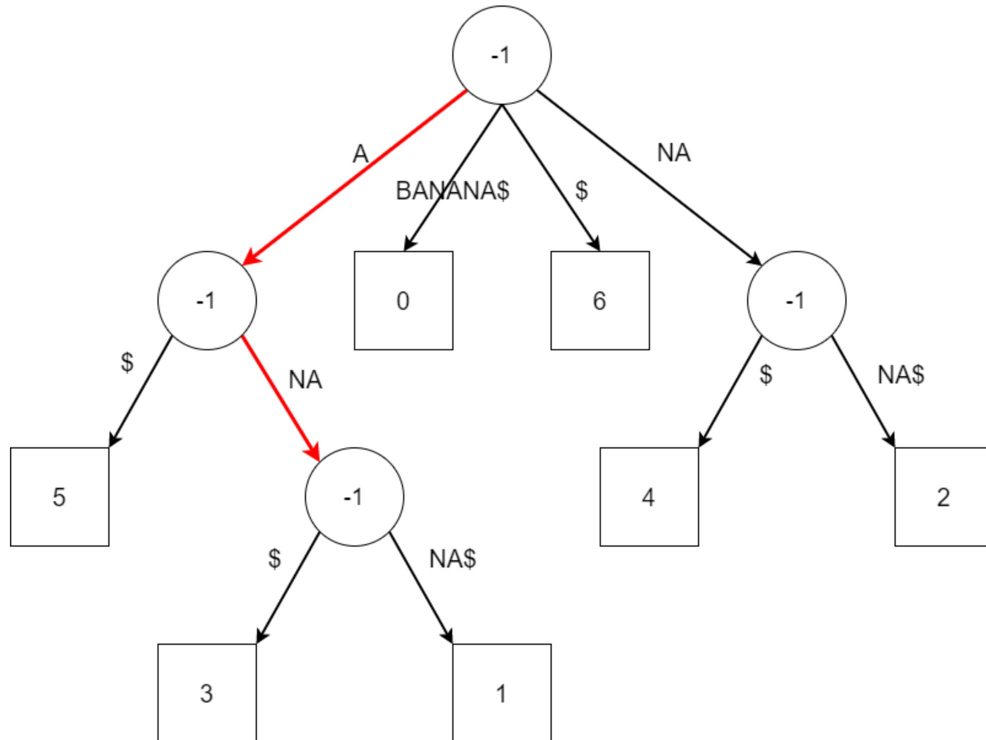
Takovýto strom nám ovšem neumožňuje správně vyhledávat v textu, protože jak bylo vysvětleno v předchozí podkapitole, indexy výskytů vyhledávaného podřetězce se nachází na listech, které jsou na nižší úrovní pod místem, kde jsme úspěšně dokončili porovnávání znaků požadovaného podřetězce se znaky, kterými je označena hrana. Pokud bychom tedy například ve slově "BANANA" vyhledávali podřetězec "ANA" pomocí takového stromu, jediný index, ke kterému bychom se dostali, je 1, ačkoliv se v tomto slově vyskytuje dvakrát. Je to způsobeno právě tím, že sufix "ANA" je zahrnut implicitně v sufixu "ANANA", ale nemáme list s indexem 3, který by jej explicitně reprezentoval.

K této situaci nedojde, pokud je na konci textu S znak, který se ve zbytku textu nevyskytuje, protože poté žádný sufix nemůže být prefixem jiného sufixu. Ovšem předpokládat, že zadaný text bude tuto podmínku vždy splňovat, není příliš reálné, a proto vždy do textu vkládáme speciální znak, který značí konec vstupního textu S.

Tento ukončovací znak (termination character) se nesmí vyskytovat jinde v textu, jak už bylo řečeno. Ovšem abychom testovali, který znak se v tomto textu nevyskytuje a můžeme jej vložit na konec, je poměrně neefektivní a časově zbytečně náročné. Kvůli tomu využíváme znaky, které

se nenachází v abecedě, ze které je tvořen vstupní text S. V literatuře se často využívá znaku '$', popřípadě '#' [8].

Pokud tento znak připojíme na konec dříve použitého slova "BANANA", můžeme vytvořit strom, ve kterém jsou explicitně vyjádřeny všechny sufixy, jak ukazuje obrázek 2.2.



Obrázek 2.2: Strom pro S = "BANANA$"[1].

Chceme-li nyní znovu vyhledat podřetězec "ANA", cesta povede přes hrany, které jsou v obrázku vyznačeny červeně. Porovnávání úspěšně skončí na interním uzlu a indexy s výskytem se nachází na listových synovských uzlech čili na indexech 1 a 3. Už se nesetkáme s problémem implicitního stromu, jelikož sufix "ANA$" již nadále není prefixem sufixu "ANANA$".

Oproti stromu, který byl vytvořen z textu bez ukončovacího znaku, tento strom dodržuje počet listů m, který má být roven počtu znaků v textu S. Navíc nám přibyl jeden list, který reprezentuje sufix *"$"* čili prázdný podřetězec. Definice popisuje implicitní strom takto:

Definice 2.2.1. Implicitní sufixový strom pro řetězec S je strom, který můžeme získat ze sufixového stromu pro řetězec $S\$$ tak, že odstraníme veškeré zakončovací znaky *$* ze všech označení hran stromu a následně odstraníme všechny hrany, jejichž popis je prázdný řetězec a nakonec odstraníme každý uzel, který nemá alespoň dva potomky [12].

2.3 Naivní algoritmus

Jedná se o nejjednodušší algoritmus pro konstrukci sufixového stromu. V praxi se nepoužívá kvůli své časové náročnosti, je ovšem dobré jej popsat a uvést zde základní principy stavby stromu a způ-

[1]Zdroj obrázku: https://www.wisdomjobs.com/e-university/data-structures-tutorial-290/suffix-tree-7249.html

soby rozdělování hran a přidávání uzlů. Tento algoritmus nám také poskytne základy pro správné pochopení složitějších algoritmů používaných pro konstrukci stromu, jako je například Ukkonenův algoritmus.

Tato jednoduchá metoda, při stavbě stromu pro řetězec S, nejdříve do prázdného stromu (pouze kořenový uzel) vloží hranu označenou sufixem $S[0..m-1]\$$, což je vlastně celý řetězec S, a poté postupně přidává další sufixy $S[i..m-1]\$$ do stromu, kde i je celé číslo v rozmezí 1 až $m-1$. Po tomto prvním kroku se ve stromě nachází kořen a list s indexem 0, k němuž vede hrana označená řetězcem $S\$$. Takovýto strom označujeme jako N_0. Stromem N_i označujeme pokročilý strom, do nějž jsou již vloženy všechny sufixy z indexů 0 až i. Strom N_{i+1} tedy vždy konstruujeme z N_i [12].

Tato $i+1$ konstrukce probíhá tak, že v kořenovém uzlu N_i začneme s vyhledáváním nejdelší cesty, jejíž označení je identické s prefixem právě vkládaného sufixu $S[i+1..m-1]$\$. Tato cesta je dle definice 2.2.1 unikátní, jelikož víme, že žádné dvě hrany z jednoho uzlu nemají stejný počáteční znak popisu. Porovnáváme tedy znaky popisu této unikátní cesty se znaky vkládaného sufixu, dokud není žádná další shoda možná. K neshodě znaků může dojít ve dvou případech.

První ze situací je, že porovnávání skončí v některém z interních uzlů, označeného například jako w. Žádná hrana vedoucí z interního uzlu w tedy nezačíná požadovaným znakem ze vkládaného sufixu. V takovém případě stačí pouze vytvořit listový uzel označený $i+1$ a k němu hranu $(w, i+1)$, vedoucí z uzlu w a označenou zbylými znaky vkládaného sufixu, které již nebyly porovnávány.

Druhý případ nastane, pokud se nacházíme uprostřed některé hrany, označované například jako (u, v). Algoritmus následně postupuje tak, že hranu (u, v) rozdělí na dvě tím, že dovnitř vloží interní uzel w za poslední znak hrany, který byl identický se znakem z $S[i+1..m-1]$, a před první odlišný znak. Hrana (u, w) je poté označena znaky (u, v), které byly identické s těmi z $S[i+1..m-1]$, a hrana (w, v) je označena zbylými znaky z (u, v). Také je vytvořen nový listový uzel $i+1$ připojený hranou $(w, i+1)$ k uzlu w, který se vkládá stejně, jako tomu bylo v první situaci.

Nyní je ve stromu obsažena unikátní cesta z kořene po list $i+1$. Cesta je označena sufixem $S[i+1..m-1]$\$. Zároveň jsme byli schopni zajistit pravidlo, které říká, že žádné dvě hrany vycházející ze stejného uzlu nejsou označeny stejným počátečním znakem.

Pokud budeme předpokládat, že abeceda, ze které je tvořen řetězec S, je konečná, tato metoda má časovou náročnost $O(m^2)$, pro řetězec délky m [12].

2.4 Ukkonenův algoritmus

Esko Juhani Ukkonen navrhl svůj lineární algoritmus pro stavbu sufixových stromů v roce 1995, kdy jej prezentoval v časopise Algorithmica [21]. Důležitá vlastnost algoritmu je, že je on-line. Zpracovává znaky vstupního řetězce jeden po jednom a to zleva doprava, aniž bychom museli znát celý řetězec. Celý algoritmus byl založen na vypozorovaném jevu, který nám říká, že sufixy řetězce S^i=$s_0...s_i$ mohou být získány ze sufixů S^{i-1}=$s_0...s_{i-1}$ připojením znaku S_i na konec každého sufixu S^{i-1}. Sufixy celého řetězce $S = S^{m-1} = s_0...s_{m-1}$ tak můžeme získat rozšiřováním nejdříve sufixu S^0 na sufix S^1 a takto stálým opakováním rozšiřování získat konečný sufix S^{m-1}, který získáme rozšířením S^{m-2}.

Tento způsob konstrukce je značně odlišný od Weinerovy a McCreightovy metody, které jsou navržené na odlišném principu zpracování vstupního textu [8]. Weinerova metoda řetězec zpracovává zprava doleva a sufixy jsou do stromu přidávány postupně od nejkratšího sufixu, až je naposledy do stromu přidán nejdelší sufix čili celý vstupní řetězec. McCreightova metoda naopak vkládá sufixy do stromu od toho nejdelšího a postupně jde až k tomu nejkratšímu. Ačkoliv je McGreightův

algoritmus odlišný v rozšiřování stromu a postupuje v problematice jinak, je ve své výsledné formě do značné míry podobný s tím Ukkonenovým. Ukkonenův algoritmus má mimo jiné také vylepšení v paměťové náročnosti.

Ukkonenův algoritmus ovšem nepřidává do stromu jednotlivé sufixy jako takové, ale konstruuje posloupnost implicitních sufixových stromů, o kterých jsme si řekli už v kapitole 2.2.1. Poslední přidávaný implicitní strom jej poté promění na plný sufixový strom pro řetězec S, který vyjadřuje každý sufix explicitně. Abychom mohli blíže popsat princip tohoto algoritmu, definujeme stromy pro jednotlivé podřetězce takto:

Definice 2.4.1. Implicitní sufixový strom pro řetězec $S[0..i]$, pro i od 0 po $m-1$, označujeme jako $\mathcal{I}_i$ [12].

Algoritmus vytvoří implicitní strom $\mathcal{I}_i$ pro každý prefix $S[0..i]$ řetězce S, počínaje od $\mathcal{I}_0$, s narůstajícím i, až po $\mathcal{I}_{m-1}$. Tento poslední strom $\mathcal{I}_{m-1}$ poté vytvoří úplný sufixový strom, který už vyjadřuje explicitně všechny sufixy. Časová náročnost potřebná k tomuto sestavení je lineární, čili $O(m)$ [21].

2.4.1 Rozšiřování stromu

Stavba stromu probíhá v m etapách, kde m je klasicky délka vstupního řetězce S. V každé etapě je vytvořen nový strom, který je postaven na stromu z předchozí etapy. V etapě $i+1$ tedy stavíme implicitní strom $\mathcal{I}_{i+1}$, ze stromu $\mathcal{I}_i$. Tuto $i+1$ etapu dále dělíme na $i+1$ rozšíření, kde se v každém rozšíření věnujeme jednotlivým sufixům $S[0..i+1]$ [12].

V rozšíření j etapy $i+1$ nejdříve hledáme cestu, která začíná v kořeni stromu a je označena podřetězcem $S[j..i]$. Jakmile ji najdeme, přidáme na konec znak $S[i+1]$ (protože jsme v etapě $i+1$). Pokud se znak $S[i+1]$ na místě kam jej chceme vkládat už nachází, nepřidáváme v tomto rozšíření nic. V praxi tedy v etapě $i+1$ přidáváme do stromu nejprve řetězec $S[0..i+1]$ a následují $S[1..i+1]$, $S[2..i+1]$ a tak dále, dokud poslední rozšíření $i+1$ poslední etapy $i+1$ nerozšíří prázdný sufix řetězce $S[0..i]$, což znamená, že přidáme do stromu znak $S[i+1]$ z řetězce S.

Pravidla rozšiřování stromu

Při rozšiřování sufixového stromu, během jednotlivých etap a rozšíření, může dojít k různým situacím. Proto existují v rámci tohoto algoritmu tři pravidla, které nám říkají, jakým způsobem máme rozšiřovat strom o nové znaky. Těmito pravidly se řídíme i při programovém zrealizování tohoto algoritmu.

Pro znázornění pojmenujme $S[j..i]$ jako β. β je sufix z $S[0..i]$. Nachází-li se algoritmus v rozšíření j, etapy $i+1$, najde nejprve cestu ve stromu označenou řetězcem β a následně rozšíří tuto cestu o nový znak, aby bylo zajištěno, že se ve stromě nachází sufix $\beta S(i+1)$. Dále přejdeme do dalšího rozšíření $j+1$ a takoto pokračujeme až k rozšíření $j=i+1$. Každé toto rozšíření se provede podle jednoho ze tří následujících pravidel, podle toho, jaká je situace.

Pravidlo 1

Cesta označená β končí na listovém uzlu. To znamená, že tato cesta vede od kořene až po konec některé listové hrany aktuálního stromu. V tomto případě rozšíření stromu provedeme tak, že přidáme znak $S(i+1)$ na konec označení této listové hrany.

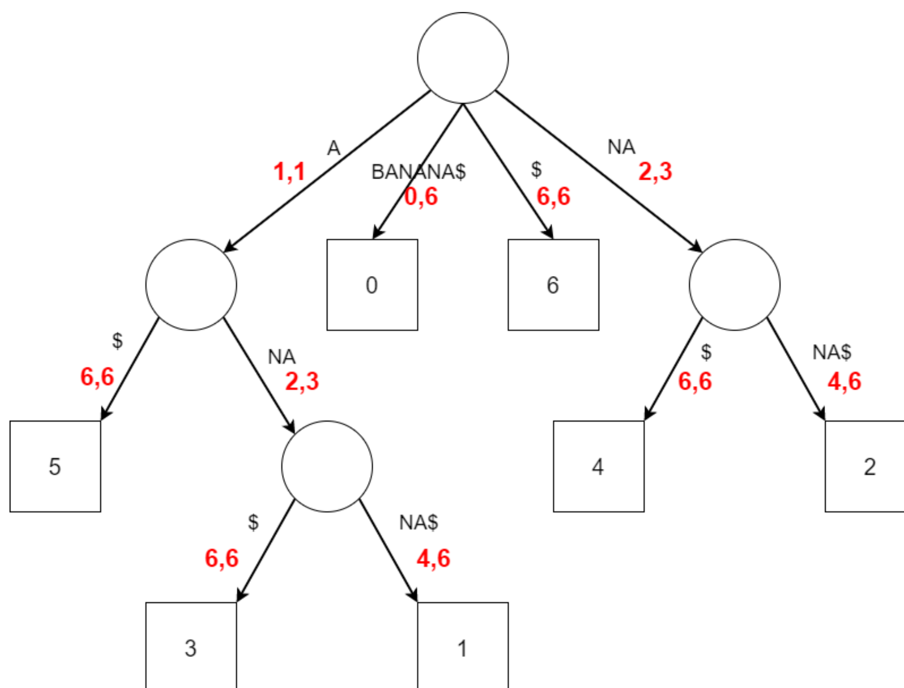
Pravidlo 2
Jakmile najdeme cestu označenou β a následně existuje alespoň jedna cesta následující za β, ale žádná z nich nepokračuje znakem $S(i+1)$, musíme vytvořit cestu novou. Do aktuálního stromu je vložena nová listová hrana označená znakem $S(i+1)$, a to za pozici cesty β. Nově vytvořenému listovému uzlu je přidělen index j. Navíc, pokud cesta β končí uprostřed některé hrany, je vytvořen také interní uzel, taktéž na místě za β.

Pravidlo 3
Některá z cest, které se nachází za cestou β, již pokračuje znakem $S(i+1)$. V tomto případě je řetězec $\beta S(i+1)$ už v aktuálním stromu obsažen, ačkoliv musíme mít na paměti, že je tento řetězec obsažen implicitně. V tomto případě tedy nemusíme dělat žádná rozšíření.

2.4.2 Snížení prostorové náročnosti

Kromě snížení časové náročnosti se Ukkonenův algoritmus zaměřuje také na prostorovou náročnost. Pro konstrukci stromu s dosud známými poznatky může strom požadovat až $\Theta(m^2)$ místa v paměti [11]. Dosud popsaný postup totiž může ve všech hranách stromu přidat celkově více znaků, než je $\Theta(m)$. Abychom byli schopni vytvořit algoritmus, jehož časová náročnost je lineární, tedy $O(m)$, je zapotřebí snížit také paměťovou náročnost, jelikož čas konstrukce je alespoň tak velký, jaká je velikost jeho výstupu, což znamená, že takovýto počet znaků dělá lineární časovou náročnost nemožnou.

Abychom v tomto směru dosáhli zlepšení, můžeme použít jiný způsob označení hran stromu. Místo toho, abychom explicitně psali podřetězec ke každé hraně, použijeme dvojici indikátorů, které budou sloužit jako počáteční a koncový index daného podřetězce z S. Jestliže má tento algoritmus uchovanou podobu původního řetězce S, může pomocí těchto indikátorů zjistit, jaký konkrétní podřetězec hrana reprezentuje a to v konstantním čase. Tímto způsobem můžeme vyjádřit jakýkoliv podřetězec ve stromě, jako by byl explicitně vložný, ovšem s pomocí konstantního počtu symbolů označujících hrany stromu. Tento způsob značení si můžeme ukázat na příkladu 2.3.



Obrázek 2.3: Komprese označení hran ve stromě pro S = "BANANA$"[2]

Pokud vezmeme strom pro S = "BANANA$", který jsme si ukázali v předchozí kapitole a uplatníme na něj kompresi označení hran, místo všech znaků, které byly použity pro označení hran, máme nyní pouze čísla, která značí rozsah daných podřetězců reprezentujících tuto hranu. Reálně se tedy ve stromě budou nacházet pouze tyto dvojice čísel, které jsou na obrázku červeně. Černý text, který je v obrázku u hran, je pouze znázornění podřetězce, který tato hrana reprezentuje.

V praxi poté Ukkonenův algoritmus při hledání cesty a porovnávání popisu hrany používá indexy, které jsou přiřazeny dané hraně, aby získal požadovaný podřetězec z S, který tato hrana reprezentuje, a až poté provádí porovnání s tímto podřetězcem. Stejným způsobem je implementováno také rozšíření, kdy je použit stejný způsob označování hran. Pokud se ve etapě $i + 1$ uplatní druhé rozšiřovací pravidlo, je nově vytvořená hrana vedoucí k listovému uzlu označena indexy $(i + 1, i + 1)$. V případě uplatnění prvního pravidla na listovou hranu, je její označení změněno z (p, q) na $(p, q + 1)$.

Tím, že popisujeme hranu pouze pomocí dvou indexů, jsou ke všem hranám stromu přiděleny pouze dvě čísla. Vzhledem k tomu, že ve stromě se nachází nanejvýš $2m - 1$ hran, sufixový strom používá pouze $O(m)$ symbolů a požaduje pouze $O(m)$ paměti [12]. Díky tomuto způsobu označování hran máme nyní možnost strom sestavit v lineárním čase $O(m)$.

2.4.3 Urychlení konstrukce

Pokud bychom nyní konstrukci stromu prováděli s danými pravidly, museli bychom vždy najít řetězec β v současném stromě a v konstantním čase přidat nový znak $S(i+1)$, abychom zajistili, že se bude řetězec $\beta S(i+1)$ nacházet ve stromě. Ovšem k přidání tohoto znaku musíme v každé etapě i-krát najít řetězec β, abychom mohli tento nový znak přidat. Pokud bychom tuto cestu β hledali,

[2]Zdroj obrázku: https://www.wisdomjobs.com/e-university/data-structures-tutorial-290/suffix-tree-7249.html

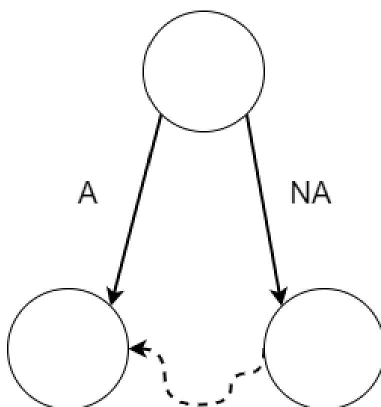
potřebný čas k projití cesty od kořene by byl $O(|\beta|)$. Tímto postupem by byla časová náročnost rozšíření j etapy $i+1$ $O(i+1-j)$. Konstrukce stromu $\mathcal{I}_{i+1}$, který je tvořen z $\mathcal{I}_i$, by zabrala $O(i^2)$ času a vytvoření kompletního stromu $\mathcal{I}_m$ by trvalo $O(m^3)$ [12].

Tento postup by však byl nevýhodný i proti naivnímu algoritmu popisovanému výše. Ovšem existuje několik metod a triků, jak tuto konstrukci urychlit na $O(m)$ z původního $O(m^3)$. Jednotlivé triky samostatně tento proces neurychlí a vypadají spíše jako heuristiky, ale pokud spojíme dohromady několik těchto metod, jsme schopni zajistit lineární časovou náročnost pro konstrukci.

Sufixový odkaz

Definice 2.4.2. Jako $x\alpha$ označme náhodný řetězec, kde x označuje samostatný znak a α označuje podřetězec. Mějme interní uzel v s hranou označenou řetězcem $x\alpha$, pokud je ve stromě jiný uzel $s(v)$ s hranou označenou α, pak ukazatel vedoucí z v do $s(v)$ je nazýván sufixovým odkazem [8].

Na obrázku 2.4 je znázorněn příklad sufixového odkazu.



Obrázek 2.4: Sufixový odkaz, znázorněný přerušovanou šipkou

Dle definice můžeme označit jako v interní uzel, který má hranu označenou podřetězcem "NA" a jako $s(v)$ interní uzel s označenou hranou "A". Označení hrany $x\alpha$ je tedy podřetězec "NA", kde znak x je v tomto případě 'N' a řetězec α je jednoznakový tedy "A".

Definice 2.4.3. V každém implicitním stromě $\mathcal{I}_i$, pokud interní uzel v má hranu k němu vedoucí označenou $x\alpha$, pak v $\mathcal{I}_i$ existuje uzel $s(v)$ s hranou označenou α [12].

Ve speciálním případě, kdy je α prázdným řetězcem, sufixový odkaz interního uzlu s hranou označenou řetězcem $x\alpha$ ukazuje na uzel kořenový. Ze samotného kořenového uzlu, který není považován za interní, pak nevede žádný sufixový odkaz.

Jako dříve počítejme s tím, že v etapě $i+1$ algoritmus hledá sufix $S[j..i]$ řetězce $S[0..i]$ v rozšíření j, pro toto j zvyšující se od 0 po $i+1$. Normálně bychom $S[j..i]$ hledali porovnáváním s označeními jednotlivých cest vedoucích z kořene stromu. Díky sufixovým odkazům získáváme možnost toto zdlouhavé vyhledávání přeskočit.

