FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Konstrukce a simulace vyhledávacích automatů přesného a přibližného vyhledávání
Student:	Tomáš Čapek
Vedoucí:	Ing. Jan Trávníček
Studijní program:	Informatika
Studijní obor:	Teoretická informatika
Katedra:	Katedra teoretické informatiky
Platnost zadání:	Do konce zimního semestru 2019/20

Pokyny pro vypracování

1) Nastudujte algoritmy konstrukce a simulace vyhledávacích automatů přesného a přibližného vyhledávání [1].
2) Nastudujte implementaci Knihovny algoritmů vyvíjené na katedře teoretické informatiky Fakulty informačních technologií.
3) Implementujte algoritmy konstrukce automatů pro přesné a přibližné vyhledávání do Knihovny algoritmů.
4) Implementujte algoritmy simulace automatů pro přesné a přibližné vyhledávání do Knihovny algoritmů.
5) Otestujte implementované algoritmy pomocí vhodně zvolených předpřipravených i náhodně generovaných vstupů.

Seznam odborné literatury

[1] Melichar, Borivoj, Jan Holub, and J. Polcar. Text searching algorithms. *Available on: http://stringology.org/athens* (2005).

doc. Ing. Jan Janoušek, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 1. března 2018

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Konstrukce a simulace vyhledávacích automatů přesného a přibližného vyhledávání

Tomáš Čapek

Katedra Teoretické informatiky
Vedoucí práce: Ing. Jan Trávníček

14. května 2018

Poděkování

Chtěl bych poděkovat Ing. Janu Trávníčkovi za ukázkové vedení práce a velkou míru trpělivosti, kterou se mnou při vedení práce měl. Dále pak svým rodičům, za veškerou podporu během studijí. Poděkování také patří všem mým kamarádům za to, že to se mnou až sem vydrželi. Poslední poděkování míří Anetě Kochové za vše, co pro mě v průběhu práce udělala.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 14. května 2018 .....

Odkaz na tuto práci

Čapek, Tomáš. *Konstrukce a simulace vyhledávacích automatů přesného a přibližného vyhledávání*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2018.

Abstrakt

Cílem této práce je implementovat algoritmy konstrukce vyhledávacích automatů. Dále se práce zabývá simulacemi tohoto druhu automatů. V práci jsou implementovány metody, používající Hammingovu, Levenshteinovu a Zobecněnou Levenshteinovu vzdálenost. Tato práce je součástí projektu Algoritmová knihovna.

Klíčová slova konečný automat, přibližné vyhledávání v textu, přesné vyhledávání v textu, simulace vyhledávacích automatů, Algoritmová knihovna

Abstract

The objective of this thesis is to implement algorithms for the construction of search automata. The paper also deals with simulations of this type of automata. Methods using Hamming, Levenshtein and Generalized Levenshtein distance are used. This thesis is a part of the project Algorithm library.

Keywords finite automaton, exact string matching, aproximate string matching, simulation of matching automaton, Algorithm library

Obsah

Seznam obrázků

Úvod

Vyhledávání řetězců v textu je jedním z nejzákladnějších a nejzásadnějších problémů, kterým se obor stringologie zabývá. Přesné vyhledávní lze zobecnit na vyhledávání sekvencí, resp. na vyhledávání s povolenou mírou chybovosti. Tyto přístupy jsou dále aplikované v praxi na širokém množství problémů, například při zpracovávání sekvencí DNA.

Cíl práce

Prvním cílem této práce je nastudování algoritmů konstrukce automatů pro přesné a přibližné vyhledávání řetězců v textu. Dalším cílem je zkoumané algoritmy naimplementovat. Cílem implementace není vytvořit co nejefektivnější řešení, ale naopak mít kód, který odpovídá matematickým předpisům daných algoritmů a je pro studenty snadno čitelný. Samozřejmou součástí této práce je vhodné testování výsledných algoritmů.

Dále se pak práce zabývá simulacemi tohoto druhu automatů. Nejprve je provedeno nastudování možných metod simulací vyhledávacích automatů a po té jsou tyto metody v rámci Algoritmové knihovny implementovány.

Struktura práce

V kapitole 1 jsou zadefinovány základní pojmy z oblasti automatů, gramatik a stringologie, které jsou použity v této práci.

V kapitole 2 je popsána Algoritmová knihovna a je stručně popsána její historie. Zároveň jsou prodiskutována existující alternativní řešení.

V kapitole 3 je probrána teorie související s konstrukcí vyhledávacích automatů.

V kapitole 4 jsou prodiskutuovány možnosti simulací vyhledávacích automatů.

Kapitola 5 se zabývá implementací a testováním.

Kapitola 1

Teoretický úvod

Tato kapitola definuje základní pojmy, které jsou použity ve zbytku této práce.

1.1 Řetězce

Není-li řečeno jinak, jsou všechny definice z této sekce převzány z [2].

Definice 1. *Abeceda* je neprázdná konečná množina symbolů. Abecedu označujeme pomocí symbolu Σ.

Příklad 1.

$\Sigma = \{ 0, 1 \}$ je binární abeceda.

$\Sigma = \{ a, b, c, ..., z \}$ je abeceda složená z malých písmen anglické abecedy.

Definice 2. *Řetězec* nad abecedou Σ je konečná sekvence symbolů z dané abecedy.

Příklad 2. 1001 je řetězec nad abecedou $\Sigma = \{ 0, 1 \}$.

Definice 3. *Prázdný řetězec* je řetězec, který se skládá z nulového počtu symbolů. Značíme pomocí ε.

Definice 4. Pomocí Σ^* označujeme sadu všech řetězců (včetně prázdného řetězce) nad abecedou Σ. Dále pak pomocí Σ^+ označujeme sadu všech neprázdných řetězců nad abecedou Σ.

Definice 5. Mějme řetězec ω nad abecedou Σ. *Délka řetězce* ω je počet symbolů v řetězci ω a označujeme $|\omega|$.

Příklad 3.

$|1001| = 4$

$|\varepsilon| = 0$

Definice 6. Množinu $\overline{p}$ nazveme *doplňkem k symbolu p* v abecedě Σ, pokud pro danou množinu platí:

$$\overline{p} = \{x | x \in \Sigma, x \neq p\}$$

Příklad 4. Mějme abecedu $\Sigma = \{$ a, b, c $\}$. Doplňkem k symbolu c v abecedě Σ je následující množina:

$$\overline{c} = \{a, b\}$$

Definice 7. *Jazyk* je libovolná množina řetězců [3].

Definice 8. Univerzální symbol *„don't care"* je speciální symbol $\circ$, který se rovná libovolnému jinému symbolu, včetně sebe samého [1].

1.2 Konečné automaty

Pokud není uvedeno jinak, jsou definice z této sekce převzány z [4].

Definice 9. *Deterministický konečný automat* je pětice, skládající se z:

1. Konečné množiny *stavů*. Označujeme Q.
2. Konečné množiny *vstupních symbolů*, neboli *vstupní abecedy*. Označujeme Σ.
3. *Přechodové funkce* $\delta : Q \times \Sigma \to Q$
4. *Počátečního stavu* q_0, kde $q_0 \in Q$.
5. Konečné množiny *koncových stavů* F, kde $F \subseteq Q$.

Deterministický konečný automat je v textu označován zkratkou DKA.

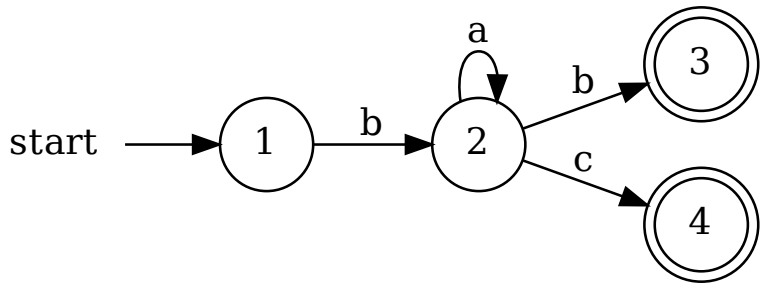
Příklad 5. Uvádím příklad konkrétního DKA.

$$A = (\{1, 2, 3, 4\}, \{a, b, c\}, \delta, 1, \{3, 4\})$$

Kde je δ definována následovně:

$$\delta(1, b) = 2$$
$$\delta(2, a) = 2$$
$$\delta(2, b) = 3$$
$$\delta(2, c) = 4$$

Funkci δ můžeme znázornit pomocí tzv. *diagramu přechodů*. Diagram pro tento automat je na obrázku 1.1.



Obrázek 1.1: Diagram přechodů deterministického konečného automatu A

Definice 10. Nechť $M = (Q, \Sigma, \delta, q_0, F)$ je konečný automat. Dvojice $(q, w) \in Q \times \Sigma$ se nazývá *konfigurace konečného automatu*. Konfiguraci (q_0, w) nazýváme *počáteční konfigurací*. Konfigurace (q, ε), kde $q \in F$, nazýváme *konečnou konfigurací*.

Definice 11. Nechť $M = (Q, \Sigma, \delta, q_0, F)$ je DKA. Relaci $\vdash_M \in (Q \times \Sigma^*) \times (Q \times \Sigma^*)$ nazýváme *přechod v deterministickém konečném automatu M*. Platí, že pokud $\delta(q, a) = p$, pak $(q, aw) \vdash_M (p, w)$ pro všechna $w \in \Sigma^*$. K-násobek vztahu $\vdash_M$ značíme pomocí $\vdash_M^k$. Symboly $\vdash_M^+$ a $\vdash_M^*$ reprezentují tranzitivní a reflexivně tranzitivní uzávěr vztahu $\vdash_M$.

Poznámka: Index M je možné vynechat, pokud je evidentní, o kterém DKA se hovoří.

Definice 12. Říkáme, že *vstupní řetězec* $w \in \Sigma^*$ je *přijímán* konečným deterministickým automatem $M = (Q, \Sigma, \delta, q_0, F)$, pokud existuje $(q_0, w) \vdash_M^* (q, \varepsilon)$, pro nějaké $q \in F$. V opačném případě je *nepřijímán*.

Definice 13. Nechť $M = (Q, \Sigma, \delta, q_0, F)$ je DKA. Jazyk:

$$L(M) = \{w : w \in \Sigma^*, (q_0, w) \vdash_M^* (q, \varepsilon), q \in F\}$$

nazveme *jazykem přijímaným deterministickým konečným automatem M*. Řetězec $w \in L(M)$ obsahuje pouze symboly ze vstupní abecedy a existuje sekvence přechodů taková, že vede z počáteční konfigurace do koncové konfigurace.

Příklad 6. Mějme automat A z příkladu 5. Dále mějme řetězec $w = baac$. Tento řetězec položme na vstup automatu A. Automat A provede následující

přechody:

$(1, baac) \vdash (2, aac) \vdash (2, ac) \vdash (2, c) \vdash (4, \varepsilon)$

Automat A tedy řetězec w přijímá.

Definice 14. *Nedeterministický konečný automat* je pětice, skládající se z:

1. Konečné množiny *stavů*. Označujeme Q.
2. Konečné množiny *vstupních symbolů*. Označujeme Σ.
3. *Přechodové funkce* δ, která mapuje z $Q \times \Sigma$ na podmnožinu Q.
4. *Počátečního stavu* q_0, kde $q_0 \in Q$.
5. Konečné množiny *koncových stavů* F, kde $F \subseteq Q$.