Cesta označená řetězcem $S[0..i]$ musí vždy končit na listovém uzlu stromu $\mathcal{I}_i$, jelikož $S[0..i]$ je nejdelší řetězec, který se v aktuálním stromě $\mathcal{I}_i$ nachází. Pokud si budeme uchovávat ukazatele na listové uzly, reprezentující tyto dosavadní nejdelší řetězce $S[0..i]$, je lehké je najít a jejich rozšíření je provedeno pomocí prvního rozšiřovacího pravidla. Takže první rozšíření libovolné etapy

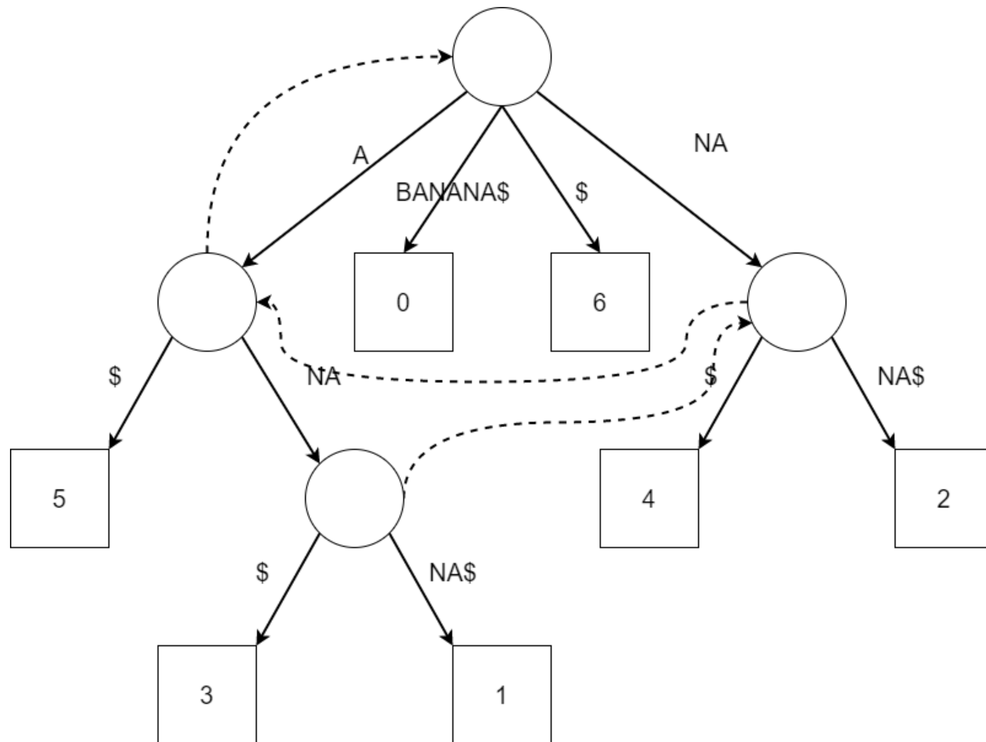
je zvláštní v tom, že nám stačí pouze konstantní čas na uplatnění prvního pravidla pro rozšíření, jelikož víme, kde je listový uzel pro tento nejdelší řetězec, pokud si na něj uchováme ukazatel [12].

Abychom nyní upřesnili pravý význam sufixového odkazu v uzlech, uvedeme příklad. Řetězec $S[0..i]$ označme jako $x\alpha$, kde x je jediný znak a α je podřetězec. Dále označme $(v, 0)$ hranu, která vede z interního uzlu v do listu, který reprezentuje tento řetězec $x\alpha$. Algoritmus musí nyní v druhém rozšíření najít konec cesty $S[1..i]$,což je v našem případě α, ve stromě $\mathcal{I}_i$. Důležité je, že uzel v je buď kořen nebo je to interní uzel stromu $\mathcal{I}_i$. Pokud se jedná o kořen, algoritmus najde cestu tak, že ji postupně hledá počínaje v kořeni, stejně jako naivní algoritmus. Ale pokud se jedná o interní uzel, pak dle definice uzel v má sufixový odkaz vedoucí k uzlu $s(v)$. Jelikož cesta vedoucí k tomuto uzlu $s(v)$ má prefix řetězce α, zbytek tohoto řetězce α musí končit v podstromu uzlu $s(v)$. Díky tomu nemusíme procházet celou cestu od kořene, ale můžeme část této cesty přeskočit a začít až na uzlu $s(v)$.

Pro detailnější popsání kroků v druhém rozšíření etapy, pojmenujme označení hrany $(v, 0)$ jako γ. Abychom nalezli α, musíme z listového uzlu 0 přejít na jeho otcovský uzel v. Dále musíme z tohoto uzlu v přejít na uzel $s(v)$, ke kterému směřuje sufixový odkaz a z $s(v)$ dále pokračovat po cestě označené γ. Na konci této cesty (tedy konec řetězce α) rozšíříme strom o nový znak dle pravidel pro rozšiřování.

Tímto způsobem tedy probíhá první a druhé rozšíření. Pro rozšíření libovolného řetězce $S[j..i]$ na $S[j..i + 1]$ pro j větší než 2, musíme opakovat tento obecný postup, při kterém v aktuálním stromu začínáme na konci cesty, označené řetězcem $S[j - 1..i]$. Přejdeme na otcovský uzel, což je buď kořenový uzel, nebo interní uzel v mající sufixový odkaz vedoucí do $s(v)$. Jako γ pojmenujme označení hrany, kterou jsme prošli při cestě k otcovskému uzlu v, za předpokladu, že v není kořenový uzel. Následně přejdeme k uzlu $s(v)$, na který ukazuje sufixový odkaz a projdeme cestu označenou řetězcem γ až na konec $S[j..i]$, kde nakonec rozšíříme strom na $S[j..i + 1]$, podle rozšiřovacích pravidel.

Na obrázku 2.5 jde vidět, jak vypadá hotový strom pro S = "BANANA$", se všemi sufixovými odkazy.



Obrázek 2.5: Sufixové odkazy ve stromě pro S = "BANANA$"[3].

Sufixové odkazy nám tak ušetří zbytečné procházení cest, kdy nemusíme v každém rozšíření hledat cestu od kořene, ale můžeme tuto cestu přeskočit. Při řešení problematiky s časovou náročností jsou sufixové odkazy velikou výhodou. Jelikož nám sufixové odkazy ještě neposkytnou lineární časovou náročnost, algoritmus implementuje ještě několik dalších triků pro zlepšení výkonu [12].

Trik 1: Přeskočení han

V rozšíření $j+1$ hledáme cestu z uzlu $s(v)$, který je označena řetězcem γ. Mysleme na to, že existence této cesty γ je zaručena díky rozšíření $j+1$, z předchozí etapy. Tato cesta γ by při normální implementaci zabrala $|\gamma|$ času, což je počet znaků na této cestě. Pokud ovšem použijeme trik, který hrany přeskakuje, čas potřebný k najití cesty se stane úměrný k počtu uzlů na cestě označené γ.

Jako g označme délku γ a jako g' délku hrany, jejíž označení má stejný první znak jako γ (mějme na paměti, že z jednoho uzlu $s(v)$ nevychází dvě hrany se stejným počátečním znakem jejich označení). Pokud délka g' je kratší než délka g, nemusíme tuto hranu porovnávat s γ a tuto hranu jednoduše přeskočíme na uzel, který se nachází na konci této hrany. Abychom věděli, které znaky jsou už porovnány (přeskočeny), nastavíme g na $g - g'$ a proměnnou h na g' + 1, následně opět najdeme hranu, jejíž první znak označení odpovídá požadovanému znaku řetězce γ, na pozici h. Obecně vždy porovnáváme délku hrany g' se zbylými znaky γ (g) a pokud je toto g větší nebo rovno g', je tato hrana přeskočena na uzel na konci hrany a nastavíme g na $g - g'$, h nastavíme na $h + g'$ a opět najdeme správnou hranu, která odpovídá našim požadavkům. Jakmile dojdeme ke hraně, kde g' je větší než g, pak na této hraně přeskočíme ke znaku na pozici g a můžeme provádět

[3]Zdroj obrázku: https://www.wisdomjobs.com/e-university/data-structures-tutorial-290/suffix-tree-7249.html

požadované vkládaní nového znaku, dle rozšiřovacích pravidel.

Trik 2: Pravidlo 3 ukončuje etapu

V libovolné etapě $i + 1$, ve které se uplatní pravidlo 3 v rozšíření j, bude toto pravidlo uplatněno ve všech zbylých rozšířeních $j + 1$ až $i + 1$ této etapy. Jelikož při uplatnění třetího rozšiřovacího pravidla platí, že cesta označená řetězcem $S[j..i]$ v současném stromu již pokračuje znakem $S(i+1)$ a taktéž tímto znakem pokračuje cesta $S[j + 1..i]$ v rozšíření $j + 1$ a stejně tomu tak je i ve všech následujících rozšířeních, až po poslední rozšíření $i + 1$ dané etapy.

Všechna rozšíření etapy $i + 1$, která jsou tímto způsobem dokončena (přeskočena) po prvním užití třetího pravidla, jsou brána jako dokončena implicitně. Oproti tomu libovolné rozšíření j, ve kterém je konec řetězce $S[j..i]$ nalezen explicitně, nazýváme jako rozšíření explicitní.

Můžeme tedy počítat s tím, že pokud se uplatní pravidlo 3 pro rozšiřovaní, není třeba dělat žádnou práci, jelikož sufix, který chceme do stromu vkládat je již v tomto stromě obsažen. Navíc se nemusíme starat o přidávání sufixových odkazů, jelikož se do stromů vkládají pouze v případě uplatnění druhého rozšiřovacího pravidla.

Trik 3: Jednou list, navždy list

Pokud je v některé etapě Ukkonenova algoritmu vytvořen nový listový uzel označený jako j, což značí počáteční index sufixu v S, potom zůstane tento uzel listem i v dalších etapách až do konce tohoto algoritmu. Pokud se vrátíme k pravidlům pro rozšiřování stromu, zjistíme, že ani jedno pravidlo nedisponuje technikou, která by prováděla rozšiřování listové hrany za hranici listového uzlu, na který ukazuje. Navíc, jakmile je vytvořen list s označením j, první pravidlo pro rozšiřování bude vždy použito při následujících etapách na rozšíření j. Pro popsání tohoto triku si připomeňme, že každý popis hrany, která se nachází v sufixovém stromě, reprezentuje část řetězce S. Tento podřetězec tedy můžeme popsat pomocí dvou proměnných p a g, které nám říkají na kterém indexu podřetězec začíná a na kterém končí. Podřetězec $S[p..q]$ tedy začíná na indexu p a končí na q. Mějme také na paměti, že kterýkoliv popis hrany vedoucí k listovému uzlu a nacházející se v $\mathcal{I}_i$ má tento index q roven číslu etapy i a v etapě $i + 1$ bude toto q inkrementováno na $i + 1$, jelikož tato hrana bude rozšířena o znak $S(i + 1)$ tak, aby byl obsažen tento nový sufix.

V etapě $i + 1$, když je vytvořen nový listový uzel a jeho hrana by byla normálně označena podřetězcem $S[p..i + 1]$, namísto klasických indexů hrany $(p,\ i + 1)$ použijeme označení (p, e), kde p je rovno číslu rozšíření j, ve kterém byl list vytvořen, a pomocí e označujeme "aktuální koncový index". Tento symbol e je globální index, jehož hodnota je nastavena na $i + 1$ na konci každé etapy. V etapě $i + 1$ víme, že první rozšiřovací pravidlo bude aplikováno v rozšířeních 0 až j_i a nepotřebujeme dělat žádnou práci na explicitních rozšířeních v těchto j_i rozšířeních. Místo toho pouze inkrementujeme konstantu e v konstantním čase a poté aplikujeme explicitní práci ve zbývajících rozšířeních, počínaje $j_i + 1$.

Pokud se ovšem v některém rozšíření j_{i+1} etapy $i + 1$ stane, že se uplatní třetí rozšiřovací pravidlo a dojde tak k ukončení etapy kvůli druhému triku, je nutné počítat s tím, že toto rozšíření j_{i+1} není bráno jako explicitně dokončené. V tom případě se v následující etapě $i + 2$ musí počítat s tím, že na rozšíření j_{i+1} se nevztahuje uplatnění třetího triku a není tedy provedeno v konstantním čase, dokud se na v některé následující etapě na j_{i+1} neuplatní druhé rozšiřovací pravidlo.

Při použití sufixových odkazů a všech urychlovacích triků můžeme pozorovat snížení časové náročnosti na $O(m)$, jelikož každé implicitní rozšíření stromu má konstantní časovou náročnost [12].

Počet explicitních rozšíření, které za dobu běhu algoritmu proběhnou jsou přímo úměrné velikosti textu m. V popisu třetího triku jsme poznamenali, že v každé etapě $i + 1$ je pouze jedno rozšíření j_{i+1}, ve kterém počítáme s explicitním rozšířením stromu. Pokud tedy počet explicitních rozšíření je rovno m, máme zajištěnou lineární časovou náročnost $O(m)$.

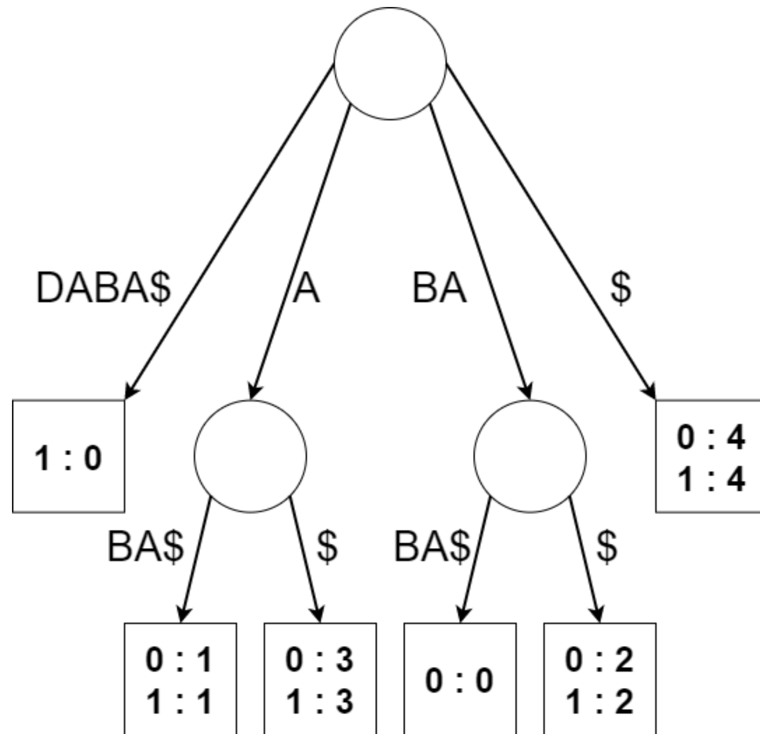
2.5 Generalizovaný sufixový strom

Dosud jsme se bavili o konstrukci stromu pouze pro jeden řetězec, ze kterého jsme sufixový strom vytvářeli. Při řešení určitých problematik spojených s vyhledáváním v řetězcích však potřebujeme, aby sufixový strom obsahoval sufixy z množiny řetězců $\{S_0, S_1, ..., S_z\}$. K sestavení takového stromu nám poslouží metody Ukkonenova algoritmu, které jsme si popsali dříve. Algoritmus pro stavbu stromu tedy není třeba celý předělávat, ale stačí jej rozšířit o několik různých vlastností. Strom však musí nést určité dodatečné informace, jako je například pořadové číslo řetězce v listovém uzlu, abychom věděli, z jakého řetězce je sufix, který nám tento listový uzel reprezentuje. Máme dva způsoby, jakými můžeme generalizovaný sufixový strom postavit.

První ze způsobů je lehký na pochopení i implementaci a požaduje pouze minoritní úpravy algoritmu pro konstrukci stromu. Nevýhodou ovšem je, že není příliš univerzální a jeho využití se hodí pouze u menšího počtu řetězců, ze kterých chceme strom sestavovat. Myšlenka je taková, že strom budeme sestavovat pouze z jednoho velkého řetězce S, který ovšem sestává ze zřetězených vstupních řetězců $S_0...S_z$. Za každý z těchto řetězců však musíme vložit ukončovací znak takový, aby nám nevznikl implicitní sufixový strom (popsáno v kapitole 2.2.1). Abychom toho docílili, musí být ukončovací znak za každým řetězcem unikátní, což znamená, že musíme mít jakousi množinu zakončovacích znaků, které budeme používat. Pokud bychom tedy chtěli sufixový strom postavit pro tisíce různých řetězců, museli bychom mít ohromné množství zvláštních zakončovacích znaků. Pokud ovšem nemáme nekonečnou řadu takovýchto znaků, nemůžeme sestavit sufixový strom pro neomezenou množinu řetězců.

Pokud použijeme druhý postup, jsme schopni tento problém vyřešit a stačí nám k tomu jediný zakončovací znak '\$'. Namísto zřetězení všech vstupních řetězců sestavíme strom pro první řetězec S_0 pomocí Ukkonenova algoritmu a jakmile máme strom hotový, spustíme tento algoritmus znovu, počínaje v kořeni stromu, pro řetězec S_1. Tímto způsobem přidáváme všechny řetězce, ze kterých chceme, aby se strom skládal. Opět si však musíme dávat pozor na problém s implicitně obsaženými sufixy, a proto musíme provést určitou modifikaci v sufixovém stromě. Pokud například dva různé vkládané řetězce mají stejný poslední znak, označme ho jako x, budeme do stromu vkládat stejný řetězec x'\$', což povede k implicitnímu vyjádření u podřetězce, který byl vkládán později. Jelikož by toto vedlo k nekorektnímu vyjádření sufixů ve stromu, musíme modifikovat listové uzly tak, aby mohly reprezentovat stejný sufix, který je ovšem obsažen ve více řetězcích. V praxi tedy takovýto list nese více než jedno číslo, které reprezentuje index, na kterém se nachází daný sufix a také index určující pořadové číslo vkládaného řetězce, ve kterém se sufix nachází. Na obrázku 2.6 je zobrazen generalizovaný sufixový strom, který je sestaven z řetězců S_0 = "BABA\$" a S_1 = "DABA\$".

Pokud chceme sestavit generalizovaný sufixový strom, který reprezentuje všechny sufixy množiny řetězců $\{S_0, S_1, ..., S_z\}$, musíme nejprve sestavit strom pro S_0, který následně budeme rozšiřovat o řetězec S_i, kde $i = 1..z$. Algoritmus provede takovéto rozšíření v čase $O(|S_i|)$ [21]. Celý generalizovaný strom tedy sestavíme v čase $O(m)$, kde m v tomto případě označuje součet délek $m_0..m_z$, což jsou délky řetězců $S_0...S_z$.



Obrázek 2.6: Generalizovaný sufixový strom pro řetězce S_0 = "BABA$" a S_1 = "DABA$". Číslo před dvojtečkou v listu označuje, ve kterém vstupním řetězci se daný sufix nachází a číslo za dvojtečkou označuje na jakém indexu se vyskytuje. Listy, které mají dva řádky reprezentují sufix, který se vyskytuje v obou řetězcích S_0 i S_1.

Nyní už víme, jak funguje konstrukce stromu pomocí Ukkonenova algoritmu a jaké všechny postupy jsou použity pro zrychlení konstrukce. V další kapitole se už zaměříme na samotnou funkčnost stromu čili na vyhledávání.

Kapitola 3

Využití sufixového stromu

V této kapitole se zaměříme především na využití sufixových stromů a na to, jaké všechny operace nám tato struktura umožňuje. Ukážeme si, jakým způsobem funguje práce nad tímto stromem a jak v něm správně hledat. Kromě použití sufixového stromu si také přiblížíme jiné postupy k vyřešení těchto problémů a řekneme si, jaké výhody a nevýhody tyto postupy přináší. Informace o tom, jakým způsobem fungují různé algoritmy a jak je implementovat, jsem čerpal z těchto zdrojů: [17, 12, 4, 3, 23].

3.1 Vyhledávání podřetězců

Vyhledávání vzoru (*Pattern*, taktéž označován jako *Needle*) v textu (*Haystack*) je klasický a asi nejčastěji řešený problém, který je spojený s problematikou vyhledávání v řetězcích. Algoritmy, které tento problém řeší, se snaží najít buď první, nebo všechny výskyty vzoru v textu a vrátit jeho počáteční index (indexy). Z možností, které nám sufixový strom nabízí, je právě tato problematika nejtriviálnější.

Existuje mnoho postupů, jak můžeme v textu hledat, ale některé jsou méně efektivní než jiné. Popíšeme si, jakým způsobem se pomocí sufixového stromu vyhledávají tyto vzory a jaké jiné postupy a algoritmy máme k řešení tohoto problému. Konkrétně si popíšeme ty nejznámější algoritmy pro porovnávání řetězců (string–matching algorithms), jako je například **Rabin–Karp** algoritmus nebo
Knuth–Morris–Pratt algoritmus. Řekneme si také něco o časové a paměťové náročnosti jednotlivých algoritmů, ale také například o jejich přednostech.

3.1.1 Sufixový strom

Po provedení předzpracování máme sestavený sufixový strom pro text S a můžeme v něm vyhledávat opakovaně různé vzory, dokud nedojde ke změně ve vstupním textu S. Díky této vlastnosti má sufixový strom velikou výhodu v tom, že toto předzpracování není povinné před každým novým vyhledáváním, a proto je velice užitečný při hledání v textu, ve kterém nedochází k častým změnám.

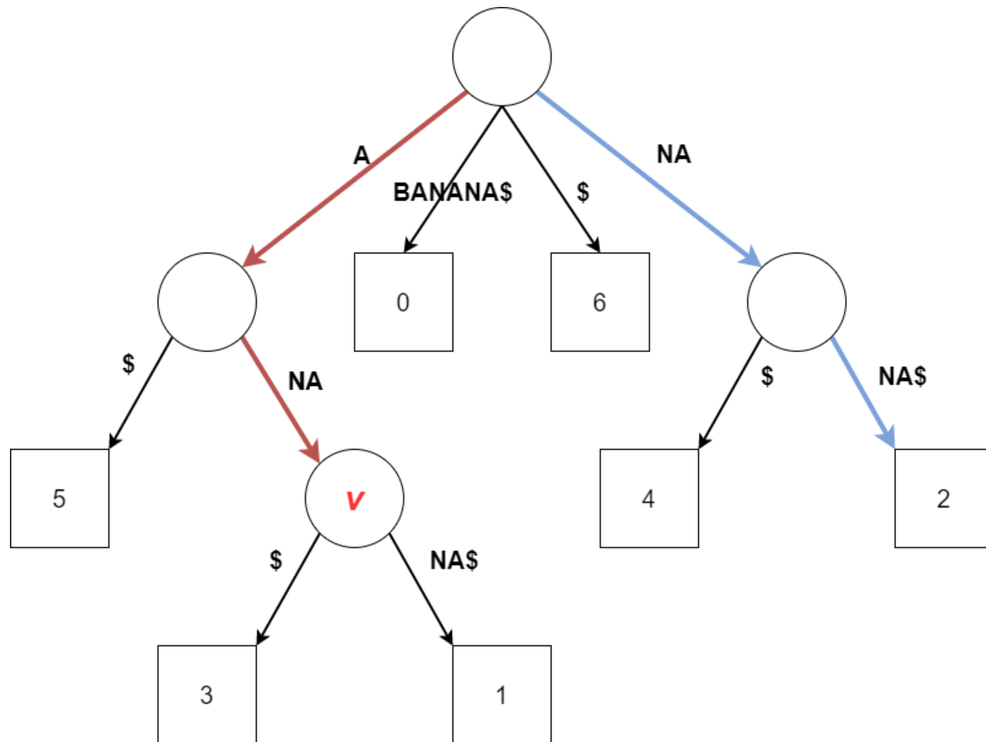
Pokud tedy chceme začít vyhledávat vzor p ve stromě, začínáme vždy v kořeni stromu a postupně porovnáváme znaky p s popisem cesty ve stromě. Nejprve v kořenovém uzlu zjistíme, jestli se v něm nachází hrana, jejíž první znak popisu je shodný s prvním znakem vzoru p. Jakmile jsme tuto požadovanou hranu našli, začneme porovnávat zbylé znaky hrany se znaky p, dokud nedojdeme k dalšímu internímu uzlu v. Jako $|(r, v)|$ označme délku hrany (r, v), kterou jsme právě přešli k internímu uzlu v. V tomto interním uzlu nyní opět hledáme hranu, jejíž počáteční znak je shodný se

znakem p, ležícím na pozici $|(r, v)|$. Tento postup opakujeme, dokud nenajdeme cestu, jejíž popis odpovídá znakům vzoru p.

Definice 3.1.1. Jako ***Threading*** vzoru p nazýváme porovnávání znaků tohoto vzoru p podél jednoznačné cesty ve stromě T, dokud nedojdeme k listu nebo nejsou nalezeny další shodné znaky ve vzoru p. Jako ***Complete threading*** je poté nazýván *threading*, při jehož porovnávání dojdeme až na konec vzoru p. V opačném případě se jedná o *incomplete threading* [19].

Jako α označme požadovanou cestu, jejíž popis je identický se vzorem p. Jakmile úspěšně porovnáme všechny znaky p s α, jedná se o *Complete threading* a s jistotou víme, že se požadovaný vzor p nachází v textu S. Samotná operace zjištění výskytu vzoru p nám zabere $O(n)$ času, kde n je délka p [23]. Jelikož veškerá práce nad stromem je projít jedinečnou cestu α, která je označená řetězcem p, odvíjí se časová náročnost právě od délky této cesty. Pokud budeme chtít zjistit index výskytu, bude záležet, kde cesta α končí. Pokud tato cesta končí na listovém uzlu, nemusíme dál procházet strom a víme, že se vzor p nachází ve stromě přesně jednou a index výskytu je právě v indexu listového uzlu, přičemž časová náročnost $O(n)$ zůstává stejná. Jestliže ovšem cesta α končí uvnitř stromu buď na některém interním uzlu (nazvěme jej v), nebo uvnitř některé nelistové hrany, vzor p se v textu S vyskytuje vícekrát. Abychom zjistili indexy všech výskytů, musíme projít celý podstrom, jehož vrcholem je právě uzel v, a v každém listovém uzlu tohoto podstromu se nachází index výskytu vzoru p. Jelikož ovšem musíme provést toto dodatečné procházení podstromu, časová náročnost se mírně zvětší na $O(n + k)$, kde n je opět délka p a jako k označujeme počet výskytů vzoru p v S.

Pro lepší představu o vyhledávání vzoru si ukážeme sufixový stromu pro text S = "BANANA" na obrázku 3.1.



Obrázek 3.1: Vyhledávání vzoru p_1 = "NANA" (modrá) a p_2 = "ANA" (červená) v S = "BANANA$"

Ve stromě budeme vyhledávat vzory p_1 = "NANA" a p_2 = "ANA". V prvním případě, pro p_1, je cesta ve stromě označena modře a končí na listovém uzlu 2. Tento vzorek se tedy v textu nachází pouze jednou a to právě na indexu 2. V druhém příkladu, kdy vyhledáváme p_2, je požadovaná cesta označena červeně a končí na interním uzlu pojmenovaném v. Ve chvíli kdy dorazíme k uzlu v, víme, že se p_2 ve stromě nachází, ale ještě neznáme indexy těchto výskytů, jelikož interní uzly nejsou označeny žádnými indexy. Abychom tedy věděli, kde se tento p_2 nachází, musíme ještě dodatečně projít všechny listy podstromu, jehož vrchol je právě uzel v. V tomto případě se v podstromě nachází dva listové uzly, které jsou popsány indexy 1 a 3. Vyhledání p_2 zabere $O(n + k)$ času, v této situaci tedy p_2 má délku 3 a počet výskytů k je 2.

3.1.2 Posuvné okno a naivní algoritmus

Máme-li vzor p, který je neprázdný řetěze o délce n, předpokládáme, že text S o délce m, ve kterém bude probíhat vyhledávání, je prozkoumán pomocí posuvného okna. Toto okno reprezentuje určitý text, který nazýváme jako obsah okna a je ve většině případů stejné délky jako p. Toto okno se posouvá skrze text S, od jeho začátku až po konec, zleva doprava.

Pokud se okno zrovna nachází na pozici j v textu S, algoritmus zjistí, jestli se vzor p na této pozici j vyskytuje tak, že porovnává znaky obsahu okna se znaky S, které začínají na pozici j. Toto porovnávání můžeme nazvat jako pokus na pozici j. Pokud je toto porovnání úspěšné, vzor p se v S vyskytuje na pozici j. Během tohoto porovnávání algoritmus může sbírat určité informace, které mohou být následně využity ve dvou směrech:

- Nastavení délky dalšího posunutí okna podle pravidel, které daný algoritmus specifikuje.

- Zapamatovat si část sesbíraných informací k úspoře počtu porovnaných znaků při dalších pokusech.

Když okno posuneme v textu z pozice j na pozici $j + d$, pro $d \geq 1$, má toto posunutí délku d. Pokud provedeme posunutí, jehož délka je $d \geq 2$, musíme mít pro tento krok opodstatnění a musíme si být jisti, že se v textu S na pozici $j + 1$ až $j + d - 1$ nenachází vyhledávaný vzor p.

Naivní algoritmus

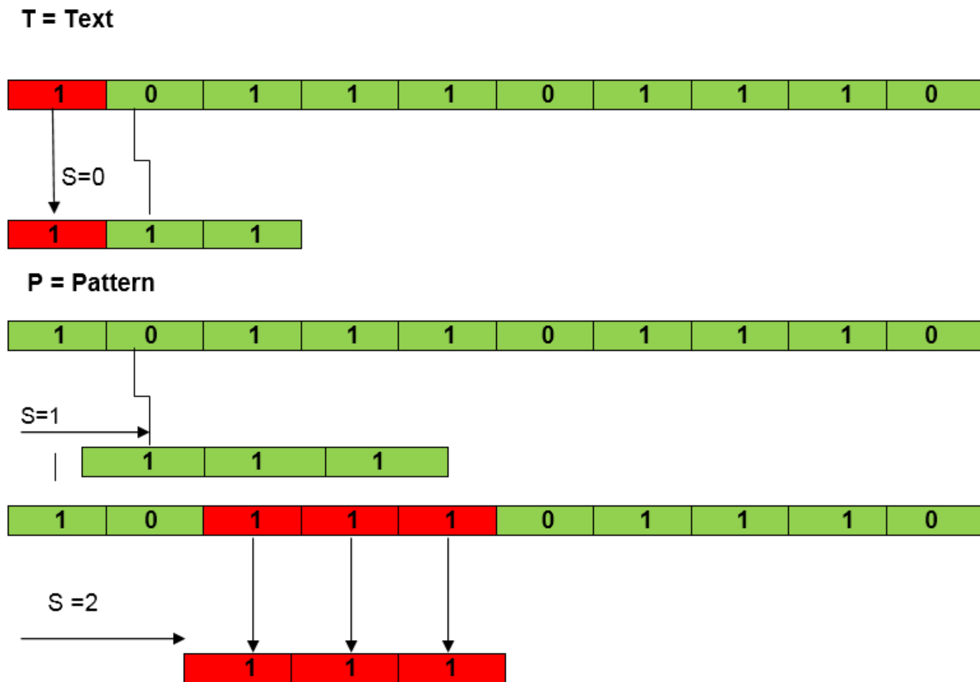
Naivní algoritmus je nejjednodušší implementací techniky posuvného okna. Okno v tomto algoritmu má délku n, podle vyhledávaného vzoru p, a toto okno je posouváno zleva doprava vždy o délku $d = 1$ po každém pokusu. Jelikož nedochází k posunům delším než o jeden znak, dochází k porovnání každého znaku v S a můžeme si být jisti, že algoritmus provádí každé platné porovnání [4].
Níže uvedený pseudokód 1 nám ukáže, jakým způsobem tento algoritmus posouvá okno a porovnává jej s textem S.

Algorithm 1: Pseudokód znázorňující vyhledávání vzoru pomocí naivního algoritmu[1]

```
int n = pattern.Length
int m = S.Length
int j
list indexes
for int i = 0 to m − n do
    for j = 0 to n − 1 do
        if S[i+j] != pattern[j] then
            //Neúspěšný pokus
            break
        end
    end
    if j == n then
        //Úspěšný pokus
        indexes.add(i)
    end
end
```

Algoritmus tedy pro každý pokus posouvá okno vždy o jeden znak a následně porovnává jeho obsah se znaky textu S, počínaje pozicí i. Pokud se znaky liší, tento pokus je neúspěšný, v opačném případě algoritmus zahlásí výskyt vzoru p na pozici i. Posun v textu je znázorněn na obrázku 3.2.

[1] Kód převzat z: https://www.geeksforgeeks.org/naive-algorithm-for-pattern-searching/



Obrázek 3.2: Posouvání okna v textu S při hledání vzoru pomocí naivního algoritmu[2].

Provedení tohoto algoritmu v nejhorším případě je $\Theta(m \times n)$, v situaci, kdy jsou například text S i vzor p mocninami stejného znaku [3]. Jelikož v případě tohoto algoritmu není třeba dělat žádné předzpracování, což znamená, že nepotřebujeme uchovávat v paměti žádná dodatečná data, jeho paměťová náročnost je nulová. Průměrný čas běhu je ovšem poměrně dobrý, jak vyplývá z následujícího tvrzení.

Tvrzení Za předpokladu, že abeceda není zredukovaná na jediný znak a také za předpokladu rovnoměrného a nezávislého rozložení znaků abecedy, je průměrný počet pokusů provedený Naivním algoritmem, pro S délky m a p délky n, $\Theta(m - n)$ [4].

Důkaz Jako c označme velikost abecedy. Počet porovnání jednotlivých znaků nutných k určení, jestli jsou dva řetězce u a v velikosti m identické je v průměru

$$1 + 1/c + ... + 1/c^{m-1}$$

nezávisle na permutaci pozic uvažovaných pro porovnání znaků řetězců. Pokud je $c \geq 2$, toto množství je méně než $1/(1 - 1/c)$, což nikdy není více než 2.

Z toho vyplývá, že průměrný počet porovnání znaků během provádění operací je méně než $2 \times (m - n + 1)$. Tudíž tento výsledek platí, jelikož je provedeno alespoň $m - n + 1$ porovnání [4].

3.1.3 Knuth–Morris–Pratt algoritmus

Knuth–Morris–Pratt (KMP) algoritmus je další často používaný algoritmus při hledání vzorů v textu. Ve své práci jej v roce 1977 popsali pánové Donald Knuth, James H. Morris a Vaughan Pratt

[2]Zdroj obrázku: https://www.javatpoint.com/daa-naive-string-matching-algorithm

jako vyhledávací algoritmus s lineární náročností [16]. Taktéž se jako v případě naivního algoritmu jedná o implementaci vyhledávání pomocí posuvného okna s tím rozdílem, že se již implementuje sběr dat, který nám pomáhá s posouváním tohoto okna o větší vzdálenost a tím pádem k rychlejšímu projití textu. Jakým způsobem se vypočítává délka jednotlivých posunů a jaká je časová a prostorová náročnost si přiblížíme níže.

U naivního algoritmu je funkčnost implementována tak, že porovnáme prvních i znaků vzoru p se znaky textu S na odpovídajících pozicích, počínaje pozicí j. Jakmile dojde k neshodě některého znaku, algoritmus posune okno pouze o jednu pozici, čili na $j+1$ a začne opět porovnávat od levého konce vzoru. Vzhledem k podobě vzoru p by ovšem často bylo možné udělat větší posun. Díky tomu, že tyto větší posuny provádíme, nám Knuth–Morris–Pratt algoritmus umožňuje provádět vyhledávání vzoru p v textu S o délce m v čase $\Theta(m)$ [16]. Aby toho byl algoritmus schopen, využívá k tomu předzpracování, jehož časová náročnost je $\Theta(n)$, čili je rovna délce vzoru. Od toho se odvíjí také prostorová náročnost algoritmu, která je taktéž $\Theta(n)$.

Pokud máme například p = "abcxabcde", který porovnáváme s textem S a neshoda nastane na pozici i = 7, tedy na znaku 'd', pak můžeme okno posunout o čtyři pozice, jelikož před znakem 'd' se nachází znaky "abc", které jsou zároveň i na začátku p a byly již úspěšně porovnány. To znamená, že pokud přeskočíme porovnání těchto tří znaků, nemusíme je opět porovnávat, ale přesto si můžeme být jisti, že se tam nachází. Je také důležité všimnout si, že tento předpoklad o délce posunu jsme byli schopni udělat bez toho, abychom znali podobu textu S, nebo aktuální polohu j okna. Jediná informace, kterou musíme znát, je poloha znaku v okně, kde došlo k neshodě s S. Knuth–Morris–Pratt je založen na tomto výpočtu délky posunu, aby mohl přeskakovat nepotřebné porovnávání a k tomuto výpočtu potřebuje $\Theta(n)$ času a paměti [16]. To jak dlouhý bude posun, nám určuje pole, které má délku totožnou s p a vytváříme jej dle následující definice.

Definice 3.1.2. Pro každou pozici i ve vzoru p, definujme sp_i, jako délku nejdelšího vlastního sufixu vzoru $p[0..i]$, který je stejný s prefixem vzoru p [12].

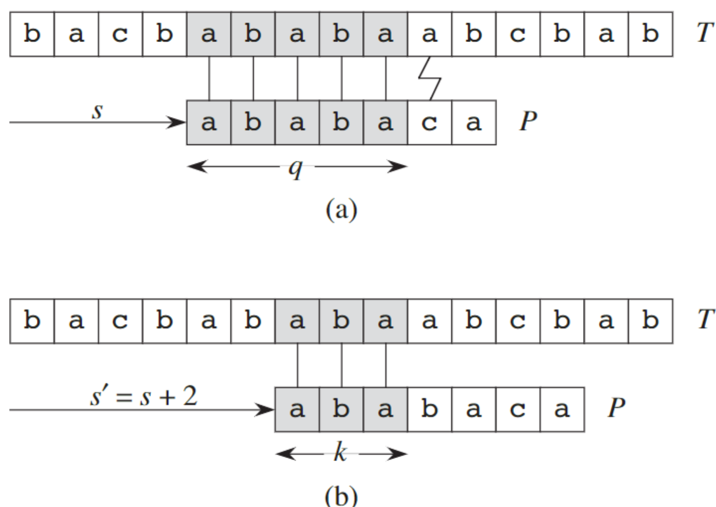
Jako sp_i (taktéž označováno jako $\pi[i]$) tedy označujeme délku nejdelšího vlastního sufixu $p[0..i]$, který končí na pozici i a shoduje se s prefixem p. Pokud máme například p = "abcaeabcabd", tak $sp_1 = sp_2 = 0$, $sp_3 = 1$, $sp_7 = 3$ a $sp_9 = 2$. sp_0 je vždy rovno nule. Obrázek 3.3 nám ukazuje, jaké jsou hodnoty sp_i pro řetězec "ababaca".

i	0	1	2	3	4	5	6
$P[i]$	a	b	a	b	a	c	a
$\pi[i]$	0	0	1	2	3	0	1

Obrázek 3.3: Funkce $\pi[i]$ (sp_i) pro vzor "ababaca" u Knuth–Morris–Pratt algoritmu. Zdroj obrázku: [3].

Jakmile při průběhu porovnávání p a S dojde k neshodě znaků na pozici $i+1$ ve vzoru a pozici k v textu, tak posun okna je o $i - sp_i + 1$ doprava, aby znak sp_i vzoru p byl zarovnán se znakem k textu S. V případě, že byl vzor p nalezen (žádná neshoda znaků), okno posuneme o $n - sp_{n-1}$ míst.

Tento způsob posunu má dvojí výhodu. Často je okno se vzorem p posunuto o více než jeden znak. A také po posunutí okna, u prvních sp_i znaků zleva vzoru p je zaručeno, že se rovnají se znaky na odpovídajících pozicích textu S. Což znamená, že pokud porovnáváme nově posunutý vzor p s odpovídajícími znaky S, můžeme začít s porovnáváním na pozici $sp_i + 1$ v p a pozici k v S. Obrázek 3.4 zobrazuje, jak vypadá posun okna.



Obrázek 3.4: V části a došlo k neúspěšnému porovnání znaku na pozici 5 $(i + 1)$ ve vzoru p a na pozici $k = 9$ v textu S. Okno posuneme o 2 pozice, podle výpočtu $i - sp_i + 1$, kde $i = 4$ a $sp_4 = 3$ a následně opět porovnáváme se znakem na pozici k v S. Zdroj obrázku: [3].

Tvrzení Za použití Knuth–Morris–Pratt algoritmu je počet porovnaných znaků nanejvýš $2m$.

Důkaz Rozdělíme-li algoritmus na porovnávací a posouvací fáze, kde se jedna fáze skládá z porovnání mezi úspěšnými posuny. Po každém posunu porovnáváme v dané fázi zleva doprava a začínáme buď posledním znakem S, který byl porovnávám v předchozí fázi, nebo se znakem, který se od něj nachází napravo. Jelikož okno reprezentující vzor p nikdy neposouváme směrem doleva, tak v kterékoliv fázi nanejvýš jedno porovnání zahrnuje znak, který byl porovnáván již v dřívější fázi. To znamená, že celkový počet znaků, které se porovnávají, se odvíjí od $m + S$, kde m je délka textu S a S je počet posunů, které se v algoritmu provedly. Je ovšem dáno, že $S < m$, jelikož po m posunech je pravý konec p s jistotou napravo od pravého konce S, takže počet provedených porovnání má horní hranici $2m$ [12].

3.1.4 Rabin-Karp algoritmus

Jako poslední si ukážeme algoritmus navržený doktorem Richardem M. Karpem a doktorem Michaelem O. Rabinem, kteří jej představili v roce 1987, jako nový algoritmus, který je schopen najít výskyt vzoru v textu v lineárním čase [15]. Je to další modifikace postupu založeného na posuvném okně. Algoritmus nejprve provádí předzpracování v čase $\Theta(n)$ a čas běhu vyhledávání je v nejhorším případě $\Theta((m - n + 1)n)$, ovšem jeho průměrný čas běhu je mnohem lepší, konkrétně $\Theta(n + m)$ [3].

V algoritmu reprezentujeme řetězce délky n pomocí mnohem kratších řetězců zvaných otisky (z anglického *"fingerprints"*). Tyto otisky pro dané podřetězce vytváříme pomocí hashovací funkce.

Představme si tedy, že máme text S, ve kterém budeme vyhledávat, a vyhledávaný vzor p, který je délky n. Vytvoříme tedy nějakou hashovací funkci, která podřetězcům délky n přiřazuje čísla z množiny $\{0, ..., q-1\}$ pro číslo q, které je dostatečně velké. Opět budeme posouvat okno délky n po textu S a pro každou polohu proběhne výpočet jeho obsahu pomocí hashovací funkce a ten pak porovnáme s hashem vyhledávaného vzoru p. Pokud se tyto dva hashe rovnají, je následně důležité ještě porovnat obsah okna se vzorem p.

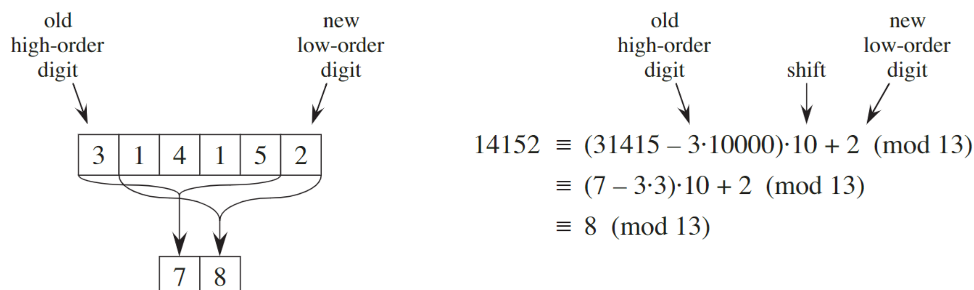
Pokud je ovšem vytvořená hashovací funkce dostatečně propracovaná a komplexní, není třeba porovnávat obsah okna se vzorem p, ale shoda hashů pro nás bude dostatečným argumentem. V tom případě už nebude potřebný čas na porovnání $\Theta(n)$, ale stane se toto porovnání proveditelné v čase konstantním [17]. Stále však budeme potřebovat určitý čas $\Theta(n)$, abychom vypočítali hash pro obsah okna. Abychom tento čas hashováním nemuseli strávit při každém posunutí okna, je možné hashovací funkci upravit tak, aby bylo každé další přepočítání hashe při posunu okna možno udělat v konstantním čase.

Abychom ovšem byli schopni těchto operací dosáhnout v konstantním čase, nesmíme zapomínat, že tyto hashe, které reprezentují jednotlivé podřetězce, mohou být příliš velké. Pokud tyto operace provádíme nad obsahem okna o délce n znaků, nemůžeme předpokládat, že čas těchto operací bude konstantní. Tato situace se však dá jednoduše vyřešit použitím operace modulo q, čímž rozsah snížíme do požadovaného intervalu. Velikost tohoto čísla q, se kterým budeme provádět operaci modulo, bychom měli vybrat tak, aby se $10q$ vešlo do datové jednotky *word* [3]. Vzorec pro výpočet hashe je tedy:

$$t_{j+1} = (d(t_j - S[j]h) + S[j+n]) mod\, q$$

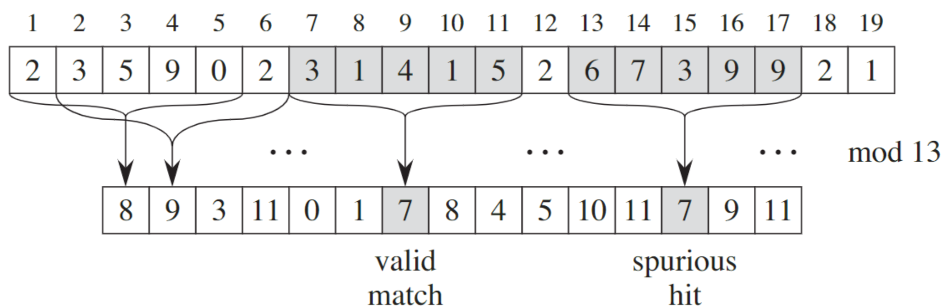
Pokud posuneme okno na pozici $j + 1$, hash pro tuto pozici nazýváme jako t_{j+1} a vypočítáváme jej na základě hashe z předchozí pozice j čili z t_j. Jestliže tedy chceme vypočítat hash pro následující pozici, aniž bychom museli provádět znovu celý výpočet, z původního hashe odebereme hodnotu znaku na pozici j ($S[j]$) a přidáme hodnotu znaku na pozici $j + n$. Jako d označujeme počet znaků abecedy, ze které se text S skládá, a jako h je označeno číslo, které získáme následujícím výpočtem: $h = d^{n-1}$.

Na obrázku 3.5 si ukážeme, jakým způsobem se pomocí hashovací funkce vypočítávají hodnoty jednotlivých podřetězců.



Obrázek 3.5: Výpočet pomocí hashovací funkce Rabin–Karpova algoritmu. V tomto případě se $d = 10$ a $q = 13$. Zdroj obrázku: [3].

Obrázek 3.6 znázorňuje text S, jehož všechny podřetězce délky n byly přepočítány hashovací funkcí.



Obrázek 3.6: Hodnoty vypočítané hashovací funkcí pro text S = "2359023141526739921". Na indexech 7 a 13 je hash identický, avšak podřetězce jsou jiné, a proto je důležité následně srovnat i vzor p a podřetězec z S v původní podobě. Zdroj obrázku: [3].

Ačkoliv Rabin-Karp algoritmus není nejvýkonnější při hledání vzorů v textu, jelikož vypočítávání hashů může být náročná operace, je tento algoritmus známý především pro svoje hojné využití v oblasti odhalování plagiátorství. Pokud máme k dispozici zdrojové materiály, tento algoritmus může rychle prohledat kontrolovaný článek a nalézt v něm instance vět ze zdrojových materiálů. Zároveň může ignorovat detaily, jako například interpunkci, nebo velká a malá písmena. Jelikož při tomto užití je počet vyhledávaných řetězců enormní, jsou ostatní algoritmy pro vyhledávání jednotlivých řetězců nepraktické.

3.2 Nejdelší opakující se podřetězec

V informatice označujeme jako problém nejdelšího opakujícího se podřetězce (*longest repeated substring*) hledání nejdelšího podřetězce, který se v textu S vyskytuje alespoň dvakrát. Dva či více řetězce v textu S nazýváme jako nejdelší opakující se podřetězce, pokud jsou naprosto identické a jejich prodloužením o další znaky by byla narušena jejich identičnost (musí být maximální). Je to další problematika, kterou nám umožňuje sufixový strom řešit. K řešení tohoto problému můžeme použít také dynamické programování. Opět si popíšeme oba dva postupy a řekneme si něco o jejich časové náročnosti.