Nedeterministický konečný automat je v textu označován zkratkou NKA.

Hlavní rozdíl mezi deterministickým a nedeterministickým konečným automatem je, že nedeterministický automat používá u funkce δ množinu stavů. Díky tomu definice 10 platí a je nutné revidovat definice pro přechod a přijetí.

Definice 15. Nechť $M = (Q, \Sigma, \delta, q_0, F)$ je NKA. Relaci $\vdash_M \subset (Q \times \Sigma^*) \times (Q \times \Sigma^*)$ nazveme *přechod v automatu* M, právě tehdy, když $p \in \delta(q, a)$, tak platí $(q, aw) \vdash_M (p, w)$, pro všechna $w \in \Sigma^*$.

Definice 16. Nechť $M = (Q, \Sigma, \delta, q_0, F)$ je NKA. Řetězec $w \in \Sigma^*$ je *přijímán* nedeterministickým konečným automatem, právě tehdy, když existuje sekvence přechodů $(q_0, w) \vdash_M^* (q, \varepsilon)$ pro nějaké $q \in F$. V opačném případě je *nepřijímán*.

Definice 17. Nechť $M = (Q, \Sigma, \delta, q_0, F)$ je NKA. Jazyk

$$L(M) = \{w : w \in \Sigma^*, (q_0, w) \vdash_M^* (q, \varepsilon), q \in F\}$$

nazveme *jazykem přijímaným nedeterministickým konečným automatem* M

Následující definice jsou převzány z [1].

Definice 18. *Nedeterministický konečný automat s ε-přechody* je pětice, skládající se z:

1. Konečné množiny *stavů*. Označujeme Q.
2. Konečné množiny *vstupních symbolů*. Označujeme Σ.
3. *Přechodové funkce* δ, která mapuje z $Q \times (\Sigma \cup \{\varepsilon\})$ na podmnožinu Q.

4. *Počátečního stavu* q_0, kde $q_0 \in Q$.

5. Konečné množiny *koncových stavů* F, kde $F \subseteq Q$.

Definice 19. Nechť $M = (Q, \Sigma, \delta, q_0, F)$ je NKA s ε-přechody. Relaci $\vdash_M \subset (Q \times \Sigma^*) \times (Q \times \Sigma^*)$ nazveme přechodem v automatu M, právě tehdy když $p \in \delta(q, a), a \in \Sigma \cup \{\varepsilon\}$, tak existuje $(q, aw) \vdash_M (p, w)$ pro všechna $w \in \Sigma^*$.

Poznámka: Definice jazyka přijímaného NKA s ε-přechody je stejná, jako pro obyčejný NKA.

Definice 20. Nechť $M = (Q, \Sigma, \delta, q_0, F)$ je libovolný konečný automat. Stav $q \in Q$ nazveme *dosažitelný*, pokud existuje takový řetězec $w \in \Sigma^*$, pro který existuje posloupnost přechodů z počátečního stavu q_0 do stavu q:

$$(q_0) \vdash_M (q, \varepsilon)$$

Stav, který není dosažitelný je *nedosažitelný*.

Algoritmus 1. Nalezení a odstranění nedosažitelných stavů [5].
Vstup: Konečný automat $M = (Q, \Sigma, \delta, q_0, F)$.
Výstup: Konečný automat $M' = (Q', \Sigma, \delta', q_0, F')$, který nemá žádné nedosažitelné stavy takový, že $L(M) = L(M')$.

Metoda:

1. Určíme množinu všech dosažitelných stavů Q_a takto:
   a) $Q_0 = \{q_0\}, i = 1$.
   b) $Q_i = \{q : q \in \delta(p, a), p \in Q_{i-1}, a \in \Sigma\} \cup Q_{i-1}$.
   c) $Q_i \neq Q_{i-1}$, pak $i = i + 1$ a jdi na krok b), jinak $Q_a = Q_i$.
2. Výsledný automat sestrojíme takto:
   $M' = (Q_a, \Sigma, \delta', q_0, F \cap Q_a)$, kde δ' bude zkonstruována takto:
   $\delta'(q, a) = \delta(q, a)$ pro všechna $q \in Q_a$.

1.3 Vyhledávání

Definice z této sekce jsou převzány z [1].

Definice 21. *Vzor* je neprázdný jazyk, který neobsahuje prázdný řetězec. [6]

Pro potřeby této práce se ale pracuje pouze se vzorem, který obsahuje jen jeden řetězec.

Definice 22. *Vyhledávání vzoru v řetězci* je problém, který řeší zda-li se daný vzor vyskytuje v daném řetězci [6].

Definice 23. Existují tři varianty vzdáleností mezi dvěma řetězci x a y, které jsou definovány podle minimálního počtu úprav

1. nahrazení (Hammingova vzdálenost, R-vzdálenost),
2. smazání, vložení a nahrazení (Levenshteinova vzdálenost, DIR-vzdálenost),
3. smazání, vložení, nahrazení a prohození sousedních symbolů (Damerauova vzdálenost, Zobecněná Levenshteinova vzdálost, DIRT-vzdálenost),

potřebných k tomu, aby se řetězec x transformoval na řetězec y.

Hammingova vzdálenost je metrika pouze pro řetězce stejné délky. Leveshteinova a Zobecněná Levenshteinova vzdálenost je aplikovatelná i na řetězce různé délky.

Vyhledávání vzoru v řetězci lze rozdělit na základní problémy, které řeší:

přesné vyhledávání Tedy vyhledávání dle definice 22.

vyhledávání sekvencí symbolů ze vzoru Ověřuje, zda-li se sekvence P vyskytuje v daném řetězci.

vyhledávání části vzoru Řeší, zda-li se nějaká část vzoru P vyskytuje v daném řetězci.

přibližné vyhledávání Ověřuje, zda-li se vzor P vyskytuje v řetězci tak, že jeho vzdálenost od řetězce je menší než daná vzdálenost.

vyhledávání s „don't care" symbolem Zjišťuje, zda-li se vzor obsahující „don't care" symbol vyskytuje v řetězci.

Kapitola 2

Algoritmová knihovna

V této kapitole je popsán projekt Algoritmová knihovna a jeho historie. Na závěr jsou prodiskutována alternativní existující řešení.

2.1 Úvod

Algoritmová knihovna je rozsáhlý projekt, který vznikl pod vedením Ing. Jana Trávníčka, na půdě FIT ČVUT. Cílem projektu je vytvořit knihovnu pro podporu výuky předmětů z oboru Teoretická informatika, jakými jsou například BI-AAG [1], BI-GRA [2] nebo MI-EVY [3].

Jedná se o sadu samostatných knihoven a spustitelných programů, napsaných v jazyce C++. Tento projekt ku příkladu obsahuje následující nástroje:

aconvert2 Tento nástroj umožňuje velké množství převodů mezi vnitřní reprezentací a okolním světem. Jako příklad uvádím převod vnitřní reprezentace konečného automatu do formátu DOT.

arand2 Jedná se o nástroj, který umožňuje náhodně generovat různé struktury, se kterými Algoritmová knihovna pracuje. Podporuje například generování nedeterministických konečných automatů nebo řetězců.

arun2 Díky tomuto nástroji je možné spustit libovolný automat s libovolným vstupem a zjistit výsledek jeho běhu.

aql2 Tento nástroj je plnohodnotný interaktivní shell, který umožňuje praktický přístup ke všem aspektům Algoritmové knihovny.

[1] Automaty a gramatiky
[2] Grafové algoritmy
[3] Efektivní vyhledávání v textu

2.2 Stručná historie Algoritmové knihovny

Práce na knihovně začala v roce 2014 prací Martina Žáka [7], který položil základ vytvořením vnitřního a komunikačního formátu. V tom samém roce Tomáš Pecka vytvořil algoritmy pro vzájemné převody regulárních výrazů, konečných automatů a regulárních gramatik [8]. Dále pak Jan Veselý [9] přidal podporu pro determinizaci konečných a zásobníkových automatů. Tato práce však byla později vylepšena Ing. Janem Trávníčkem.

V roce 2015 přidal Štěpán Plachý [10] podporu pro stromy a stromové automaty, i s některými základními algoritmy, jako například generování náhodných stromů.

V roce 2016 pak Jan Brož přidal podporu pro grafové algoritmy. Jedná se o algoritmy nalezení minimální kostry, maximálního toku a minimálního řezu pro orientované i neorientované grafy [11]. Dále pak David Rosca vytvořil algoritmus pro ověření isomorfismu grafů [12]. Tato práce je doposud pouze experimentální. V další práci Martin Kočička vytvořil LR-parser [13]. Algoritmové knihovně se také opět věnoval Tomáš Pecka, jehož práce se zaobírala regulárními stromovými výrazy a jejich převody [14]. Robin Obůrka přidal podporu pro vyhledávání ve stromech. Konkrétně pomocí mrtvých zón a sousměrného algoritmu [15].

V roce 2017 se pokusil Jan Parma vytvořit podporu pro kompresní algoritmy [16], nicméně se zatím nejedná o dokončenou část Algoritmové knihovny. Dále pak Ing. Trávníček v tomto roce dokončil formalismy pro transformace gramatik. Václav Mareš se pokusil vytvořit GUI [17], které bylo velkou inspirací pro Ing. Trávníčka pro vytvoření CLI rozhraní. Alexander Shatrovskii ve své práci vytvořil algoritmus pro vyhledávání repetic ve stromových strukturách [18]. Poslední prací, která přispěla do tohoto rozsáhlého projetku, je další práce od Štěpána Plachého, který se v ní věnoval minimalizacím stromových a zásobníkových automatů [19].

2.3 Alternativní existující řešení

V této kapitole se nevěnuji alternativám celé Algoritmové knihovny, ale pouze těmi, které řeší stejný problém, jako tato práce.

2.3.1 klawson88/LevenshteinAutomaton

Tento projekt [20] napsaný v jayzce Java slibuje efektivní práci kolem Levenshteinova vyhledávacího automatu, za pomocí dynamického programování a simulace běhu konečného automatu. Tato malá knihovna umí pouze zjišťovat informace ohledně vzdáleností řetězců a jejich sad, nikoliv vyhledávat. Podporuje pouze DIR-vzdálenost. Neobsahuje možnost konstrukce automatu jako takového.

2.3.2 libFLASM

Knihovna *libFLASM* [21] umožňuje přibližné vyhledávní nad textem fixní velikosti. Podporuje Hammingovu vzdálenost (D-vzdálenost) a DIR-vzdálenost. Neobsahuje možnost konstrukce konkrétních automatů.

2.3.3 byteseek

Projekt *byteseek* [22] je knihovna napsaná v jayzce Java, která poskytuje nástroje pro vyhledávání řetězců v textu v sublineárním čase. Poskytuje možnost vyhledávat jak řetězce, tak sekvence symbolů nebo bytů. Navíc obsahuje i nástroj pro vyhledávání pomocí regulárních výrazů. Neobsahuje možnost konstrukce konkrétního vyhledávacího automatu.