S touto problematikou se můžeme setkat v bioinformatice, kde v posloupnosti genů můžeme hledat opakující se podřetězce. Tyto opakující se části se mohou vyskytovat buď přilehlé k sobě v textu S (tzv. *tandem repeats*), nebo ve větší vzdálenosti od sebe. Některé z těchto přilehlých opakujících se podřetězců sestávají z krátkých subsekvencí, nazývaných jako mikrosatelity (*microsatellites*) a v dnešní bioinformatice jsou hojně využívané u genetického mapování [20].

Hledání těchto opakujících se struktur je důležitá problematika u biologických sekvencí, jelikož většina chromozomu Y v lidském těle je tvořena z opakujících se subsekvencí. Mimo jiné je také mnoho chorob způsobeno strukturálními opakováními.

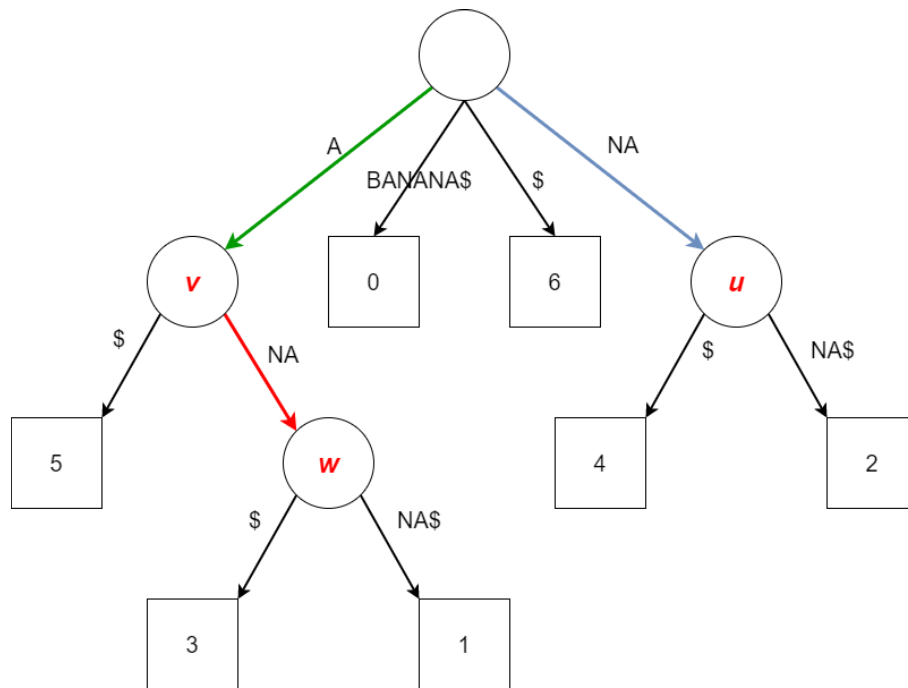
Dále se také se sufixovými stromy můžeme setkat u algoritmů provádějících kompresi dat. Tyto algoritmy totiž hledají opakující se data ve vstupním řetězci a provádí kompresi. Sufixový strom například využívá Burrowsova–Wheelerova transformace [2].

3.2.1 Sufixový strom

Jak již bylo řečeno, sufixový strom nám umožňuje řešit problém nejdelšího opakujícího se podřetězce. Navíc je toto vyhledávání možné bez jakékoliv provedené změny v konstrukci stromu.

Připomeňme si, jak vypadá struktura sufixového stromu a jak jsou v něm vyjádřeny jednotlivé podřetězce. Víme, že označení cesty, která vede k libovolnému internímu uzlu, je podřetězec textu S, který se v něm vyskytuje alespoň dvakrát, jelikož definice 2.1.1 nám určuje, že každý interní uzel má alespoň dva potomky. Čili tato cesta může pokračovat ke dvěma či více listovým uzlům, respektive podřetězec, který reprezentuje, se nachází na více indexech. Pokud by tato cesta vedla k uzlu listovému, výskyt podřetězce by už nebyl vícenásobný.

Abychom tedy našli tento nejdelší opakující se podřetězec, musíme najít nejhlubší interní uzel. Jelikož však pracujeme se sufixovým stromem, hloubku neurčuje počet interních uzlů, které se nacházejí nad daným uzlem, ale délka cesty, která k internímu uzlu vede. Vyhledávání tohoto podřetězce probíhá až v sestaveném stromu, což znamená, že tento průchod a vyhledávání nejhlubšího uzlu provádíme až v sestaveném stromě. Na obrázku 3.7 si ukážeme, jak toto prohledávání vypadá v textu S = "BANANA$".



Obrázek 3.7: Nejdelší opakující se podřetězec, nalezený sufixovým stromem, v S = "BANANA$" je "ANA" (interní uzel w, ke kterému vede zeleno-červená cesta).

Ve stromu jsou celkem tři interní uzly reprezentující podřetězce, které se v textu S vyskytují vícekrát. Uzel v, ke kterému vede zelená cesta, reprezentuje podřetězec "A". Tento podřetězec sice má v textu S největší počet výskytů, jenže není nejdelší. Dalším interním uzlem je u s modrou cestou označenou podřetězcem "NA", který ovšem taktéž neoznačuje nejdelší podřetězec. Nejdelší opakující se podřetězec se tedy nachází na uzlu w, k němuž vede zeleno-červená cesta, která je označena podřetězcem "ANA". Tento podřetězec je nejdelší a zároveň je opakující se, jelikož se vyskytuje na indexu 1 a 3.

Nalezení nejdelšího opakujícího se podřetězce v textu S o délce m, nám za použití sufixového stromu zabere $O(m)$ času [23]. Tento čas nezahrnuje konstrukci stromu, která taktéž zabere dalšího $O(m)$ času [12].

3.2.2 Dynamické programování

K vyřešení problému se také může využít dynamického programování. Tento postup využívá dvourozměrného pole, které uchovává informace o délce jednotlivých podřetězců, nacházejících se v textu S, které se opakují. Právě kvůli tomuto dvourozměrnému poli však razantně stoupá paměťová náročnost algoritmu.

Algoritmus v podstatě hledá nejdelší opakující se sufix, pro všechny prefixy v textu S. Následující pseudokód 2 ukazuje, jak tento algoritmus pracuje:

Algorithm 2: Pseudokód znázorňující vyhledání nejdelšího opakujícího se podřetězce pomocí dynamického programování[3]

```
int LRS[m+1][m+1]
int max
int maxIndex
for i = 1 to m do
    //j = i + 1 znamená, že začínáme až vedle (nad) hlavní diagonály a prvky na ní
      nevyplňujeme
    for j = i + 1 to m do
        if S[i−1] == S[j−1] then
            LRS[i, j] = LRS[i−1, j−1] + 1
            if LRS[i, j] > max then
                max = LRS[i, j]
                maxIndex = i
            end
        else
            LRS[i, j] = 0
        end
    end
end
return S.substring(maxIndex – max, max)
```

Pomocí těchto dvou vnořených cyklů vyplňujeme již dříve zmíněné dvourozměrné pole, do kterého vkládáme délky opakujících se podřetězců. Už při tomto průchodu pole si uchováváme data o dosavadní maximální délce, abychom toto pole nemuseli následně procházet znovu. První sloupec a první řádek musíme nechat nulové, abychom se při porovnávání prvních znaků nesnažili přistoupit za hranice indexů pole. Algoritmus musí počítat s tím, že hodnoty na hlavní diagonále musí zůstat nevyplněny, jelikož na těchto souřadnicích se nachází celý text S o délce m a ten není opakující se. Výsledné pole je zobrazeno na obrázku 3.8.

[3] Kód převzat z: https://www.geeksforgeeks.org/longest-repeating-and-non-overlapping-substring/

		B	A	N	A	N	A
	0	0	0	0	0	0	0
B	0	0	0	0	0	0	0
A	0	0	0	0	1	0	1
N	0	0	0	0	0	2	0
A	0	0	0	0	0	0	3
N	0	0	0	0	0	0	0
A	0	0	0	0	0	0	0

Obrázek 3.8: Dvourozměrné pole, které obsahuje délku opakujících se podřetězců. Vstupní text S je "BANANA" a jeho nejdelší opakující se podřetězec "ANA" má délku 3. Červenou barvou jsou označeny pole hlavní diagonály, které musí zůstat nulové.

V textu je tedy nejdelší opakující se podřetězec délky 3 a v poli se záznam o této délce nachází na pozici $i = 4$ a $j = 6$. Pozici tohoto podřetězce získáme odečtením délky (tedy 3) od indexů v poli (4 a 6), poté získáme pozice výskytů, které jsou 1 a 3.

Musíme však pro text o délce m znaků vytvořit matici o velikosti $m \times m$, tedy m^2. Tato skutečnost ovlivňuje jak časovou, tak na prostorovou náročnost, které jsou v obou případech $O(m^2)$ [14].

3.3 Nejdelší společný podřetězec

Toto vyhledávání se provádí nad dvěma, nebo více texty, kde hledáme jejich nejdelší společný podřetězec. Postup u této problematiky je podobný jako u nejdelšího opakujícího se podřetězce, musíme však počítat právě s více texty. U sufixového stromu musíme přizpůsobit konstrukci, abychom mohli vytvořit strom, který obsahuje více textů (generalizovaný sufixový strom popsaný v kapitole 2.5).

V oblasti bioinformatiky se tato možnost využije, pokud hledáme určité souvislosti mezi dvěma sekvencemi. Pokud hledáme nejdelší společný podřetězec, hledáme jej právě mezi těmito sekvencemi. Doktor Donald Ervin Knuth v roce 1970 avizoval, že vyřešit problém nejdelšího společného podřetězce je nemožné v lineárním čase [5]. Avšak s příchodem sufixových stromů a za použití zmíněného generalizovaného sufixového stromu jsme schopni tuto problematiku v lineárním čase vyřešit.

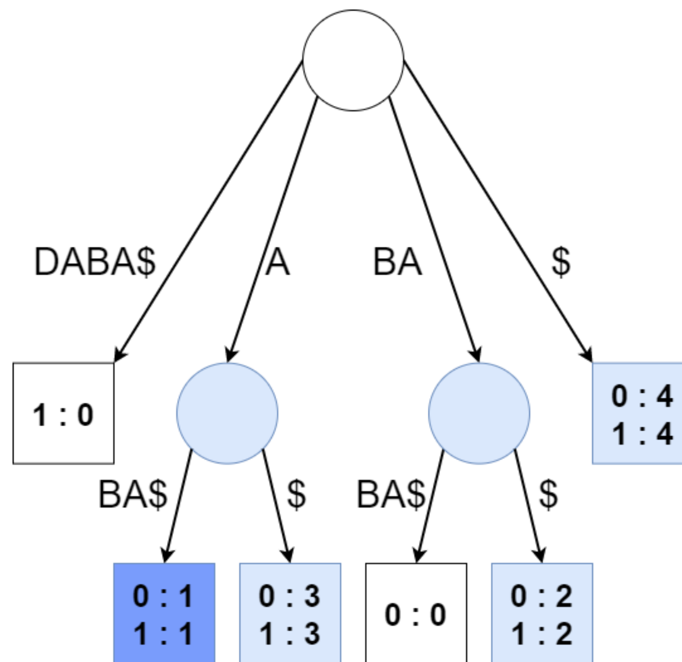
3.3.1 Sufixový strom

Abychom mohli najít nejdelší společný podřetězec, nestačí nám k tomu běžný sufixový strom, který jsme dosud používali pro řešení předchozí problematiky nad řetězci. Nyní už musíme upravit způsob, jakým strom sestavujeme a navíc i některé vlastnosti stromu samotného. K řešení tedy musíme využít generalizovaného sufixového stromu, o kterém pojednává kapitola 2.5.

Ve stromě poté vyhledáváme nejdelší společný podřetězec podobně, jako jsme vyhledávali u problematiky nejdelšího opakujícího se podřetězce. Musíme najít nejhlubší listový uzel, který reprezentuje sufixy ze všech vstupních řetězců, anebo musíme najít alespoň nejhlubší interní uzel,

je ovšem potřeba zaručit, že v podstromu tohoto interního uzlu se nachází listové uzly, které reprezentují sufixy všech řetězců, ze kterých se generalizovaný sufixový strom skládá. Tuto kontrolu musíme provádět, abychom zaručili, že se tento nejdelší podřetězec opravdu nachází ve všech vstupních řetězcích. Popřípadě je možno si vybrat jenom některé texty, ve kterých chceme najít nejdelší společný podřetězec.

Vyhledávání nejdelšího společného podřetězce si opět ukážeme na příkladu. Budeme hledat nejdelší společný podřetězec v řetězcích S_0 = "BABA" a S_1 = "DABA":



Obrázek 3.9: Generalizovaný strom pro texty S_0 = "BABA$0" a S_1 = "DABA$1". Světle modrou barvou jsou označeny uzly, reprezentující podřetězce z obou textů. Tmavě modrou barvou je označen uzel, který reprezentuje podřetězce (v tomto případě sufixy, jelikož se jedná o uzel listový) z obou vstupních textů a má největší hloubku. Nejdelší společný podřetězec těchto textů je "ABA".

V obrázku 3.9 jsou modrou barvou označeny uzly, které reprezentují podřetězce z obou textů. Právě z těchto modrých uzlů budeme hledat ten s největší hloubkou. Musíme ovšem myslet na to, že pokud některá hrana vedoucí k takovému uzlu má označení končící zvláštním zakončovacím znakem '$\$$', tento znak se ke hloubce konkrétního uzlu nepřičítá. Takže ačkoliv listový uzel, který je na obrázku první zprava, obsahuje podřetězec z obou textů (v tomto případě se jedná o reprezentaci prázdného podřetězce), jeho výsledná hloubka je 0. Nejdelší společný podřetězec je tedy "ABA" a je délky 3.

Díky generalizovanému sufixovému stromu můžeme tento problém vyřešit s časovou náročností $O(m_0 + m_1)$, čili s lineární náročností, kde m_0 a m_1 jsou délky řetězců S_0 a S_1 [12]. Tento čas nezahrnuje konstrukci stromu, kterou pomocí Ukkonenova algoritmu provedeme za $O(m_0 + m_1)$ času [12].

3.3.2 Dynamické programování

Problém je také řešitelný za pomoci dynamického programování. Podobně jako u předchozího problému k vyřešení používáme dvourozměrné pole, což ovšem vede k problému, že pomocí tohoto pole můžeme řešit pouze problém nejdelšího společného podřetězce dvou řetězců. Abychom mohli řešit problém více řetězců, museli bychom použít vícerozměrná pole, jejichž rozměr by záležel na počtu řetězců. Kvůli této skutečnosti není tento postup univerzální a kód by se musel měnit v závislosti na počtu řetězců, ve kterých chceme vyhledávat, a také by se výrazně zvětšovala časová a paměťová náročnost, která by byla už pro tři řetězce kubická, pro čtyři řetězce kvartická atd.

Postup je obdobný jako u vyhledávání nejdelšího opakujícího se podřetězce. Pole zaplňujeme celočíselnými hodnotami, které označují délku podřetězců nacházejících se v obou řetězcích. Pole procházíme pomocí dvou vnořených cyklů, kde aktuální řádek označujeme jako i a sloupec jako j. Hodnotu pro políčko na pozici $[i, j]$ vypočítáme tak, že porovnáme znaky $S_0[i-1]$ a $S_1[j-1]$ a jsou-li identické, do pole na pozici $[i, j]$ vložíme hodnotu z pozice $[i-1, j-1]$ a přičteme 1. V opačném případě, tedy pokud jsou znaky odlišné, na pozici $[i, j]$ vložíme 0. Už při tomto průchodu pole si uchováváme data o dosavadní maximální délce, abychom toto pole nemuseli následně procházet znovu. První řádek a první sloupec zůstávají nulové a cyklem začínáme až na $i = 1$ a $j = 1$. Tímto způsobem máme ošetřen přístup za hranici indexů pole.

Největší číslo v poli nám poté říká délku nejdelšího společného podřetězce a z indexů pole, ve kterém se toto největší číslo nachází, zjistíme pozici podřetězce ve vstupních řetězcích. V případě nejdelšího společného podřetězce už však procházíme a vyplňujeme celé pole, nikoliv pouze část nad hlavní diagonálou.

Pole pro texty S_0 = "BABA" a S_1 = "DABA" je znázorněno na obrázku 3.10.

		B	A	B	A
	0	0	0	0	0
D	0	0	0	0	0
A	0	0	1	0	1
B	0	1	0	2	0
A	0	0	2	0	3

Obrázek 3.10: Dvourozměrné pole vytvořené za pomoci dynamického programování pro S_0 = "BABA" a S_1 = "DABA". Červenou barvou je označeno políčko s největším číslem čili políčko s nejdelším podřetězcem. Nejdelší společný podřetězce je tedy "ABA", jehož délka je 3.

Políčko s nejvyšším číslem se nachází na pozici $i = 4$ a $j = 4$. Pozici tohoto nejdelšího společného podřetězce získáme odečtením délky od těchto indexů a v obou případech dostaneme číslo 1,

což je index, na kterém se podřetězec nachází v textu S_0 i S_1.

Časová náročnost tohoto postupu je $O(m_0 \times m_1)$, kde m_0 je délka S_0 a m_1 je délka S_1 [9]. Tato náročnost je způsobena tím, že musíme projít dvourozměrné pole o velikosti $m_0 \times m_1$, což určuje také paměťovou náročnost, která je taktéž $O(m_0 \times m_1)$.

3.4 Nejdelší palindromický podřetězec

Dalším problémem, na který se zaměříme, je hledání palindromu v textu. Nejdelší palindrom sice vyhledáváme pouze v jednom textu, ale algoritmy, které k vyhledávání používáme, vyhledávají nad dvěma řetězci, jak si popíšeme níže. Opět si popíšeme postup pomocí sufixového stromu a postup pomocí dynamického programování. Oba dva algoritmy jsou však velice podobné vyhledávání nejdelšího společného podřetězce.

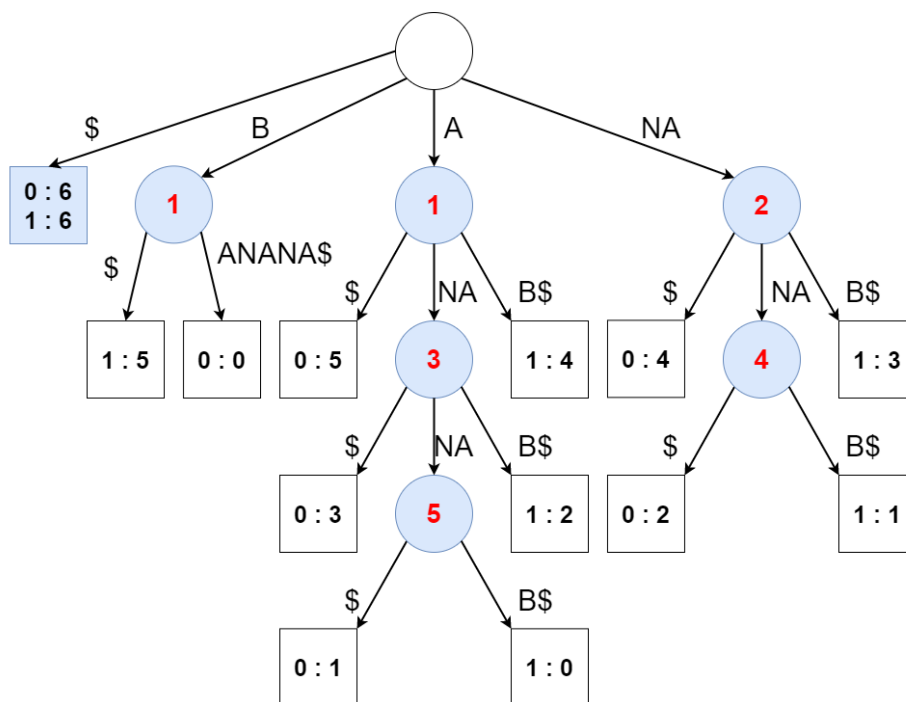
Jako palindrom v textu S označujeme podřetězec u, u nějž platí, že $u = u^r$, kde u^r je obracený podřetězec u, jinými slovy se palindromický řetězec čte stejně zleva doprava jako zprava doleva. Například u námi často používaného řetězce "BANANA" je nejdelší palindromický podřetězec u = "ANANA", jelikož rovněž u^r = "ANANA". Palindrom $u = S[i..i + |u| - 1]$ nazýváme jako maximální pouze tehdy, pokud už nejde rozšířit, čili podřetězec $u = S[i..i + |u|]$ už není palindrom. Pokud najdeme všechny tyto maximální palindromy, můžeme pomocí nich vyjádřit každý palindrom, který se v textu nachází.

Řešení této problematiky je rovněž požadováno v oboru bioinformatiky, kdy hledáme sekvence základních párů ve vzorcích DNA, které jsou stejné i v obrácené podobě a mají tedy vlastnost palindromu. Enzymy, které štěpí tato specifická místa, se nazývají restrikční enzymy.

3.4.1 Sufixový strom

Stejně jako u předchozího řešeného problému nejdelšího společného podřetězce budeme využívat generalizovaného sufixového stromu. Ačkoliv nejdelší palindrom hledáme v jednom řetězci S, budeme sestavovat strom pro dva řetězce. Jednak řetězec S v klasické podobě a následně pro řetězec S^r, což je řetězec, který získáme obrácením řetězce S. Po sestavení tohoto generalizovaného sufixového stromu budeme opět vyhledávat nejhlubší listový uzel reprezentující oba řetězce, nebo interní uzel, v jehož podstromu se nachází listy, které reprezentují podřetězce z S i S^r. Jakmile najdeme tento uzel, našli jsme tím i nejdelší palindromický podřetězec.

Opět si zde uvedeme příklad toho, jak takový sufixový strom vypadá a kde se v něm tento nejdelší palindrom nachází. Na obrázku 3.11 je strom sestavený pro text S = "BANANA$":



Obrázek 3.11: Generalizovaný sufixový strom pro řetězce S = "BANANA$" a S^r = "ANANAB$". Modrou barvou jsou označeny uzly, reprezentující podřetězce z obou textů a čísla červenou barvou označují hloubku uzlu. Nejdelší palindromický podřetězec tohoto řetězce je tedy "ANANA".

Opět máme modře vyznačeny interní uzly stromu, v jejichž podstromech se nachází listy obou vkládaných řetězců S i S^r, nebo listové uzly, které reprezentují sufixy z obou textů. Nejhlubší interní uzel má v tomto případě hloubku 5 a cesta k němu reprezentuje podřetězec "ANANA", což je také nejdelší palindrom ve vstupním řetězci "BANANA". Opět musíme myslet na to, že k hloubce uzlu se nepřičítá zakončovací znak '$'.

Čas, ve kterém zvládneme toto hledání provést je $O(m)$, kde jako m označujeme součet délek řetězců S a S^r [20]. Tento čas nezahrnuje konstrukci stromu, kterou pomocí Ukkonenova algoritmu provedeme taktéž za $O(m)$ času [12].

3.4.2 Dynamické programování

Dalším způsobem vyhledání nejdelšího palindromu je postup pomoci dynamického programování. Vyhledávání palindromu v textu S je velice podobné jako vyhledávání nejdelšího společného podřetězce. Opět vytváříme dvourozměrné pole, které naplníme čísly reprezentující délku podřetězce, jež se nachází v obou řetězcích. Toto pole vytváříme pro řetězec S a jeho obrácenou podobu S^r, stejně jako je tomu u sufixového stromu. Výpočet délky těchto podřetězců nevyplňujeme pro celé pole, nýbrž pouze pro pozice na hlavní diagonále a nad ní. Políčka pod hlavní diagonálou nevyplňujeme a necháme v nich nulové hodnoty.

Pole procházíme pomocí dvou vnořených cyklů, kde aktuální řádek označujeme jako i a sloupec jako j. První sloupec a první řádek jsou opět zaplněny nulovými hodnotami a vnořené cykly je neprochází. Hodnotu pro políčko na pozici $[i, j]$ vypočítáme tak, že porovnáme znaky $S[i-1]$ a $S^r[j-1]$ a jsou-li identické, do pole na pozici $[i, j]$ vložíme hodnotu z pozice $[i-1, j-1]$ a přičteme 1. V opačném případě, tedy pokud jsou znaky odlišné, na pozici $[i, j]$ vložíme 0. Už při tomto průchodu pole si uchováváme data o dosavadní maximální délce, abychom toto pole nemu-

seli následně procházet znovu.

Obrázek 3.12 zobrazuje výsledné pole, pro vyhledávání nejdelší palindromu v řetězci S = "BANANA".