2.3.4 dk.brics.automaton

Tato knihovna [23] je projekt vytvořený v jazyce Java, který se zaměřuje na efektivní práci s konečnými automaty se zaměřením na práci s regulární výrazy. Projekt obsahuje implementaci deterministických i nedeterministických automatů, umožňuje je konstruovat (hlavně z regulárních výrazů) a poté je spouštět se zadaným vstupem. Neobsahuje možnost přibližného vyhledávání a nesestavuje automaty podle námi zkoumaných metod.

Kapitola 3

Vyhledávání v textu

V této kapitole je probrána teorie, na základě které je posléze provedena implementace konstrukční části.

Nejprve bude problém vyhledávání v textu rozdělen na podproblémy, které tato práce řeší. Následně se práce zabývá přesným vyhledáváním. Poté je probráno vyhledávání přibližné. Na závěr je prodiskutováno vyhledávání s „don't care“ symbolem.

Není-li uvedeno jinak, je zdrojem této kapitoly [1].

3.1 Úvod do problematiky

V následujících sekcích jsou probrány metody konstrukce konečných automatů, které řeší různě formulované problémy kolem vyhledávání řetězců v textu. Po dohodě s vedoucím práce se tato práce zabývá pouze následujícími problémy:

- přesné a přibližné vyhledání jednoho vzoru,
- přesné a přibližné vyhledání sekvence,
- přesné a přibližné vyhledání vzoru a sekvence s „don't care“ symbolem.

3.2 Přesné vyhledávání

Jedná se o vyhledávání podle definice 22. Výstupem každého algoritmu je konečný automat, který přijímá řetězce, jež obsahují námi definovaný vzor.

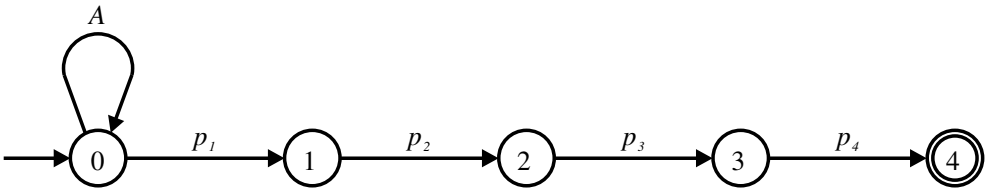
3.2.1 Přesné vyhledání jednoho vzoru

Tento automat je ze všech probíraných automatů ten úplně nejjednodušší. Jeho konstrukce je velmi přímočará, ale pro úplnost uvádím formální zápis tohoto algoritmu.

Algoritmus 2. Konstrukce automatu pro přesné vyhledávání jednoho vzoru.
Vstup: Vzor $P = p_1p_2...p_m$
Výstup: NKA pro vyhledání jednoho vzoru v řetězci.
Metoda: NKA $M = (\{q_0, q_1, ..., q_m\}, \Sigma, \delta, q_0, q_m)$. Funkci δ konstruujeme následovně:

1. $q_{i+1} \in \delta(q_i, p_{i+1})$ pro $0 \leq i < m$
2. $q_0 \in \delta(q_0, a)$ pro všechna $a \in \Sigma$

Diagram přechodů výsledného automatu se nachází na obrázku 3.1.



Obrázek 3.1: Diagram přechodů NKA pro vyhledání vzoru $P = p_1p_2p_3p_4$ [1].

3.2.2 Přesné vyhledání jedné sekvence

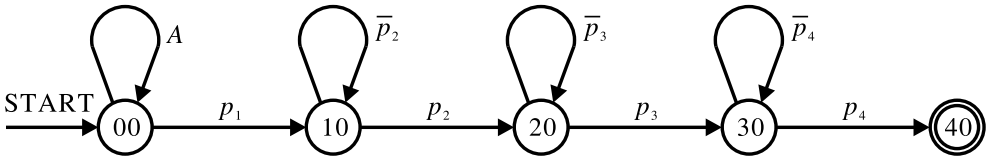
Tento algoritmus vznikne úpravou 2 a to vytvořením smyček v jednotlivých stavech následujícím způsobem. Vytvořené smyčky slouží k tomu, aby pohltily symboly, které se nenacházejí v daném vzoru. Upravený algoritmus vypadá následovně:

Algoritmus 3. Konstrukce automatu pro přesné vyhledávání jedné sekvence.
Vstup: Vzor $P = p_1p_2...p_m$
Výstup: NKA pro vyhledání jedné sekvence v řetězci.
Metoda: NKA $M = (\{q_0, q_1, ..., q_m\}, \Sigma, \delta, q_0, q_m)$. Funkci δ konstruujeme následovně:

1. $q_i \in \delta(q_i, a)$ pro $0 < i < m$ a pro všechna $a \in \Sigma$, pro která platí $a \neq p_{i+1}$
2. $q_{i+1} \in \delta(q_i, p_{i+1})$ pro $0 \leq i < m$
3. $q_0 \in \delta(q_0, a)$ pro všechna $a \in \Sigma$

Poznámka: Kroky 2 a 3 jsou stejné jako v algoritmu 2.

Diagram přechodů výsledného automatu se nachází na obrázku 3.2.



Obrázek 3.2: NKA pro vyhledávání jedné sekvence ze vzoru $P = p_1p_2p_3p_4$ [1].

3.3 Přibližné vyhledávání

V této sekci jsou probrány tři druhy přibližného vyhledávání v textu.

Přibližné vyhledávání dělíme podle vzdálenosti, kterou používáme k měření odlišnosti vyhledávaných řetězců, viz definice 23.

Definice 24. *Úrovní* nazveme počet chyb během nějaké přibližné vyhledávací procedury.

3.3.1 Vyhledávání jednoho vzoru pomocí Hammingovy vzdálenosti

K sestavení tohoto automatu potřebujeme navíc jeden parametr, který bude určovat počet chyb. Tento parametr nazveme k.