		B	A	N	A	N	A
	0	0	0	0	0	0	0
A	0	0	1	0	1	0	1
N	0	0	0	2	0	2	0
A	0	0	0	0	3	0	3
N	0	0	0	0	0	4	0
A	0	0	0	0	0	0	5
B	0	0	0	0	0	0	0

Obrázek 3.12: Dvourozměrné pole vytvořené za pomoci dynamického programování pro S = "BANANA" a S^r = "ANANAB". Červenou barvou je označeno políčko s největším číslem. Nejdelší palindromický podřetězce je tedy "ANANA", který je dlouhý 5 znaků.

Oproti sufixovému stromu je tento postup mnohem náročnější a to časovou i paměťovou náročností. Jelikož pro řetězec délky m musíme vytvořit matici velikosti $m \times m$, je paměťová náročnost kvadratická čili $O(m^2)$ [7]. To stejné platí i pro časovou náročnost, která je taktéž $O(m^2)$.

Kapitola 4

Návrh aplikace

V této kapitole si popíšeme, jaké jsou požadavky na aplikaci a také návrh této aplikace. Aplikace má pomocí grafického rozhraní uživateli umožnit využívat funkce sufixového stromu a zároveň zobrazit informace o porovnání jednotlivých algoritmů. Jedna z částí návrhu je vytvoření přehledného shrnutí funkcí sufixového stromu a také shrnutí informací o porovnání této struktury s ostatními algoritmu, které jsme si popsali v předchozí kapitole. Grafické rozhraní má tedy hlavní účel uživateli umožnit využít veškeré funkce, které sufixový strom nabízí. Dále jsem také při návrhu počítal s testy, které budou měřit časovou výkonnost jednotlivých algoritmů a jejich výstup bude moci být zobrazen v grafech.

4.1 Požadavky na aplikaci

Hlavním důvodem, proč tato aplikace vznikla, je přiblížit uživateli širokou škálu možností a operací, kterou nám sufixové stromy poskytují. Prvním a nejdůležitějším z požadavků tedy byla implementace sufixového stromu, což obnáší vytvoření této struktury a implementaci algoritmu, který zvládne provést konstrukci v lineárním čase. Veškerá tato implementace by měla být formou knihovny, která by pak poskytovala funkce sufixového stromu.

Následujícím požadavkem byla implementace dalších algoritmů, kterými se řeší stejná problematika, jako pomocí sufixových stromů. Druhá implementovaná knihovna tedy obsahovala algoritmy, které jsme si popsali v předchozí kapitole, jako je například Knuth–Morris–Pratt algoritmus, nebo řešení problematiky pomocí dynamického programování. Funkce této knihovny jsou poté použity k porovnání těchto algoritmu se sufixovým stromem.

Abychom mohli uživateli přiblížit funkce sufixového stromu a srovnání této struktury s jinými postupy, bylo zapotřebí také vytvořit grafické rozhraní, které by demonstrovalo tyto funkce. Hlavními požadavky na toto uživatelské rozhraní nebylo cílit na nejpřívětivější design, ale především demonstrovat veškerou funkčnost sufixového stromu a provést funkční srovnání časové náročnosti s ostatními algoritmy. Také důležitým cílem bylo dát uživateli možnost využit knihovnu s touto strukturou.

Abychom provedli co nejpřesnější měření časové náročnosti, nestačilo se spoléhat na uživatelské rozhraní, ale bylo také zapotřebí vytvořit testy, které by toto měření provedly. Dalším požadavkem tedy je vytvořit sadu testů, které budou zkoumat časovou efektivnost jednotlivým algoritmů a z jejich výstupu bude možné udělat grafické zobrazení časové náročnosti.

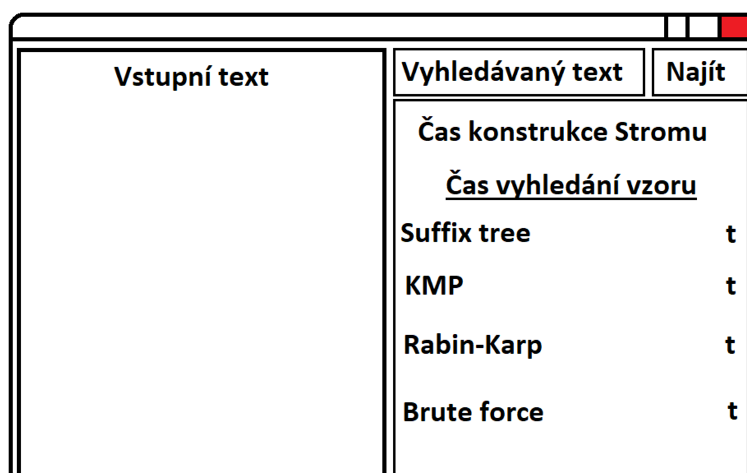
Všechny zmíněné požadavky tedy můžeme shrnout do následujících bodů:

- Vytvořit knihovnu implementující sufixový strom

- Implementovat algoritmus pro konstrukci sufixového stromu v lineárním čase
- Vytvořit metody, které nám umožní vyhledávat v sufixovém stromě a řešit problematiku s řetězci
- Implementovat jiné postupu a algoritmy pro řešení této problematiky
- Vytvořit grafické uživatelské rozhraní, které bude demonstrovat funkce sufixového stromu a srovnávat jeho výkonnost s ostatními algoritmy
- Vytvořit sadu testů, které budou srovnávat časovou náročnost jednotlivých postupů

4.2 Grafické rozhraní

Jelikož uživatelským rozhraním necílíme na širokou veřejnost a aplikace nemá za úkol být každodenně využívána, ale má spíše zastupovat funkci zobrazení výsledků výzkumu datové struktury, není kladen velký důraz na design grafického rozhraní. Hlavní důraz je kladen na jednoduché, avšak výkonné rozhraní, které bude dobře demonstrovat implementované algoritmy. Aplikace musí vyřešit danou problematiku nad řetězcem a správně změřit k tomu potřebný čas.



Obrázek 4.1: Návrh grafického rozhraní pro vyhledávání vzoru v textu

Grafické rozhraní se bude skládat z několika oken, z nichž každé bude uživateli umožňovat využít některou z funkcí sufixového stromu, jako je například vyhledání vzoru, nalezení nejdelšího společného podřetězce atd. Přepínání mezi těmito okny se bude provádět pomocí tlačítek, která se budou nacházet v horní části obrazovky. Například okno pro vyhledání vzoru má uživateli umožnit zadat vstupní text, ve kterém se bude vyhledávat a ze kterého budeme konstruovat sufixový strom. Následně bude možno zadat a vyhledat podřetězec, který bude ve vstupním textu zvýrazněn, jak to ukazuje obrázek 4.1. Rozhraní použije k vyhledání všechny implementované algoritmy a zobrazí výsledky dosažené jednotlivými algoritmy společně s časem potřebným pro splnění problému.

4.3 Testování algoritmů

Abychom získali co nejpřesnější čísla pro hodnocení časové náročnosti jednotlivých algoritmů, aplikace bude obsahovat sadu testů, které se o toto měření budou starat. Pro každý algoritmus budou

použita stejná vstupní data, která budou v různém rozsahu. Vstupní data budou mít podobu textu, jehož délku budeme zvětšovat a pozorovat tak, jak jednotlivé algoritmy reagují. Výstupem těchto testů bude čas, který byl potřebný pro vyřešení dané problematiky.

Kapitola 5

Implementace sufixového stromu

V této kapitole si popíšeme implementaci konstrukce sufixového stromu a také implementaci vyhledávacích operací v sufixovém stromě. Kromě toho se tato kapitola bude zabývat implementací Ukkonenova algoritmu a grafického rozhraní aplikace, která byla výstupem této práce.

5.1 Využité technologie

Jako platformu, na které jsem implementoval všechny knihovny s algoritmy, uživatelské rozhraní a testy jsem si vybral **.NET Core** od firmy Microsoft a veškerý kód byl napsán v jazyce **C#**. Tento jazyk a platforma umožňuje dobrou práci s objekty, což je pro vytváření stromu a velkého množství uzlů dobrá vlastnost.

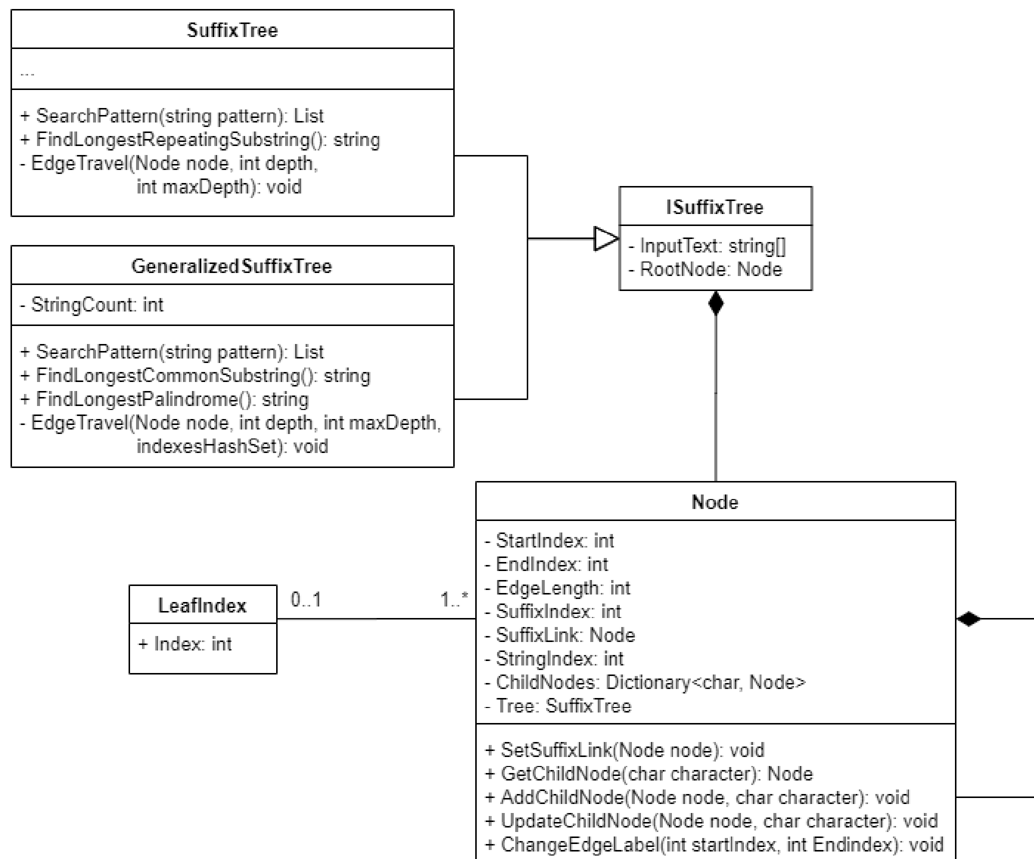
Testování a měření proběhlo pomocí xUnit testů, které jsou taktéž na této platformě. Grafické rozhraní, které má za úkol znázornit využití sufixových stromů a porovnat je s ostatními algoritmy, bylo vytvořeno pomocí systému **WPF** (Windows Presentation Foundation), taktéž na platformě .NET. Pro uchování veškerého kódu v cloudu a pro verzování jsem využil repozitář v **Microsoft Azure**.

5.2 Vytvoření datové struktury

Nejdříve bylo potřeba vytvořit datovou strukturu, která bude znázorňovat sufixový strom. To zahrnovalo vytvoření tříd pro reprezentaci stromu jako celku a také pro reprezentaci jednotlivých uzlů, ze kterých se tento strom skládá. Na obrázku 5.1 je vyobrazen diagram tříd pro třídy stromu a uzlů.

Třída reprezentující uzel sufixového stromu musí obsahovat veškerá data, která nám umožňují správnou reprezentaci vstupního řetězce. Jelikož ke každému uzlu vede nanejvýš jedna hrana, která vede od otcovského uzlu, a tato hrana je k uzlu vázána, tak nebylo třeba vytvářet samostatnou třídu pro reprezentaci této hrany, ale stačilo, aby třída uzlu nesla informace o této hraně. Díky tomu, že není potřeba ke každé instanci třídy uzlu vytvářet další instanci hrany, jsme schopni zredukovat množství vytvořených objektů a tím i potřebné množství v paměti a čas k vytvoření těchto objektů.

Uzel tedy musí nést informace o indexu, na kterém leží sufix, který tento uzel reprezentuje. Pokud se jedná o nelistový uzel, který tedy žádný sufix nereprezentuje, je tento index nastaven na záporné hodnoty. Jelikož však třída uzlu zastává také úlohu hran stromu, je nutné, aby jsme v této třídě měli informace o počátečním a koncovém indexu podřetězce, který tato hrana, vedoucí od otcovského uzlu k tomuto uzlu, reprezentuje. Toto jsou celočíselné hodnoty a samotný uzel nenese žádný podřetězec ze vstupního řetězce, ale místo toho se odkazuje na třídu stromu, která



Obrázek 5.1: Diagram tříd sufixového stromu.

si tento vstupní řetězec uchovává (blíže popsáno v kapitole 2.4.2). Uzel také musí nést informaci o tom, z jakého řetězce se skládá podřetězec, který tento uzel reprezentuje. Tato informace je využita v případě, kdy sestavujeme generalizovaný sufixový strom, který se skládá z více vstupních řetězců.

Velice důležitou otázkou implementace bylo správné uchovávání odkazů na synovské uzly. Je velice důležité řešit, jakým způsobem budeme uchovávat informace o potomcích, jelikož uzel může mít veliké množství synovských uzlů, kde toto množství může být až $|\Sigma|$, což je velikost abecedy, ze které je tvořen vstupní text S. Máme několik možností, jak odkazy na potomky uchovávat. Můžeme vytvořit pole o velikosti $|\Sigma|$ a indexovat jej pomocí znaků z abecedy $|\Sigma|$. Toto pole nám umožňuje přistoupit na daný index v konstantním čase, ale zabírá $\Theta(|\Sigma|)$ místa a to i v případě, kdy má uzel například jenom dva potomky [12]. Tato možnost je tedy časově velice výhodná, ale prostorově náročná.

Další možností je použít jednosměrně vázaný lineární seznam. Tato možnost řeší problém s prostorovou náročností, jelikož počet položek seznamu se rovná počtu potomků uzlu. Při náhodném přístupu je časová náročnost naopak v nejhorším případě $O(|\Sigma|)$, což může mít dopad na efektivnost sufixového stromu.

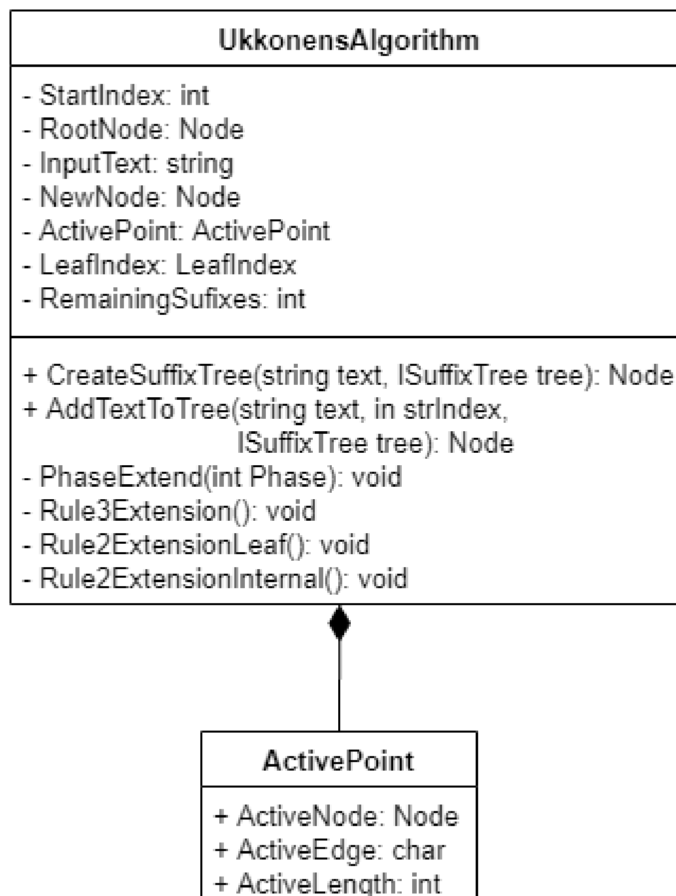
Třetí možností je využít hashovací tabulku, díky které můžeme vyřešit nedostatky obou předchozích struktur. V implementaci jsem využil kolekci Dictionary, která je implementací hashovací tabulky a je k dispozici v .NET Core. Do této kolekce můžeme vkládat objekty pod určitými jedinečnými klíči, což je pro náš sufixový strom ideální, jelikož jako objekt vkládáme synovský uzel a jako jeho klíč použijeme první znak, kterým je označena k němu vedoucí hrana. Tento znak je pro danou kolekci jedinečný, o čemž hovoří definice 2.1.1. Tato kolekce řeší problém s prostorovou

náročností, jelikož místo v paměti je závislé na počtu přidaných položek. Taktéž máme vyřešen problém s časovou náročností, protože přístup do této kolekce za použití klíče je konstantní a blíží se k $O(1)$ [1].

Abychom mohli při konstrukci stromu vytvářet univerzální metody jak pro obyčejný sufixový strom, tak pro ten generalizovaný, máme vytvořeno rozhraní (*interface*), které nám říká jaké atributy musí mít implementace tohoto rozhraní. Sufixový strom v sobě uchovává vstupní řetězec (v případě generalizovaného stromu kolekci řetězců), ze kterého se tento samotný strom skládá. Jednotlivé implementace rozhraní v sobě poté nesou metody pro vyhledávání ve stromě, jako například vyhledání nejdelšího opakujícího se podřetězce.

5.3 Implementace Ukkonenova algoritmu

Pro implementaci Ukkonenova algoritmu jsem vytvořil knihovnu tříd, která v sobě uchovává veškerá data a metody, které jsou potřebné ke správnému sestavení sufixového stromu. Metoda třídy sufixového stromu pak na vstupu příjme text, ze kterého chceme strom zkonstruovat, a posílá jej dalším metodám, které následně pracují na samotné konstrukci stromu. Jako výstup této metody je poté instance třídy `Node`, která reprezentuje kořenový uzel vytvořeného stromu. V diagramu tříd na obrázku 5.2 jsou opět vyobrazeny metody a vlastnosti třídy.



Obrázek 5.2: Diagram tříd ukazující statickou třídu, která implementuje Ukkonenův algoritmus.

Při konstrukci si uchováváme vstupní řetězec, z jehož částí následně strom tvoříme. Atribut `leafIndex` je pomocný objekt, který přidělujeme všem listovým uzlům. Tento objekt má jediný celočíselný atribut a slouží k tomu, abychom mohli v konstantním čase inkrementovat rozsah všem listovým hranám (třetí urychlovací trik konstrukce popsaný v kapitole 2.4.3). Dále si zaznamenáváme odkaz na uzel, který uchováváme v `newNode`, abychom pro tento nový uzel v následujícím rozšíření mohli určit, kam povede jeho sufixový odkaz (popsáno v kapitole 2.4.3). V `remainingSuffixCount` uchováváme počet sufixů, které je potřeba přidat do stromu. Pokud je při rozšiřování stromu o daný sufix uplatněno třetí rozšiřovací pravidlo, je sufix ve stromě obsažen pouze implicitně. To znamená že v této proměnné uchováváme počet sufixů, které jsou v aktuálním rozestavěném stromě obsaženy implicitně (o této tématice pojednává kapitola 2.2.1). Důležitým atributem je objekt `activePoint`, pomocí kterého si značíme aktuální pozici ve stromě. Tento aktivní bod nás v každém rozšíření nasměruje k místu, kde budeme rozšiřovat strom o nový znak [2, 10]. Objekt má tři atributy, a to `activeNode`, `activeEdge` a `activeLength`. Atribut `activeNode` nám říká, z kterého uzlu v aktuálním rozšíření začínáme strom procházet k požadovanému místu, kde se bude rozšiřovat. Pomocí `activeEdge` víme, kterou hranou se z `activeNode` k tomuto místu vydat a `activeLength` nám říká, jak daleko se požadované místo nachází. Tento aktivní bod v každém rozšíření aktualizujeme a posuneme ho na nové místo pomocí již zmíněného sufixového odkazu.

Algoritmus se při tvorbě skládá z m etap, kde m je délka vstupního řetězcem, a každá etapa se dále dělí na několik rozšíření (detailně popsáno v kapitole 2.4.1). Každá etapa konstrukce stromu probíhá dle diagramu ukázaném na obrázku 5.3.

Na začátku etapy vždy inkrementujeme `remainingSuffixCount`, jelikož chceme v dané etapě strom rozšířit o jeden nový znak a následně také rozšíříme popis listových hran, které se už ve stromě nachází (trik 3). Nyní už začneme s cestováním z aktivního bodu k požadovanému místu rozšiřování. Nejdříve proběhne kontrola, jestli z uzlu `activeNode` vychází hrana začínající znakem `activeEdge`. Pokud se takový hrana v daném uzlu nenachází, vytvoříme nový listový uzel a listovou hranu označíme požadovaným znakem. Jestliže hrana existuje, zkontrolujeme, zdali se nedá hrana přeskočit a uplatnit tak trik 1. Při uplatnění triku 1 však musíme aktivní bod posunout na nižší uzel a upravit jeho atributy, následně se opět vrátíme ke kontrole, jestli z tentokrát už nižšího uzlu vychází požadovaná hrana. Takto pokračujeme, dokud nedojdeme k požadovanému místu, kde chceme vložit nový znak. Zkontrolujeme, jestli se požadovaný znak na hraně již nenachází a pokud tomu tak je, uplatňuje se třetí rozšiřovací pravidlo a tím i trik číslo 2 a aktuální etapa končí. V opačném případě je provedeno rozšíření pomocí druhého rozšiřovacího pravidla a je vytvořen nový interní a listový uzel. Dále také dekrementujeme `remainingSuffixCount`, jelikož byl strom o požadovaný sufix rozšířen a aktualizuje se aktivní bod pomocí sufixového odkazu. Všimněme si, že pokud se uplatní trik č. 2, `remainingSuffixCount` se nedekrementuje a v následující etapě je tato proměnná vyšší, jelikož požadovaný sufix není ve stromě vyjádřen explicitně.

Kromě metody pro vytvoření obyčejného sufixového stromu třída disponuje také metodou pro rozšíření sufixového stromu o další řetězce. Tato metoda je využívána při tvorbě generalizovaných sufixových stromů. Metoda vezme již existující sufixový strom a rozšíří jej o daný řetězec, který dostane na vstupu.

5.4 Vyhledávání ve stromě

Při vyhledávání ve stromě je potřeba procházet buď celý strom (např. nejdelší společný podřetězec), nebo pouze některý z podstromů (vyhledávání podřetězce). Ve většině těchto operací se k procházení používá rekurzivního volání funkcí, avšak v některých operacích je použito i iterativní procházení, nebo kombinace obou.

Při vyhledávání podřetězce ve stromě nejdříve postupujeme iterativně, kdy hledáme cestu z kořenového uzlu, jejíž označení je shodné s textem vyhledávaného podřetězce. Pseudokód algoritmu č. 3 ukazuje, jak tento iterativní postup vypadá.

Algorithm 3: Vyhledání podřetězce v textu pomocí sufixového stromu

```
Node = Root.Child[Pattern[0]]
while Pattern.Length > 0 do
    if Node == null then
        //Vyhledávaný podřetězec se v textu nevyskytuje
        return null
    end
    if Node.Label != Pattern.Substring then
        //Vyhledávaný podřetězec se v textu nevyskytuje
        return null
    end
    //Z Pattern odebereme prvních Node.Length znaků
    Pattern = Pattern.Substring
    //Do Node uložíme nový uzel, jehož popis hrany má stejný počáteční znak jako
     Pattern
    Node = Node.Child[Pattern[0]]
end
//Pattern se v textu vyskytuje. Indexy s výskytem vzoru se nachází v podstromu
 s vrcholem Node
return Node
```

Metoda tedy s vyhledáváním začne vždy v kořenovém uzlu , ze kterého prvně musíme najít hranu, která se shoduje s vyhledávaným vzorem. Jakmile ji najdeme přesuneme se na uzel, ke kterému tato hrana vede, porovnáme podřetězec hrany se vzorem a opět hledáme další hranu z aktuálního uzlu. Tento postup opakujeme, dokud není porovnán celý vzor, nebo dokud se v některém místě nepodařilo najít další cestu, která by se se vzorem shodovala a v takovém případě se tento podřetězec v textu nevyskytuje. Tato metoda buď zahlásí, že se vyhledávaný podřetězec nepodařilo najít, nebo vrátí uzel, v jehož podstromě se nachází indexy výskytu podřetězce.

Pokud nám tedy metoda navrátila uzel ze stromu, máme ověřeno, že se vyhledávaný podřetězec v textu nachází, ale nemáme ještě získány indexy výskytů tohoto podřetězce. Musíme tedy projít podstrom, jehož vrcholem je navrácený uzel. V této fázi už používáme rekurzivní procházení stromu a projdeme celý tento podstrom, jelikož vyhledáváme všechny listové uzly. Pseudokód algoritmu č. 4 znázorňuje tento průchod podstromu.

Metoda `GetIndexes`, kterou voláme rekurzivně, projde celý podstrom a jakmile dojde k některému listu v podstromu, přidá index sufixu do seznamu, který dostala v argumentu při volání.

Algorithm 4: Nalezení indexů podřetězce v textu pomocí rekurzivního procházení sufixového stromu

```
GetIndexes(Node, IndexList) begin
    if Node.ChildNodes.Count == 0 then
        //Dorazili jsme k listovému uzlu
        IndexList.Add(Node.SuffixIndex)
        return
    end
    foreach childNode in Node.Childnodes do
        GetIndexes(childNode, IndexList)
    end
    return
end
```

Tímto způsobem se nám vrátí všechny indexy, které se nachází v listových uzlech v tomto podstromu.

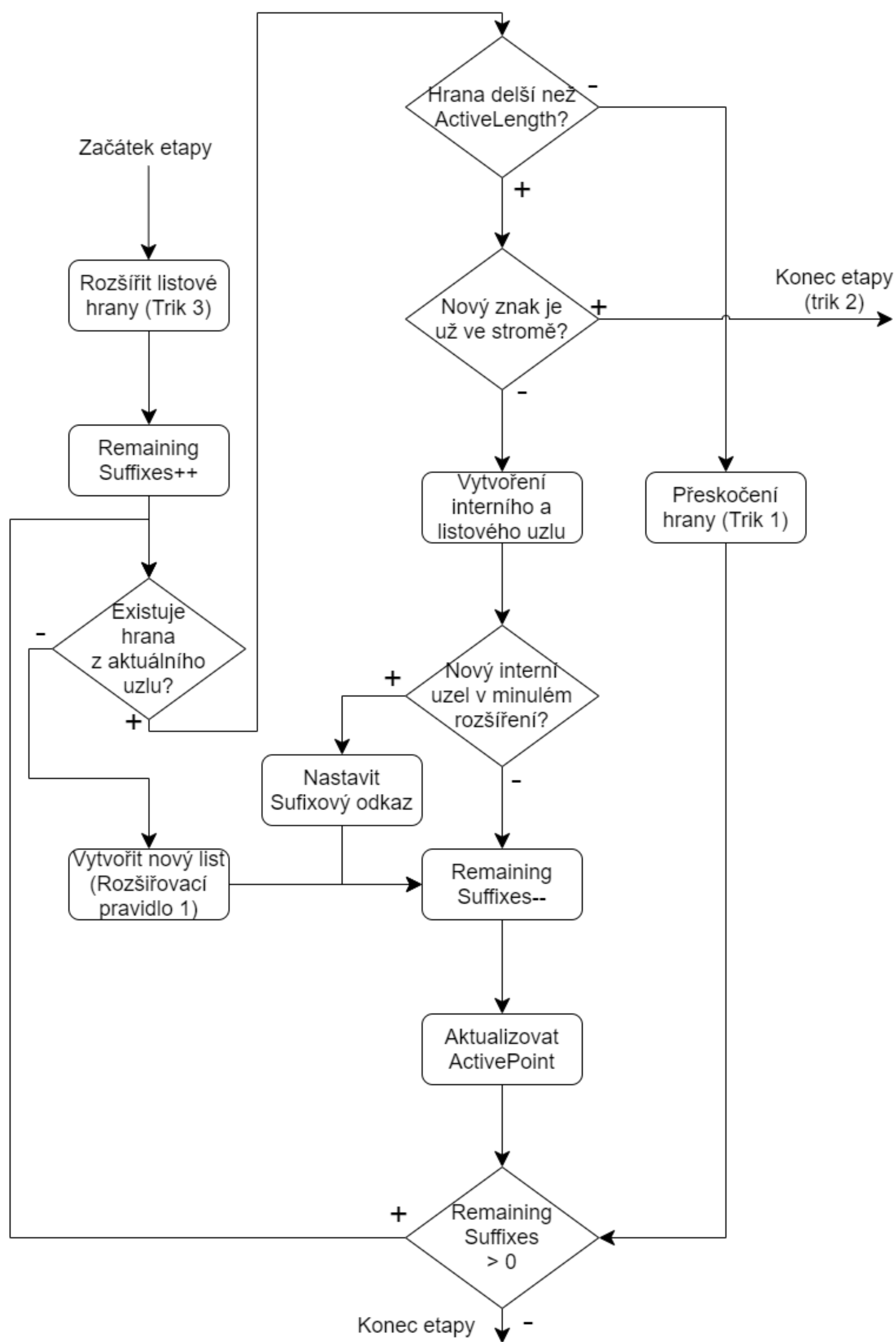
Stejně tak při hledání nejdelšího opakujícího se podřetězce procházíme strom rekurzivně. Funkce opět volá sama sebe pro každého potomka aktuálního uzlu a hledá nejhlubší interní uzel.

5.5 Grafické rozhraní

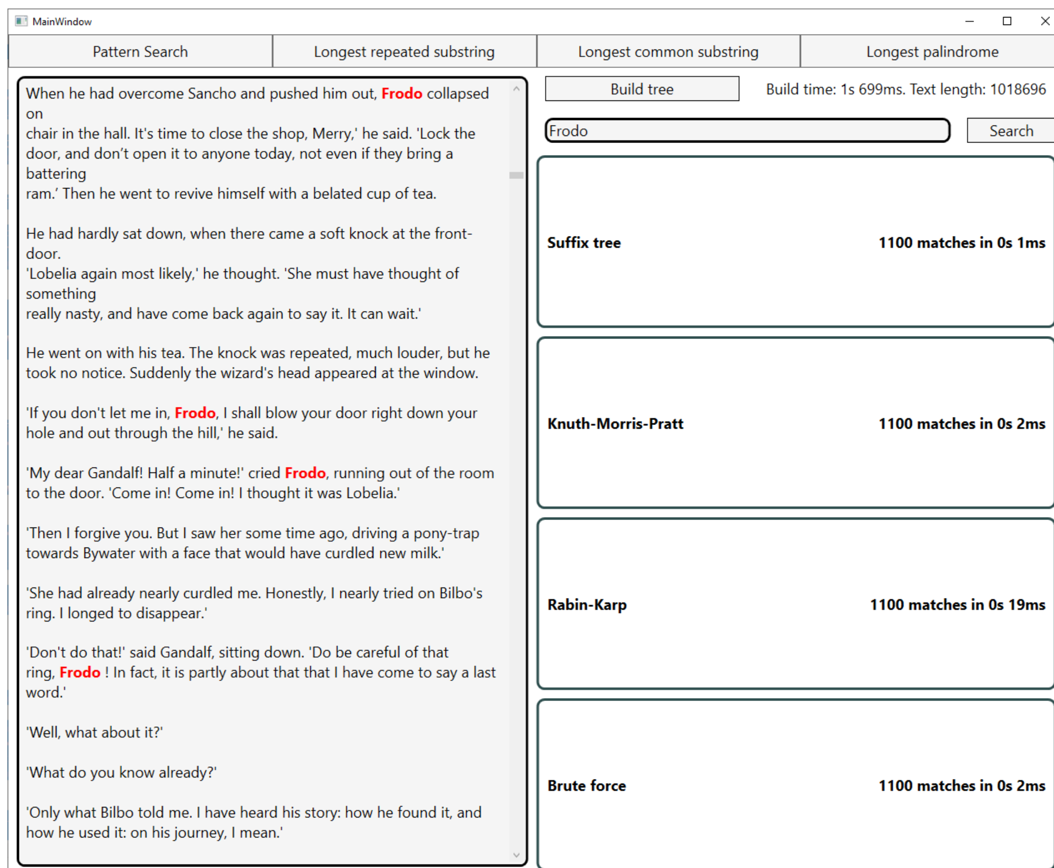
Grafické rozhraní bylo implementování pomocí systému WPF na platformě .NET Core. Jedná se o desktop aplikaci, která byla vytvořena za pomoci ovládacích prvků poskytnutých touto platformou. K vytvoření tohoto rozhraní nebylo potřeba použít žádné externí knihovny, nebo ovládací prvky třetích stran.

Pro každou z implementovaných funkcí sufixového stromu je vytvořené samostatné okno. Jednotlivá okna pro tyto funkce byly implementovány jako *UserControl* a mezi těmito ovládacími prvky se přepíná pomocí horních tlačítek, které slouží jako menu. Aplikace si ukládá informace z jednotlivých oken, takže při přepínání mezi těmito okny nedojde k přepsání dat, která jsme do okna už jednou zadali. Přidávání jednotlivých komponent okna a jejich pozicování se provádí v jazyce *XAML*, ve kterém můžeme jednotlivé komponenty rozmisťovat po okně například pomocí mřížek. Veškeré ovládání těchto komponent a propojení s knihovou implementovaných algoritmů je poté implementováno v kódu na pozadí (v angličtině *Code–Behind*).

Na obrázku 5.4 je zobrazeno okno pro vyhledání vzoru v textu. Do okna pro text s vyhledáváním může uživatel vložit libovolně dlouhý text a následně sestavit sufixový strom pro tento zadaný text. Aplikace na obrazovce zobrazí informace o tom, kolik znaků tento vstupní řetězec má a za jaký čas proběhla konstrukce stromu. Následně uživatel může začít vyhledávat v textu řetězec, který zadá do druhého pole. Jakmile je zahájeno vyhledávání, aplikace využije všechny implementované algoritmy pro vyhledávání a u každého algoritmu změří čas, který k tomu tyto algoritmu potřebovaly. Platforma .NET nám k měření času poskytuje třídu *Stopwatch*, která umožňuje přímo v kódu určit začátek a konec měření času. Tímto způsobem jsme schopni zachytit čas potřebný k provedení námi požadovaného kódu. Jakmile veškeré vyhledávání skončí, je na obrazovce zobrazen počet nalezených výskytů a čas, který byl k nalezení u těchto algoritmů potřeba. Dále je také ve vstupním textu zvýrazněn vyhledávaný řetězec, a to právě za pomocí indexů, které nám nám navrátila metoda sufixového stromu.



Obrázek 5.3: Vývojový diagram ukazující průběh rozšíření etapy v Ukkonenově algoritmu.



Obrázek 5.4: Okno pro vyhledávání řetězce v textu ve výsledné aplikaci.

Kapitola 6

Měření

V této kapitole se podíváme na výsledky měření, které proběhly pomocí implementovaných testů. Testování časové náročnosti proběhlo u všech implementovaných funkcí popsaných v kapitole 3. Následně proběhlo porovnání této časové náročnosti i s ostatními implementovanými algoritmy, které jsme si popsali. Kromě testování samotného vyhledávání byl otestován samotný Ukkonenův algoritmus pro konstrukci stromu. Tento algoritmus je otestován jak pro časovou, tak i prostorovou náročnost, kde jsme testovali, jak tato náročnost reaguje na velikost vstupního textu, ze kterého strom konstruujeme. Veškeré výsledky těchto měření jsou zaneseny do grafů.

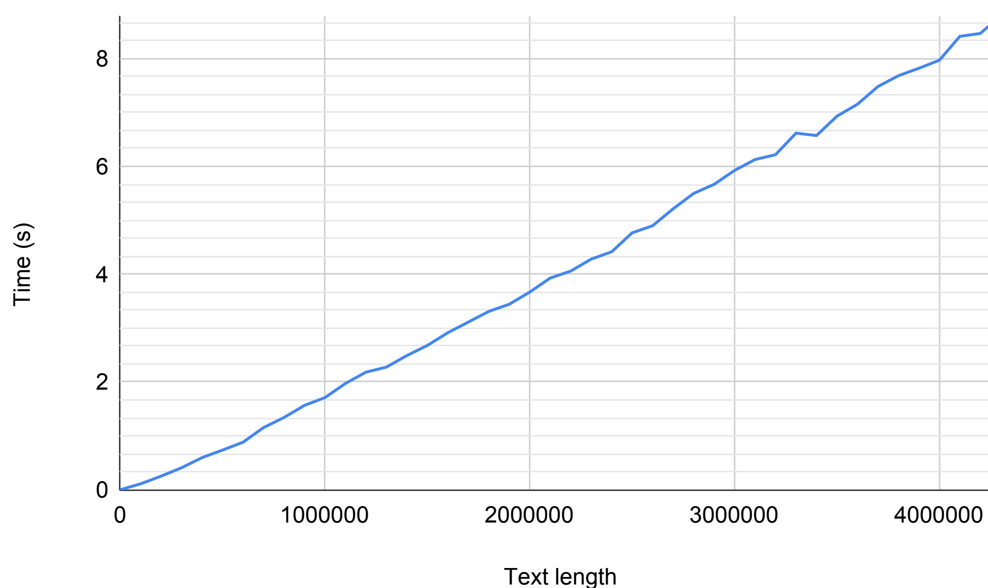
6.1 Vstupní data

Jako vstupní data používáme text, který je dlouhý 4 300 000 znaků, a pomocí cyklu spouštíme algoritmy s textem od nulové délky až po jeho maximální délku. V jednotlivých iteracích cyklu pak měříme čas, za který algoritmus splnil úkol pro text dané délky. Aby byla získaná data co nejpřesnější, každé měření je se stejnými daty provedeno pětkrát a výsledná data jsou pak jejich průměrem.

Výstupem testů je pak sada hodnot, které reprezentují časovou nebo paměťovou náročnost daného algoritmu. Tyto hodnoty byly následně zadány do tabulkového procesoru a výsledky jednotlivých měření byly zprůměrovány. Výsledné hodnoty byly následně zaneseny do grafů.

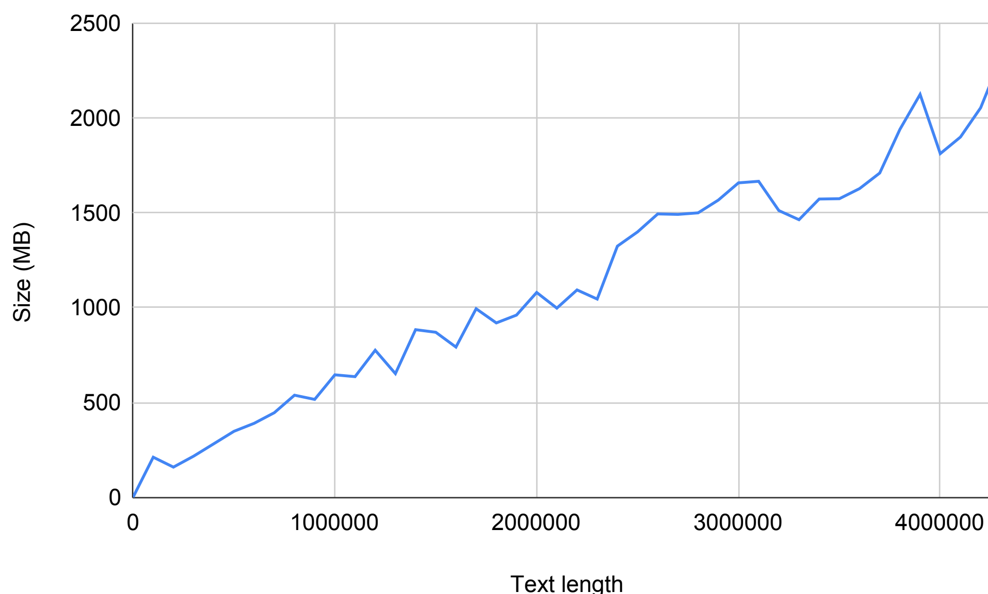
6.2 Ukkonenův algoritmus

Prvně se podíváme na výsledky měření Ukkonenova algoritmu. Testování probíhalo pro vstupní text v rozsahu 0 až 4 300 000 znaků. Obrázek 6.1 zobrazuje časovou náročnost konstrukce stromu pomocí Ukkonenova algoritmu.



Obrázek 6.1: Čas potřebný pro sestavení sufixového stromu pomocí Ukkonenova algoritmu v závislosti na počtu znaků.

Jak jsme si řekli v kapitole 2.4.3, časová náročnost Ukkonenova algoritmu je lineární a odvíjí se od délky vstupního textu. Z grafu, který ukazuje čas potřebný pro sestavení pomocí Ukkonenova algoritmu, můžeme sledovat, že má lineární vývoj v závislosti na délce vstupního textu, takže odpovídá časové náročnosti $O(m)$, která je uváděná v literatuře [12].

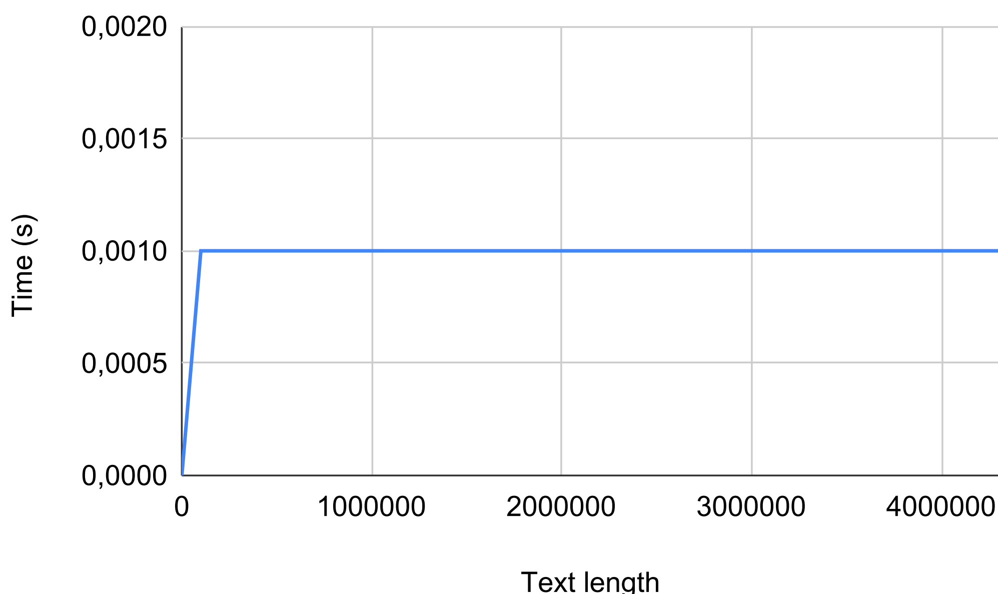


Obrázek 6.2: Paměťová náročnost sufixového stromu sestaveného pomocí Ukkonenova algoritmu v závislosti na počtu znaků.

Graf 6.2 znázorňuje, jak roste paměťová náročnost aplikace v závislosti na velikosti vstupního textu. Velikost obsazené paměti je v grafu vyjádřena v *MB* a stejně jako v grafu časové náročnosti má křivka podobu lineární funkce. Díky tomu, že jednotlivé hrany nenesou podřetězce ze vstupního textu, ale pouze indexy začátku a konce podřetězce, jak jsme si popsali v sekci 2.4.2, je tato prostorová náročnost $O(m)$, kde m je délka vstupního řetězce [6].

6.3 Vyhledávání podřetězce

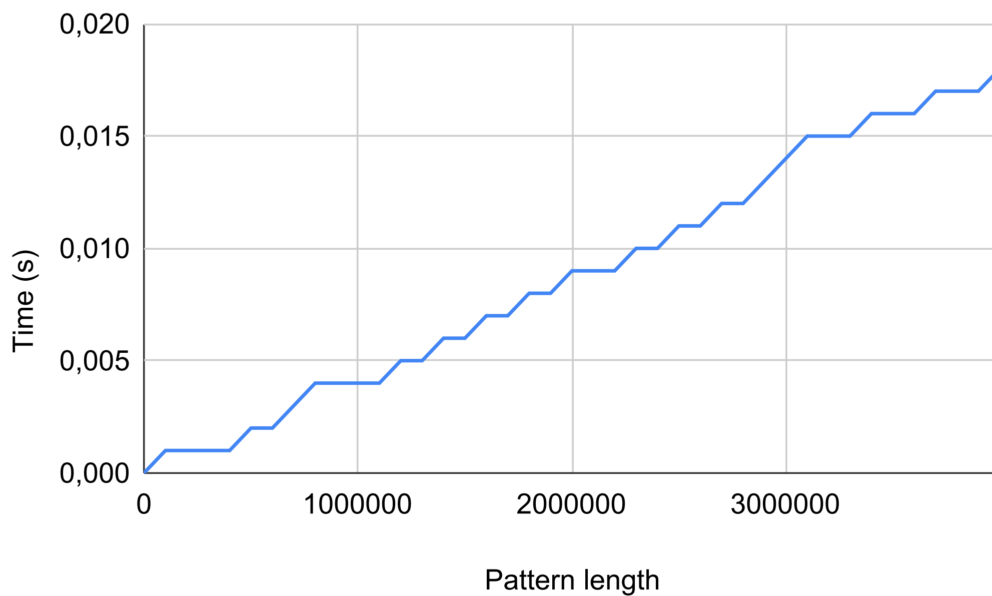
V této sekci se zaměříme na výsledky při vyhledávání vzoru v textu. Nejdříve se podíváme na samotné výsledky sufixového stromu a poté jej srovnáme s ostatními algoritmy. Nejdříve si však připomeňme, jak sufixový strom pracuje při vyhledávání řetězce. V kapitole 3.1.1 jsme si řekli, že časová náročnost sufixového stromu je poměrně odlišná od náročnosti ostatních popisovaných algoritmů.



Obrázek 6.3: Čas potřebný k vyhledání vzoru v textu za pomoci sufixového stromu v závislosti na délce textu s fixní délkou vyhledávaného vzoru.

Podívejme se na obrázek 6.3, kde je zobrazena časová náročnost pro vyhledávání vzoru, který má stále stejnou velikost (100 000 znaků), v textu, jehož délka se zvětšuje. Uvedli jsme, že časová náročnost při vyhledávání vzoru není závislá na délce textu, ve kterém vyhledáváme, jelikož procházíme pouze tu část stromu, která je označena vyhledávaným vzorem. Jelikož tedy neprocházíme všechny větve stromu, to znamená že ani neprocházíme celý text, je časová náročnost vázána pouze na délku vyhledávaného vzoru, přesně jak jde vidět v grafu.

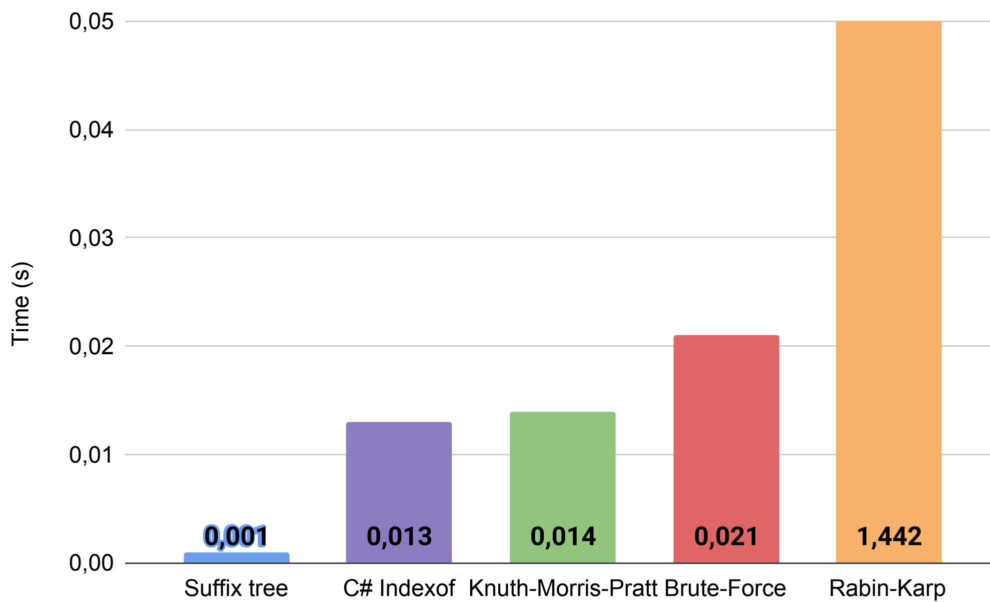
Graf 6.4 nám ukazuje opačný případ, kdy máme délku textu stejnou a měníme délku vyhledávaného vzoru. Zde čas stoupá v závislosti na tom, jak zvětšujeme délku vzoru. Potvrzuje se nám tedy udávaná časová náročnost $O(n)$, kde n je délka vyhledávaného vzoru. Další výhodou také je, že při tomto testování nebylo potřeba sufixový strom po každém zvětšení vzoru znovu sestavit, ale byl sestaven pouze jednou pro tyto testy, jelikož se délka vstupního textu neměnila.