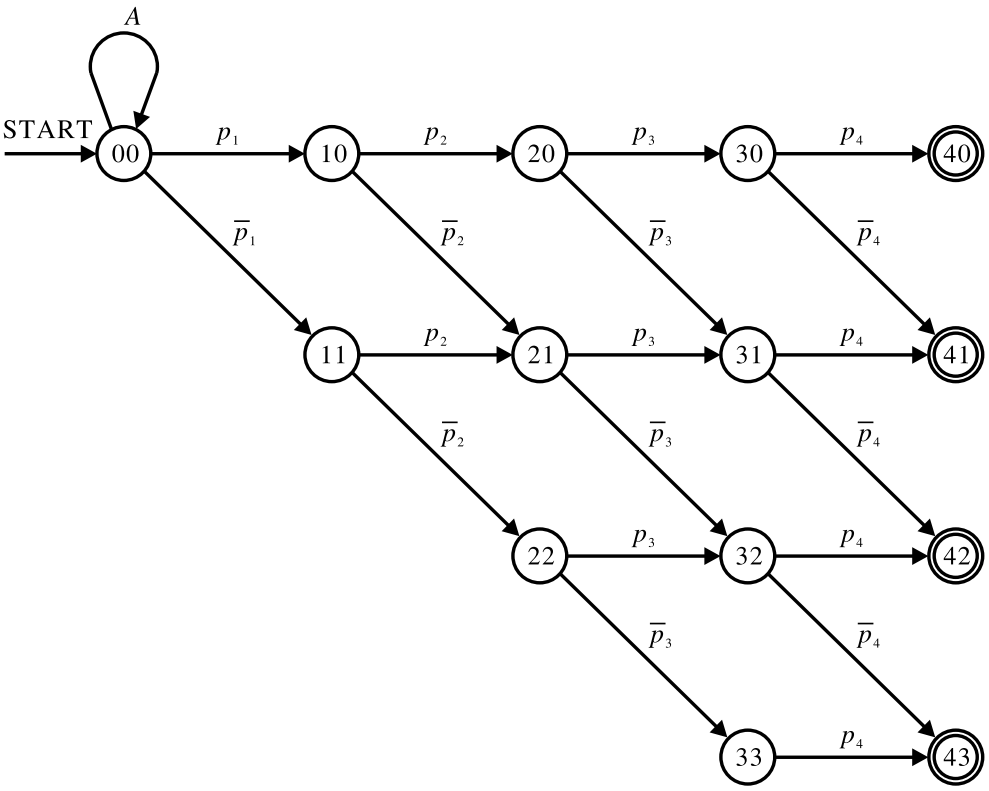
Pro sestavení tohoto druhu automatu využijeme již známý algoritmus 2. Sestavíme si nejprve $k + 1$ těchto automatů a následně přidáme „diagonální“ přechody, které budou využity pro přechod při nalezení chyby ve vyhledáváném řetězci. Kopie těchto automatů nám udržují informaci o současné úrovni, ve které se vyhledávací proces nachází. V průběhu tohoto procesu ale vzniknou nedostažitelné stavy, které je nutné odstranit. Ukázka výsledného automatu je na obrázku 3.3.

Algoritmus formálně zapíšeme následujícím způsobem:

Algoritmus 4. Konstrukce automatu pro nalezení vzoru s Hammingovou vzdáleností nejvýše k.
Vstup: Vzor $P = p_1p_2...p_m$, nejvyšší povolený počet chyb k
Výstup: NKA pro nalezení jednoho řetězce s Hammingovou vzdáleností maximálně k.
Metoda:

1. Vytvoříme sekvenci $k + 1$ automatů dle algoritmu 2. Tyto automaty nazveme $M' = (Q'_j, \Sigma, \delta'_j, q'_{0j}, F'_j)$ pro $j = 0, 1, ..., k$. Jejich stavy nazveme $q_{0j}, q_{1j}, ..., q_{mj}$.

2. Vytvoříme výsledný automat $M = (Q, \Sigma, \delta, q_0, F)$ následujícím způsobem:
   $Q = \bigcup_{j=0}^{k} Q'_j$,
   $\delta(q, a) = \delta'_j(q, a)$ pro všechna $q \in Q, a \in \Sigma, j = 0, 1, 2, ..., k$,



Obrázek 3.3: Přechodová funkce vyhledávacího automatu využívající Hammingovu vzdálenost pro vzor $P = p_1p_2p_3p_4$, kde $k = 3$ [1]