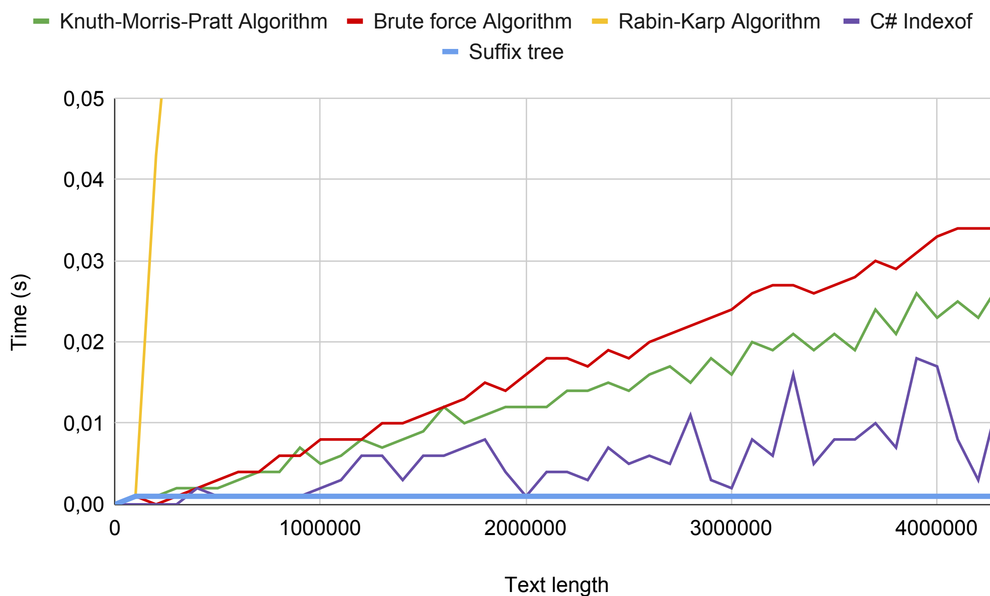
Obrázek 6.4: Čas potřebný k vyhledání vzoru v textu za pomoci sufixového stromu v závislosti na délce vyhledávaného vzoru s fixní délkou textu.

Nyní se už dostaneme k samotnému porovnání sufixového stromu s ostatními algoritmy. Všechny algoritmy byly spuštěny nad stejným textem a vyhledávaly stejný vzor. V prvním porovnání na grafu 6.5 jsou uvedeny výsledky tentokrát pro konkrétní vyhledávání vzoru o délce 100 000 v textu délky 4 300 000 (opět průměr z pěti měření). Kromě sufixového stromu bylo vyhledávání provedeno také pomocí algoritmů zmíněných v kapitole 3.1 a navíc také pomocí metody *Indexof()*, která je zabudovaná v platformě .NET.

Naměřené výsledky v případě sufixového stromu neobsahují čas konstrukce, což znamená, že hodnoty časové náročnosti jsou u sufixového stromu čistě pro vyhledání vzoru v už sestaveném stromě. Z výsledků v grafu si můžeme všimnout, že sufixový strom byl ve vyhledávání oproti ostatním algoritmům více než desetkrát efektivnější. Výkonnost sufixového stromu se skrývá právě v jeho časové náročnosti, která se jako u jediného z těchto algoritmů odvíjí od délky vyhledávaného vzoru, namísto od délky textu, jak je tomu u ostatních algoritmů. Pokud bychom však k těmto hodnotám přičetli i čas konstrukce stromu, sufixový strom by byl pro toto vyhledávání neefektivní. Hlavní výhody sufixového stromu tedy najdeme při vyhledávání v textu, který se nemění a vyhledáváme v něm opakovaně různé vzory.



Obrázek 6.5: Čas běhu jednotlivých algoritmů při vyhledávání vzoru o délce 100 000 znaků v textu dlouhém 4 300 000 znaků. Hodnoty sufixového stromu jsou pro samotné vyhledávání a není přičten čas konstrukce stromu.



Obrázek 6.6: Porovnání časové náročnosti algoritmů při vyhledávání vzoru o délce 100 000 znaků v závislosti na délce textu. Hodnoty sufixového stromu jsou pro samotné vyhledávání a není přičten čas konstrukce stromu.

Na grafu 6.6 je zobrazeno, jak se časová náročnost jednotlivých algoritmů vyvíjí v závislosti na rostoucí délce vstupního textu. Zde se opět projevuje odlišná časová náročnost sufixového stromu,

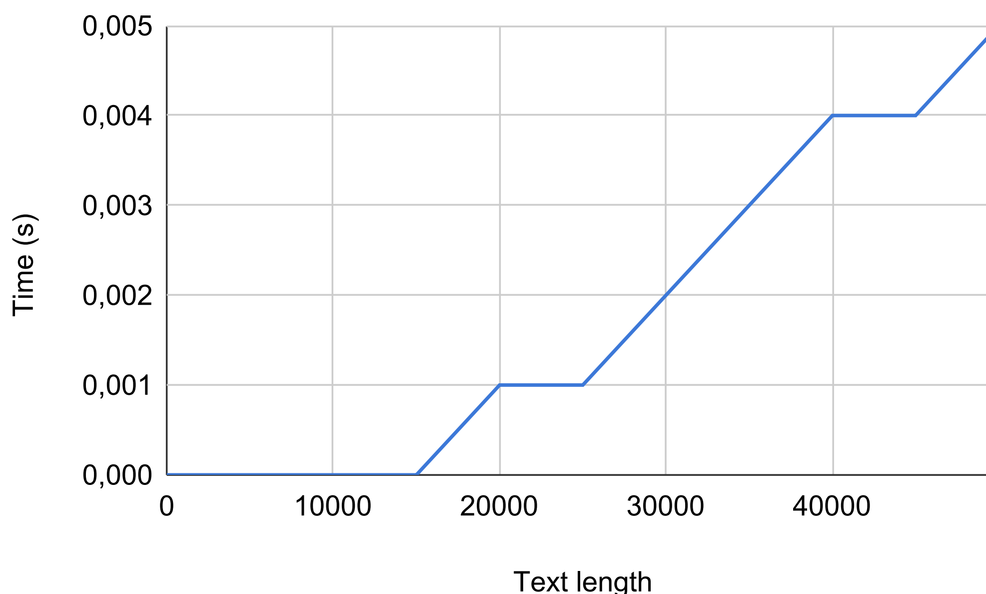
u kterého nezáleží, jak dlouhý je vstupní text. Délka textu se pohybovala od 0 znaků až po 4 300 000 znaků. Vyhledávaný vzor měl délku 100 000 a délka byla pro testováni nezměněna.

Jako nejrychlejší se v tomto testu ukázal být sufixový strom, jehož čas zůstával konstantní kvůli fixní délce vzoru. Nesmíme však zapomínat na to, že pokud dochází k častým změnám v textu, tak sufixový strom ztrácí svojí výhodu. Jako druhý nejrychlejší postup se ukázala být metoda *Indexof* platformy .NET. Blízko u sebe se umístili Knuth–Morris–Pratt a naivní algoritmus, jejichž čas potřebný k vyhledání stoupá lineárně.Ackoliv má Rabin-Karp algoritmus lineární časovou náročnost, ukázal se ve vyhledávání vzorů jako nejméně efektivní. Vyhledávání vzorů v textu však není jeho hlavní účel, jelikož jeho využití najdeme spíše při odhalování plagiátorství.

Grafy časové náročnosti jednotlivých algoritmů najdeme v příloze A.

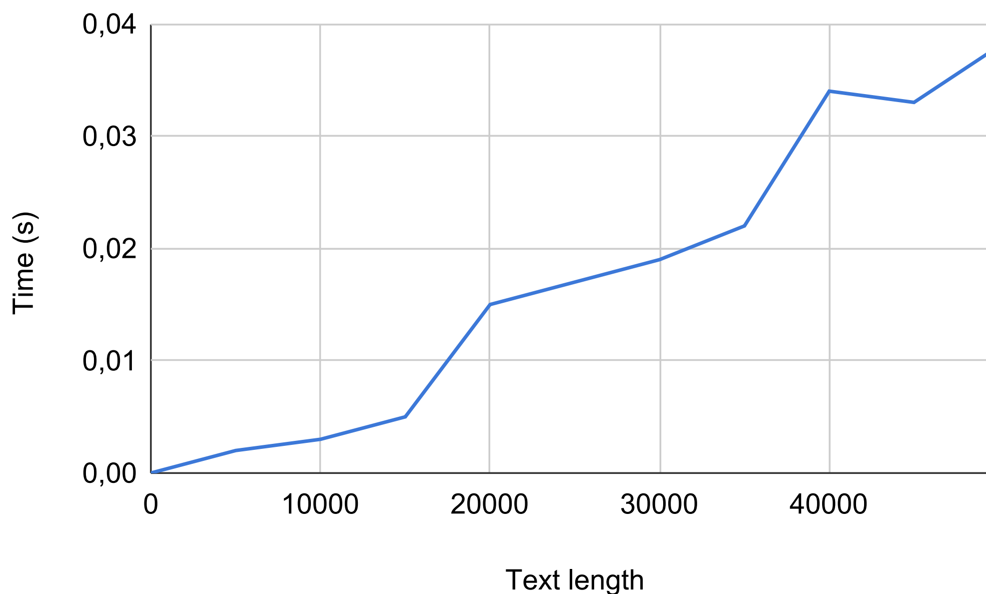
6.4 Nejdelší opakující se podřetězec

Nyní se podíváme na pokročilé funkce sufixového stromu a budeme jej porovnávat s dynamickým programováním. V kapitole 3.2 jsme si popsali, co je to nejdelší opakující se podřetězec a kde se využívá. Také jsme si řekli něco o tom, jak tento podřetězec hledáme pomocí sufixových stromů a dynamického programování a řekli jsme si něco o náročnostech těchto postupů. U tohoto měření byla zkrácena maximální délka vstupního textu na 50 000 znaků, jelikož řešení pomocí dynamického programování má vysokou paměťovou náročnost a při větším vstupním textu docházelo k přeplnění systémové paměti.



Obrázek 6.7: Čas potřebný k vyhledání nejdelšího opakujícího se podřetězce v textu za pomoci sufixového stromu v závislosti na délce vstupního textu. Uvedené hodnoty časové náročnosti znázorňují samotné vyhledávání.

Graf 6.7 nám ukazuje potřebný čas k samotnému vyhledání nejdelšího opakujícího se podřetězce pomocí sufixového stromu. Do těchto hodnot časové náročnosti není připočítán čas potřebný pro konstrukci stromu. V případě tohoto vyhledávání už neprocházíme pouze část stromu, ale procházíme celý strom, jelikož musíme najít nejhlubší interní uzel. Časová náročnost této operace tedy



Obrázek 6.8: Čas potřebný k vyhledání nejdelšího opakujícího se podřetězce v textu za pomoci sufixového stromu v závislosti na délce vstupního textu. Do hodnot časové náročnosti je započten i čas potřebný pro sestavení stromu.

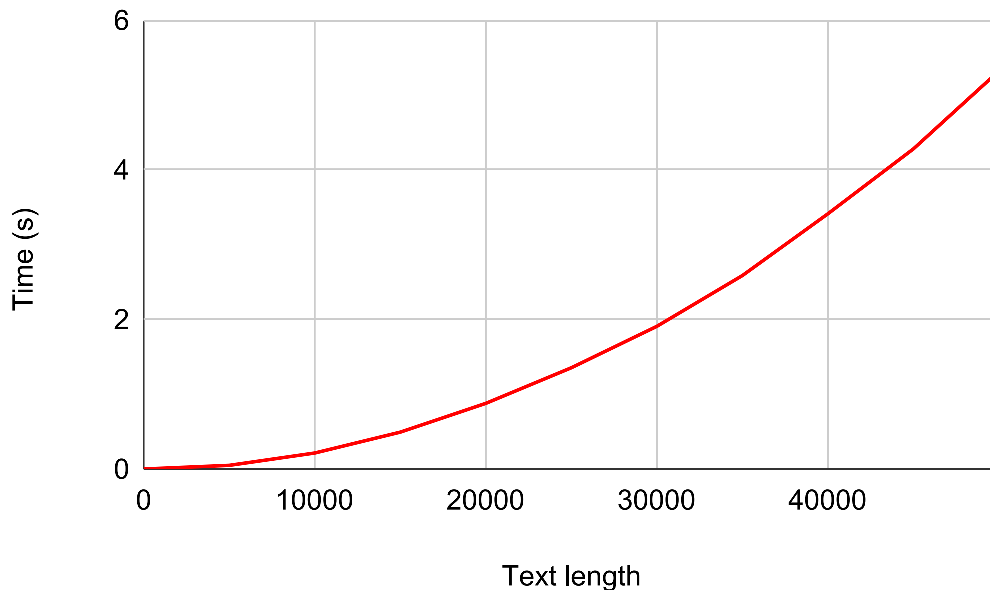
reaguje na délku vstupního textu jinak, než tomu bylo u vyhledávání vzoru v grafu 6.3, kde se časová náročnost neodvíjela od délky textu. Podle křivky v grafu vyjadřující tuto časovou náročnost můžeme určit, že se jedná o lineární nárůst, což nám také určuje časová náročnost $O(m)$ [23].

Protože však při vyhledávání nejdelšího opakujícího se podřetězce nevyhledáváme opakovaně ve stejném stromě, jelikož bychom dostávali stejné výsledky, je v tomto případě strom sestavován opakovaně pro každé vyhledávání. Graf 6.8 ukazuje, jak vypadá čas potřebný pro vyhledání nejdelšího opakujícího se podřetězce včetně sestavení sufixového stromu pro vstupní text.

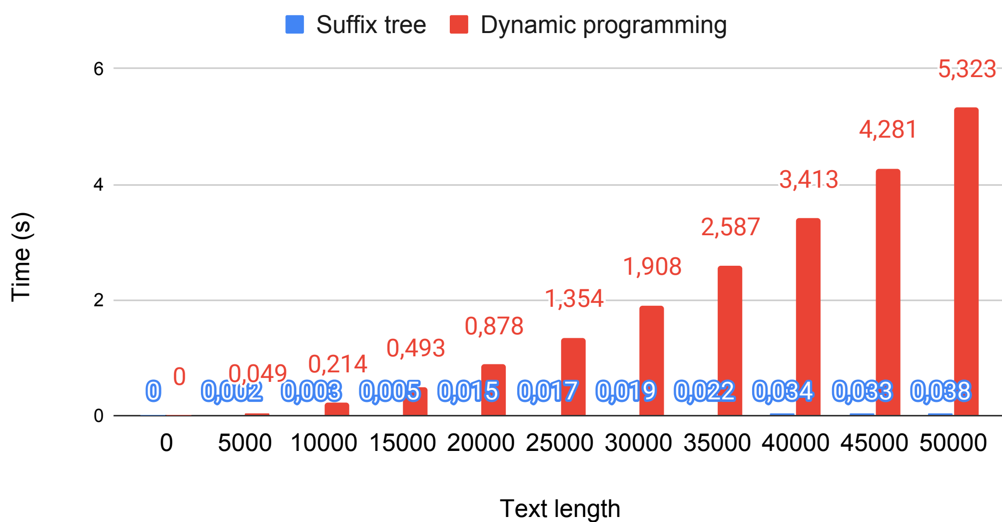
Na obrázku 6.9 je vyobrazen čas potřebný pro řešení stejné problematiky, tentokrát však za pomoci dynamického programování. Už na první pohled je viditelné, že časová náročnost není u tohoto postupu stejného charakteru, jako tomu bylo u sufixového stromu. Křivka, která svým tvarem připomíná spíše parabolu, nám říká, že časová náročnost tohoto postupu není lineární, ale její nárůst je kvadratický, což nám potvrzuje časovou náročnost $O(m^2)$, kterou jsme pro tento algoritmus uváděli [14].

Nyní se už v grafu 6.10 podíváme na přímé srovnání těchto dvou postupů. Dynamické programování je se svojí kvadratickou náročností oproti sufixovému stromu velice neefektivní a to jak časovou, tak i prostorovou náročností. Sufixový strom je pro řešení této problematiky mnohem efektivnější a to i přesto, že provádíme novou konstrukci stromu pro každé vyhledávání.

Pro vyřešení problematiky s nejdelším společným podřetězcem a nejdelším palindromickým podřetězcem je časový charakter totožný s řešením problematiky nejdelšího opakujícího se podřetězce u sufixového stromu i dynamického programování. Dynamické programování tuto problematiku řeší v kvadratickém čase $O(m^2)$ a sufixový strom v lineárním čase $O(m)$. Veškeré grafy zobrazující čas řešení této problematiky jsou obsaženy v příloze A.



Obrázek 6.9: Čas potřebný k vyhledání nejdelšího opakujícího se podřetězce v textu za pomoci dynamického programování v závislosti na délce vstupního textu.



Obrázek 6.10: Porovnání časové náročnosti sufixového stromu (hodnoty časové náročnosti obsahují také čas potřebný ke konstrukci stromu) a dynamického programování v závislosti na délce vstupního textu.

Kapitola 7

Závěr

Hlavním účelem této práce bylo seznámit se se strukturou sufixových stromů, s algoritmy pro jejich konstrukci a s problematikou, kterou nám sufixové stromy umožňují řešit. Dalším úkolem bylo uvést v jakém oboru se tato problematika řeší a jaké další postupy k řešení využíváme. Výstupem práce je ucelená aplikace s grafickým rozhraním, která umožňuje uživateli pro ním zvolená data řešit problematiku spojenou s řetězci pomocí sufixových stromů, jako je například hledání vzoru v textu, nebo vyhledávání palindromů. Pro realizaci této aplikace jsem implementoval Ukkonenův algoritmus na konstrukci stromu, jelikož nám oproti naivnímu algoritmu umožňuje tuto konstrukci provést s lineární časovou i paměťovou náročností.

Při porovnávání potřebného času pro vyhledávání vzoru v textu byl sufixový strom v porovnání například s Knuth–Morris–Pratt algoritmem mnohonásobně rychlejší pro data o velikém rozsahu, kde tento vzor našel více než dvacetkrát rychleji. Důležité ovšem je, že čas potřebný k sestavení stromu, ačkoliv je lineární, je stále vysoký, a proto jsou sufixové stromy ideální pro použití v textu, ve kterém nedochází často ke změnám. V takovém případě sufixový strom překonává všechny ostatní měřené algoritmy. Pro řešení pokročilejší problematiky, jako je například vyhledávání nejdelšího opakujícího se podřetězce, je sufixový strom v porovnání s dynamickým programováním mnohem efektivnější a to jak z pohledu časové, tak i paměťové náročnosti. Jelikož se však tato práce zaměřuje pouze na přesné vyhledávání v textu, nevyužijeme plně výhodu dynamického programování, které má silnou stránku v přibližném porovnávání řetězců.

Při dalším studiu v této oblasti bych se chtěl zaměřit na hlavní nevýhodu sufixového stromu, což je právě náročné předzpracování textu. Tento problém může řešit využití jiných algoritmů, které konstruují sufixový strom paralelně a jejich uváděná časová náročnost je $O(log^4 m)$ pro text délky m [13].

Literatura

[1] *Dictionary Class (System.Collections.Generic) – Microsoft Docs*. Microsoft Corporation. Dostupné z: https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.dictionary-2?view=netcore-3.1.

[2] ADJEROH, D., BELL, T. a MUKHERJEE, A. *The Burrows-Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching*. Springer US, 2008. Springer series in statistics. ISBN 9780387789095.

[3] CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L. a STEIN, C. *Introduction To Algorithms*. MIT Press, 2001. Introduction to Algorithms. ISBN 9780262032933.

[4] CROCHEMORE, M., HANCART, C. a LECROQ, T. *Algorithms on Strings*. Cambridge University Press, 2007. ISBN 9780521848992.

[5] CROCHEMORE, M., ILIOPOULOS, C. S., LANGIU, A. a MIGNOSI, F. The longest common substring problem. *Mathematical Structures in Computer Science*. Cambridge University Press. 2017, roč. 27, č. 2, s. 277–295.

[6] FARACH COLTON, M. Optimal suffix tree construction with large alphabets. *Proceedings 38th Annual Symposium on Foundations of Computer Science*. 1997, s. 137–143. ISSN 0272-5428.

[7] GEPPINO, P. *Dynamic Programming*. Department of Information Engineering, School of Engineering – University of Padua, leden 2015. Dostupné z: http://www.dei.unipd.it/~geppo/DA2/DOCS/dynprog.pdf.

[8] GIEGERICH, R. a KURTZ, S. From Ukkonen to McCreight and Weiner: A Unifying View of Linear-Time Suffix Tree Construction. *Algorithmica*. 1997, roč. 19, č. 3, s. 331–353. Dostupné z: https://doi.org/10.1007/PL00009177. ISSN 1432-0541.

[9] GOLIN, M. *Longest Common Subsequences and Substrings*. Department of Computer Science and Engineering – The Hong Kong University of Science and Technology, listopad 2014. Dostupné z: http://www.cs.ust.hk/mjg_lib/Classes/COMP3711H_Fall14/lectures/LCS_Slides.pdf.

[10] GROSSI, R., SILVESTRI, F. a SEBASTIANI, F. *String Processing and Information Retrieval: 18th International Symposium, SPIRE 2011, Pisa, Italy, October 17-21, 2011, Proceedings*. Springer Berlin Heidelberg, leden 2011. Lecture Notes in Computer Science, sv. 7024. ISBN 9783642245831.

[11] GROSSI, R. a ITALIANO, G. F. Suffix Trees and their Applications in String Algorithms. In:. InProceedings of the 1st South American Workshop on String Processing, 1993.

[12] GUSFIELD, D. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997. ISBN 9780521585194.

[13] HARIHARAN, R. Optimal Parallel Suffix Tree Construction. *Journal of Computer and System Sciences*. 1997, roč. 55, č. 1, s. 44 – 69. Dostupné z: http://www.sciencedirect.com/science/article/pii/S0022000097914963. ISSN 0022-0000.

[14] INENAGA, S. a BANNAI, H. Finding Characteristic Substrings from Compressed Texts. *International Journal of Foundations of Computer Science*. 2009, roč. 23, č. 02, s. 261–280.

[15] KARP, R. M. a RABIN, M. O. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*. 1987, roč. 31, č. 2, s. 249–260.

[16] KNUTH, D. E., MORRIS, J. J. H. a PRATT, V. R. Fast Pattern Matching in Strings. *SIAM Journal on Computing*. 1977, roč. 6, č. 2, s. 323–350.

[17] MAREŠ, M. a VALLA, T. *Průvodce labyrintem algoritmů*. CZ.NIC, z.s.p.o., 2017. ISBN 9788088168195.

[18] MCCREIGHT, E. M. A Space-Economical Suffix Tree Construction Algorithm. *J. ACM*. New York, NY, USA: Association for Computing Machinery. Duben 1976, roč. 23, č. 2, s. 262–272. Dostupné z: https://doi.org/10.1145/321941.321946. ISSN 0004-5411.

[19] OBERHUBER, T. *Pattern Matching*. Mathematical Modelling Group Department of Mathematics Faculty of Nuclear Sciences and Physical Engineering, Czech Technical University in Prague. Dostupné z: http://geraldine.fjfi.cvut.cz/~oberhuber/data/vyuka/ubio/08-pattern-matching.pdf.

[20] SUNG, W. *Algorithms in Bioinformatics: A Practical Introduction*. CRC Press, 2009. Chapman & Hall/CRC Mathematical and Computational Biology. ISBN 9781420070347.

[21] UKKONEN, E. On-line construction of suffix trees. *Algorithmica*. 1995, roč. 14, č. 3, s. 249–260. Dostupné z: https://doi.org/10.1007/BF01206331. ISSN 1432-0541.

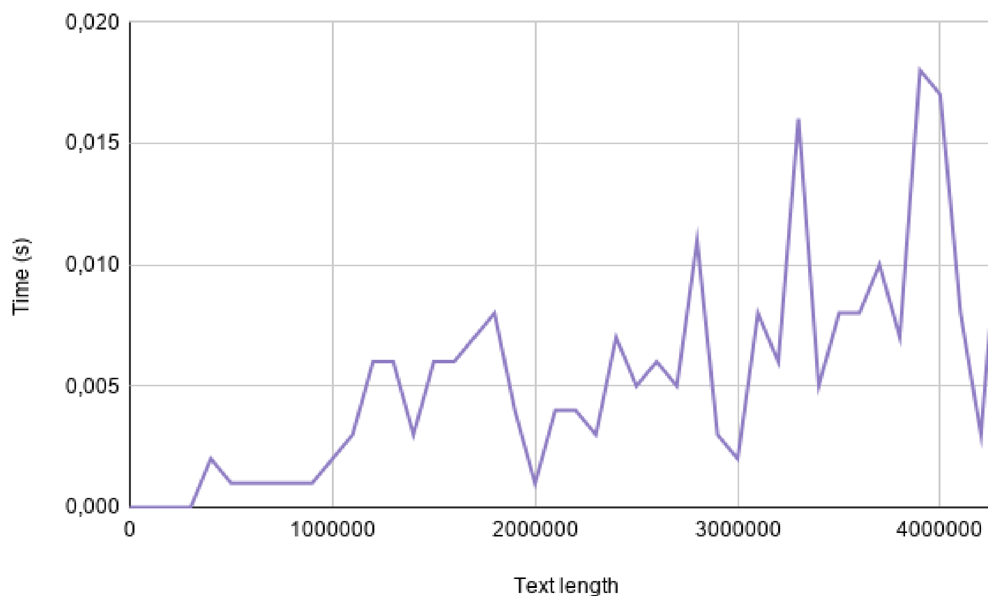
[22] WEINER, P. Linear Pattern Matching Algorithm. In: *In Proc 14th IEEE Symp Switching and Automata Theory*. Listopad 1973, s. 1–11.

[23] WING KIN, K. S. *Suffix Tree and Suffix Array*. Department of Computer Science, School of Computing – National University of Singapore, září 2017. Dostupné z: https://www.comp.nus.edu.sg/~ksung/cs5238/2007Sem1/note/CS5238_2007_Lect4Note_SuffixTree.pdf.

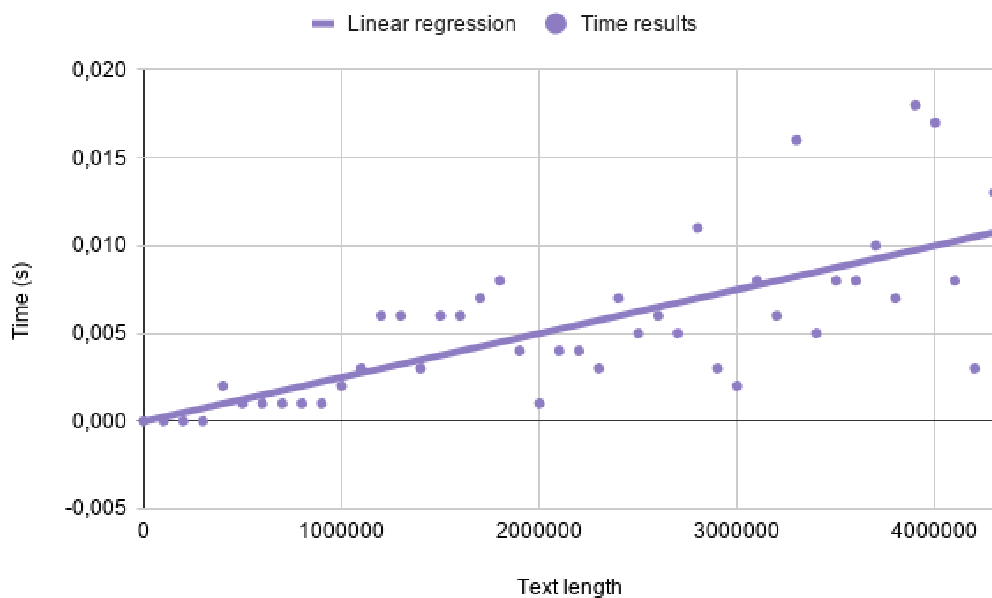
Příloha A

Výsledky měření

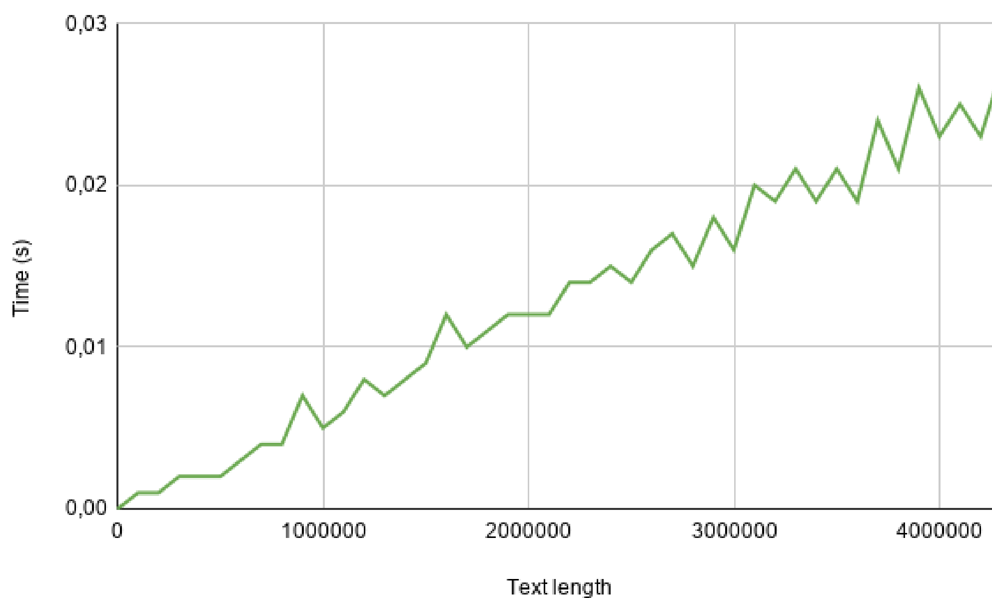
Tato příloha obsahuje grafy s hodnotami časové náročnosti jednotlivých algoritmů. Pro vyhledávání vzoru byl použit vstupní text s velikostí od 0 po 4 300 000 znaků a vzor o délce 100 000 znaků. Vyhledávání nejdelšího společného podřetězce a nejdelšího palindromu proběhlo nad textem o délce 0 až 50 000 znaků.



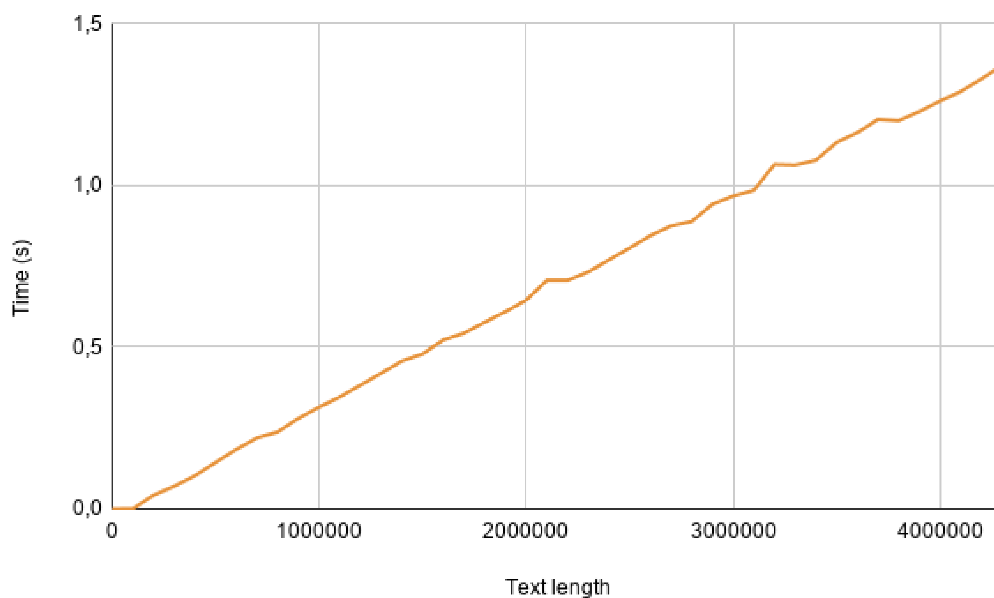
Obrázek A.1: Čas potřebný k vyhledání vzoru v textu za pomoci metody *Indexof*, z knihovny platformy C#, v závislosti na délce textu s fixní délkou vzoru.



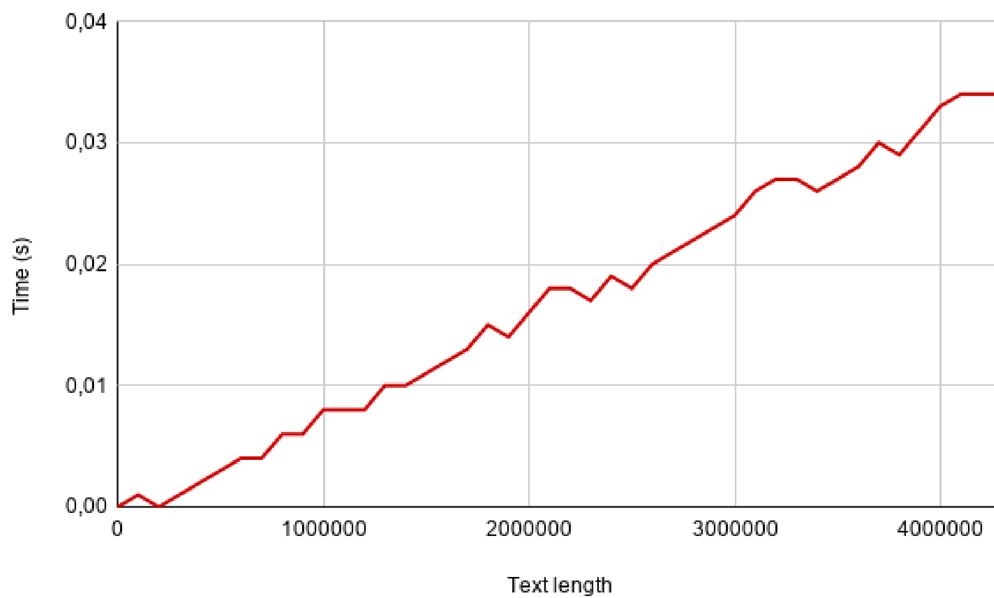
Obrázek A.2: Lineární regrese výsledků metody *Indexof* pomocí metody nejmenších čtverců.



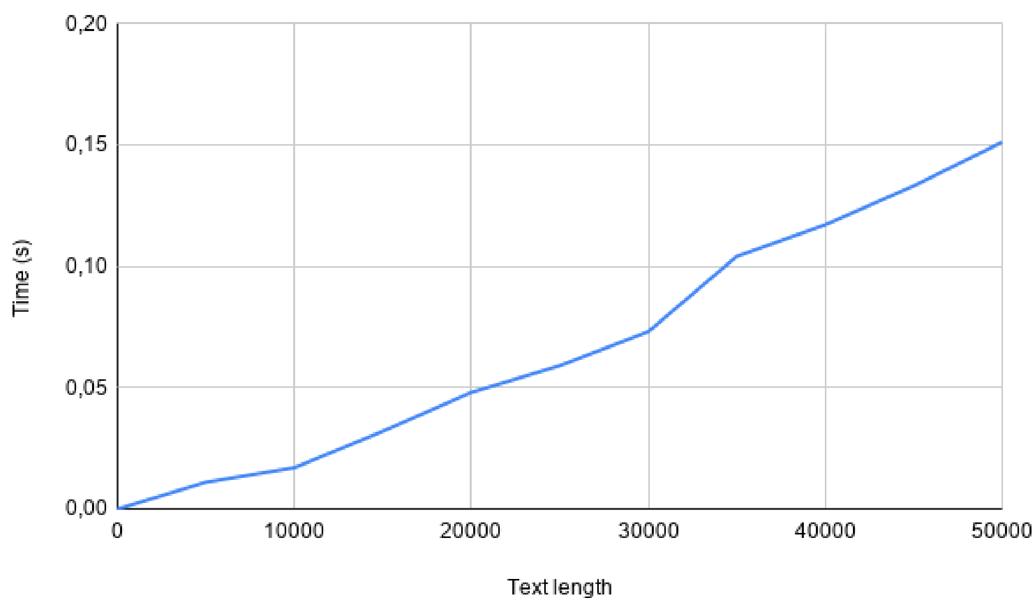
Obrázek A.3: Čas potřebný k vyhledání vzoru v textu za pomoci Knuth–Morris–Pratt algoritmu v závislosti na délce textu s fixní délkou vzoru.



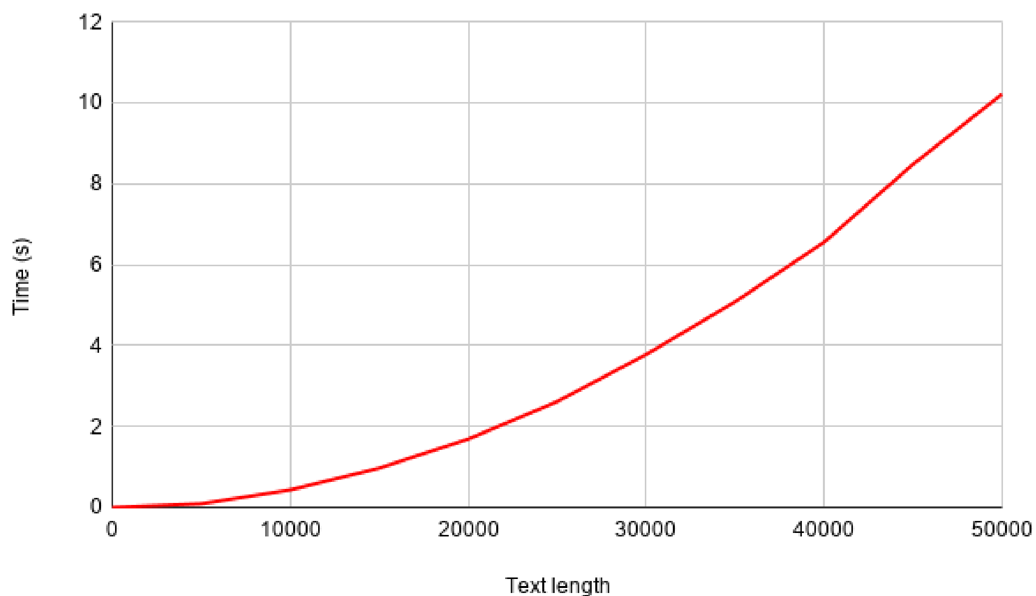
Obrázek A.4: Čas potřebný k vyhledání vzoru v textu za pomoci Rabin–Karp algoritmu v závislosti na délce textu s fixní délkou vzoru.



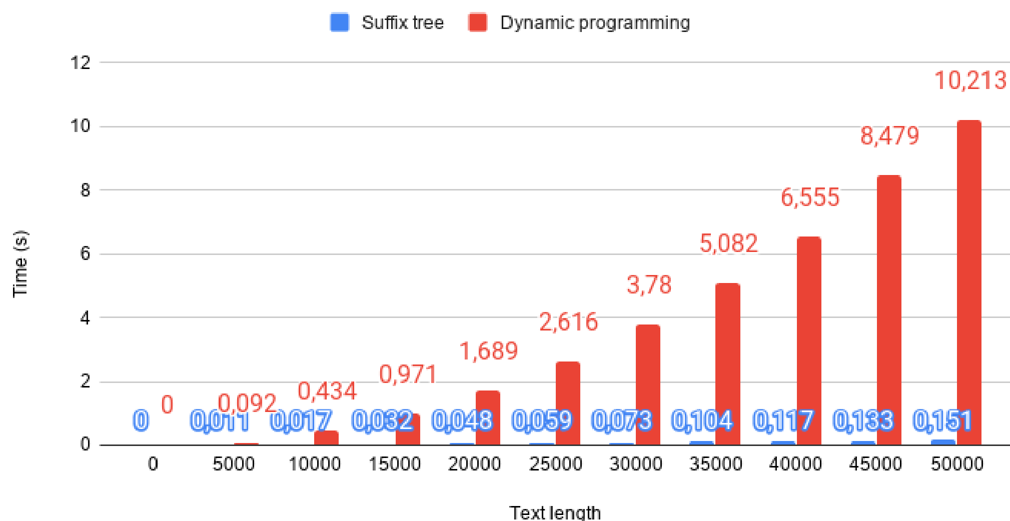
Obrázek A.5: Čas potřebný k vyhledání vzoru v textu za pomoci naivního algoritmu algoritmu v závislosti na délce textu s fixní délkou vzoru.



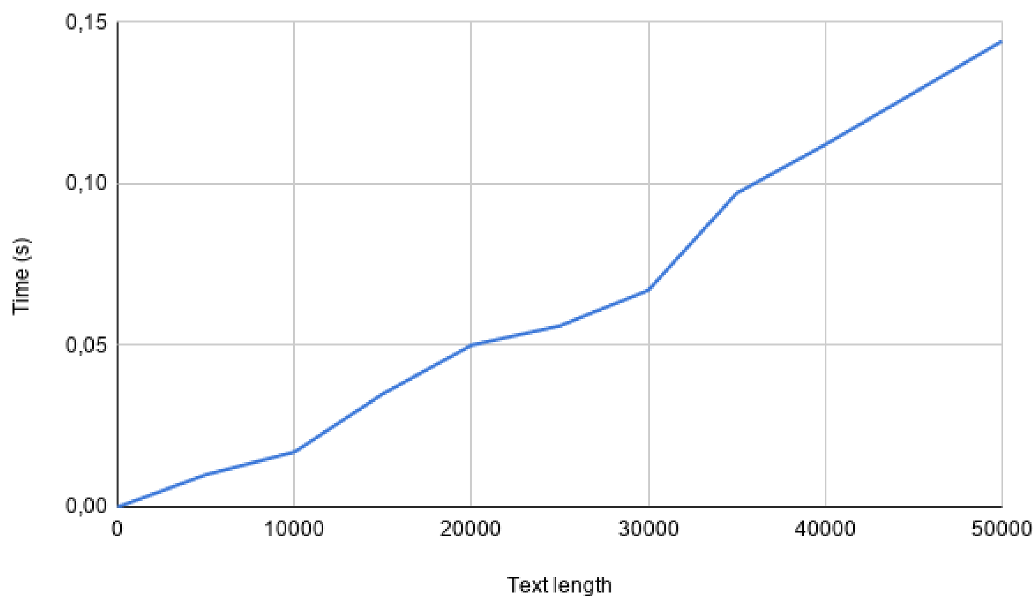
Obrázek A.6: Čas potřebný k vyhledání nejdelšího společného podřetězce dvou textů za pomoci sufixového stromu v závislosti na délce dvou vstupních textů (nejedná se o součet délek řetězců). Do hodnot časové náročnosti je započten i čas potřebný pro sestavení stromu.



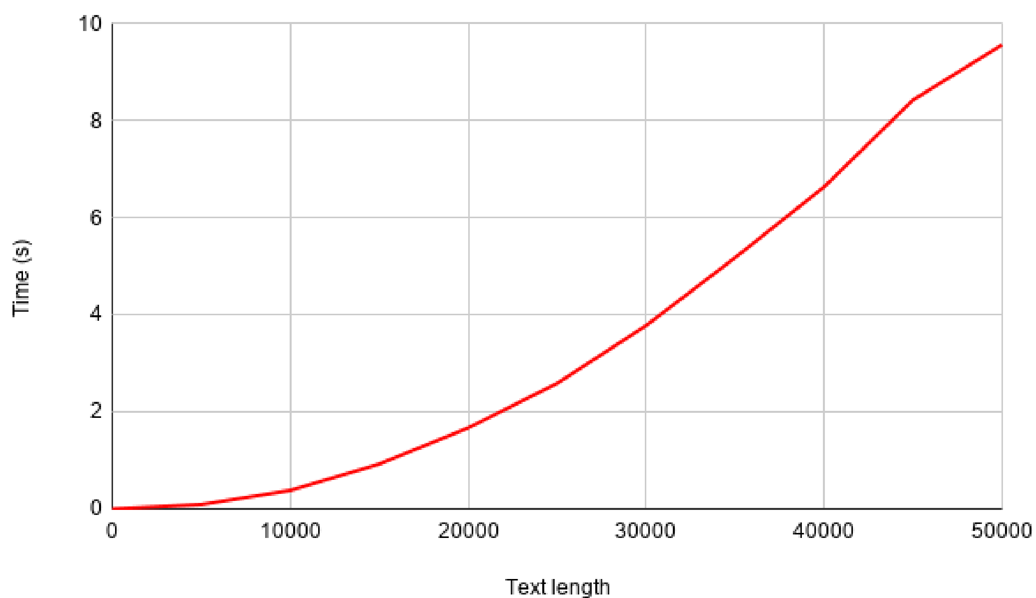
Obrázek A.7: Čas potřebný k vyhledání nejdelšího společného podřetězce dvou textů za pomoci dynamického programování v závislosti na délce dvou vstupních textů (nejedná se o součet délek řetězců).



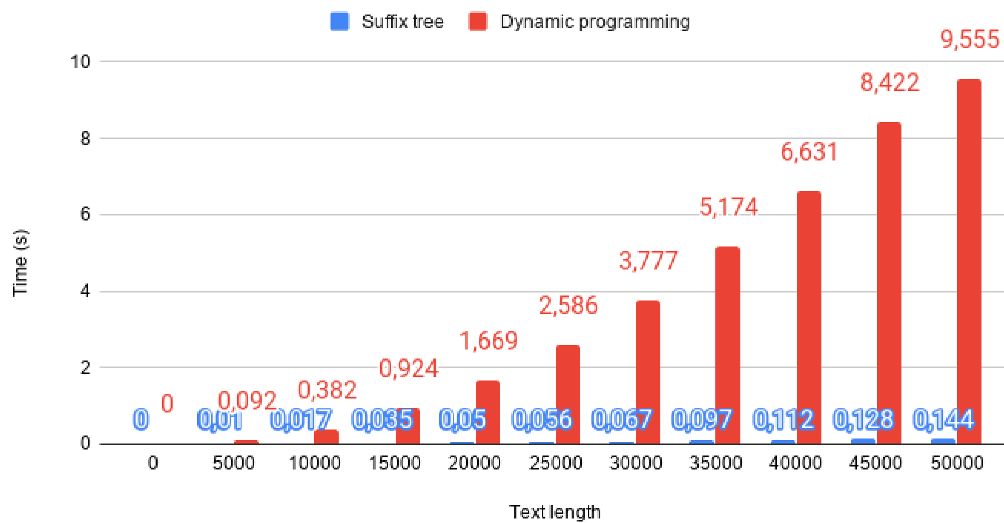
Obrázek A.8: Porovnání časové náročnosti pro vyhledání nejdelšího společného podřetězce za pomoci sufixového stromu (hodnoty časové náročnosti obsahují také čas potřebný ke konstrukci stromu) a dynamického programování v závislosti na délce vstupních textů.



Obrázek A.9: Čas potřebný k vyhledání nejdelšího palindromu v textu za pomoci sufixového stromu v závislosti na délce vstupního textu. Do hodnot časové náročnosti je započten i čas potřebný pro sestavení stromu.



Obrázek A.10: Čas potřebný k vyhledání nejdelšího palindromu v textu za pomoci dynamického programování v závislosti na délce vstupního textu.



Obrázek A.11: Porovnání časové náročnosti pro vyhledání nejdelšího palindromu za pomoci sufixového stromu (hodnoty časové náročnosti obsahují také čas potřebný ke konstrukci stromu) a dynamického programování v závislosti na délce vstupního textu.

Příloha B

Obsah CD

- Soubor README.txt s popisem adresářů
- Návod pro spuštění aplikace a testů
- Zdrojové kódy aplikace
- Spustitelný soubor s výstupní aplikací
- Text bakalářské práce v PDF
- Zdrojový kód bakalářské práce v LaTeX