$q_0 = q_{00}$,
$F = \bigcup_{j=0}^{k} F'_j$

3. Odstraníme všechny stavy, které jsou nedostupné z q_0.

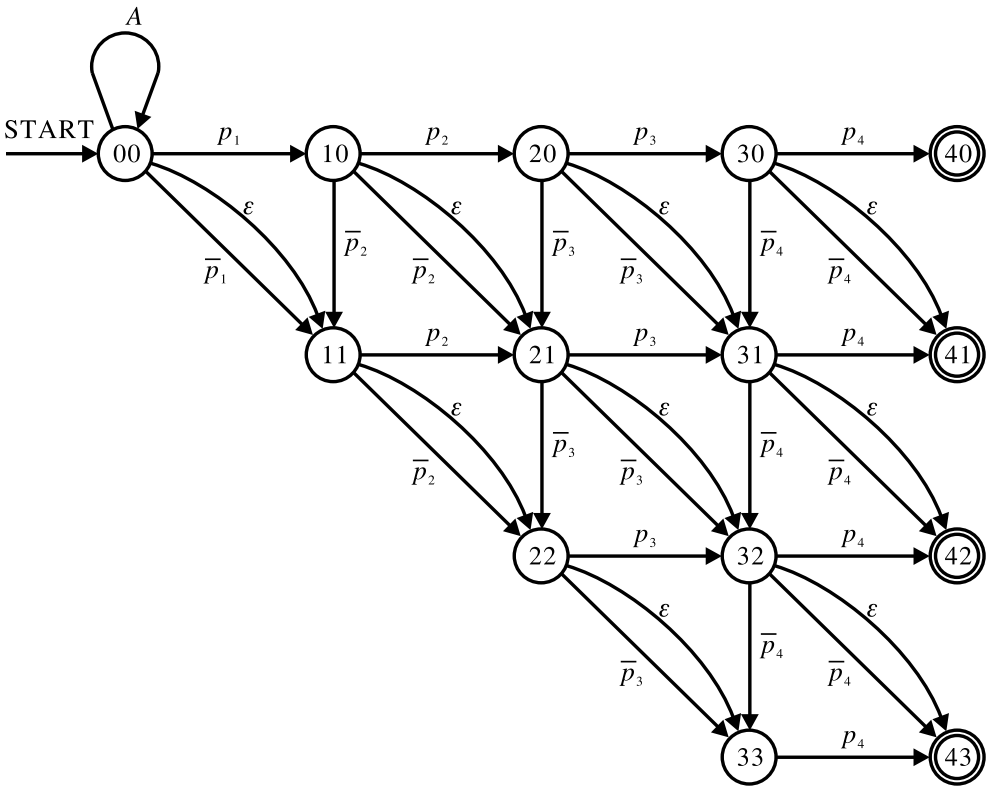
3.3.2 Vyhledání jednoho vzoru pomocí Levenshteinovy vzdálenosti

Konstrukci tohoto automatu zajistíme rozšířením algoritmu 4. Protože DIR-vzdálenost obsahuje kromě operace nahrazení i operace vložení a odstranění, je nutné tyto operace zohlednit v přechodové funkci a to následujícími způsoby:

1. Přidáním ε-přechodů, které budou reprezentovat odstranění symbolu. Tyto přechody budou „diagonální“.

2. Přidáním „vertikálních“ přechodů, které reprezentují vložení symbolu.

Výsledný automat se nachází na obrázku 3.4.



Obrázek 3.4: Přechodová funkce vyhledávacího automatu vyhledávající vzor s Levenshteinovou vzdáleností nejvýše 3. Vzorem je $P = p_1p_2p_3p_4$ [1].

Formálně tento algoritmus zapíšeme následovně:

Algoritmus 5. Konstrukce automatu vyhledávající vzor s Levenshteinovu vzdáleností nejvýše k.
Vstup: Vzor $P = p_1p_2...p_m$, nejvyšší povolený počet chyb k
Výstup: NKA s ε-přechody pro nalezení jednoho vzoru s Levenshteinovou vzdáleností nejvýše k.
Metoda:
Sestavíme automat pomocí algoritmu 4. Tento automat nazveme $M' = (Q', \Sigma, \delta', q'_{00}, F')$.

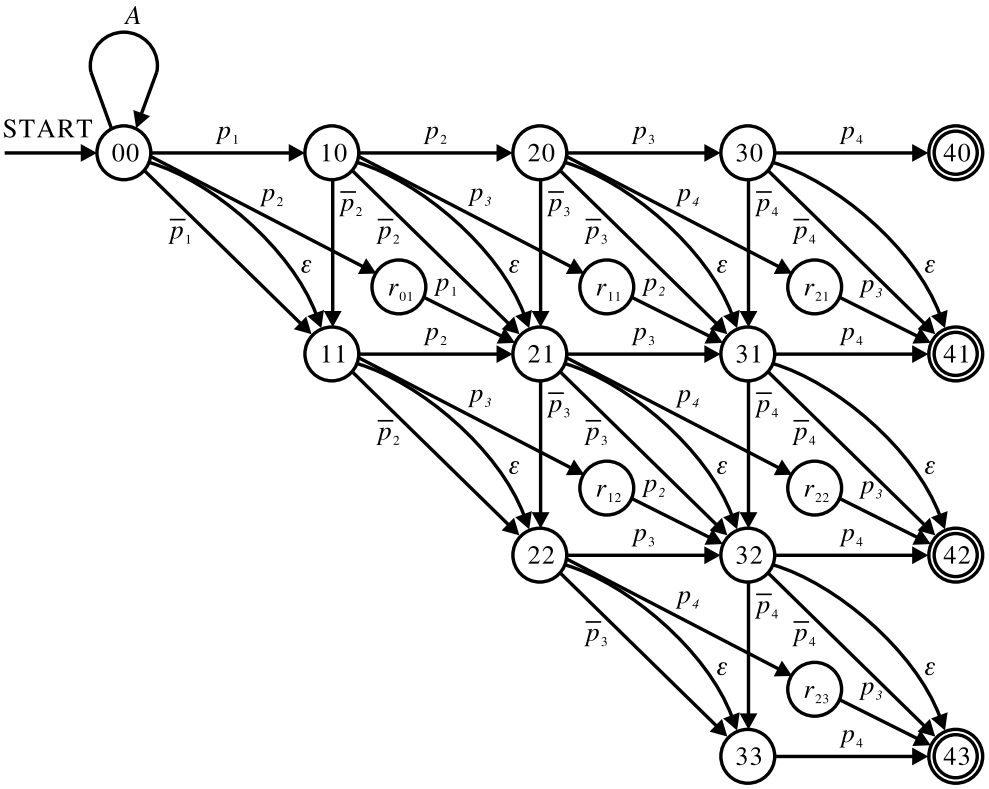
Poznámka: Automat M' bude mít stavy v následujícím tvaru:

$$\begin{array}{ccccccc} q'_{00} & q'_{01} & q'_{02} & q'_{03} & \cdots & q'_{0m} \\ & q'_{11} & q'_{12} & q'_{13} & \cdots & q'_{1m} \\ & & \ddots & & & \\ & & & q'_{kk} & \cdots & q'_{km} \end{array}$$

Výsledný automat $M = (Q, \Sigma, \delta, q_0, F)$ sestavíme následovně:
$\delta(q, a) = \delta'(q, a)$ pro všechna $q \in Q, a \in \Sigma$,
$\delta(q_{ij}, \varepsilon) = \{q_{i+1,j+1}\}$, pro všechna $i = 0, 1, ..., m-1, j = 0, 1, 2, ..., k-1$,
$\delta(q_{ij}, a) = \{q_{i,j+1}\}$, pro všecha $i = 1, 2, ..., m-1, j = 0, 1, 2, ..., k-1, a \in \Sigma \setminus \{p_{i+1}\}$.

3.3.3 Vyhledávání jednoho vzoru pomocí Zobecněné Levenshteinovy metody

Tento automat sestrojíme rozšířením algoritmu 5. Automat rozšíříme o „ploché diagonální“ přechody, které zajistí prohození dvou sousedících symbolů. Obrázek 3.5 obsahuje ukázku takového automatu.



Obrázek 3.5: Přechodová funkce vyhledávacího automatu, který vyhledává vzor se Zobecněnou Levenshteinovou vzdáleností nejvýše 3. Vzorem v tomto případě je $P = p_1p_2p_3p_4$ [1].

Algoritmus formálně zapíšeme následovně:

Algoritmus 6. Konstrukce automatu vyhledávajícího vzor se Zobecněnou Levenshteinovou vzdáleností nejvýše k.
Vstup: Vzor $P = p_1p_2...p_m$, nejvyšší povolený počet chyb k

Výstup: NKA s ε-přechody pro nalezení jednoho řetězce se Zobecněnou Levenshteinovou vzdáleností nejvýše k.
Metoda:
Sestavíme automat pomocí algoritmu 4. Tento automat nazveme $M' = (Q', \Sigma, \delta', q'_{00}, F')$.

Poznámka: Automat M' bude mít stavy v následujícím tvaru:

$$\begin{array}{ccccccc} q'_{00} & q'_{01} & q'_{02} & q'_{03} & \cdots & q'_{0m} \\ & q'_{11} & q'_{12} & q'_{13} & \cdots & q'_{1m} \\ & & \ddots & & & \\ & & & q'_{kk} & \cdots & q'_{km} \end{array}$$

Výsledný automat $M = (Q, \Sigma, \delta, q_0 0, F)$ sestavíme následujícím způsobem:
$Q = Q' \cup \{r_{ij} : j = 1, 2, ..., k, i = j-1, j, ..., m-2\}$,
$\delta(q, a) = \delta'(q, a)$ pro všechna $q \in Q, a \in \Sigma \cup \{\varepsilon\}$,
$\delta(q_{ij}, a) = r_{ij}, j = 1, 2, ..., k, i = j-1, j, ..., m-2$, pokud $\delta(q_{i+1,j}, a) = q_{i+2,j}$,
$\delta(r_{ij}, a) = q_{i+2,j+1}, j = 1, 2, ..., k, i = j-1, j, ..., m-2$, pokud $\delta(q_{ij}, a) = q_{i+1,j}$.

3.3.4 Přibližné vyhledání jedné sekvence

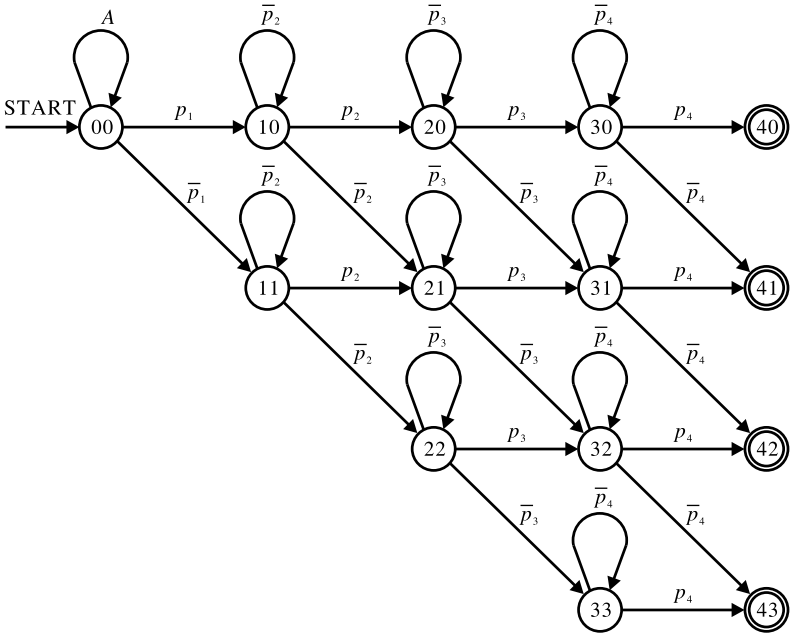
V prvním kroku algoritmu si sestavíme druh vyhledávacího automatu, který chceme využít pro vyhledávání sekvencí. V druhém kroku pak stačí přidat „smyčky“ do stavů, které mají alespoň jeden odchozí přechod.

Na následujících obrázcích se nacházejí modifikované verze předchozích automatů pro vyhledávání sekvencí. Jedná se o automaty využívající Hammingovu vzdálenost 3.6, Levenshteinovu vzdálenost 3.7 a o Zobecněnou Levenshteinovu vzdálenost 3.8.

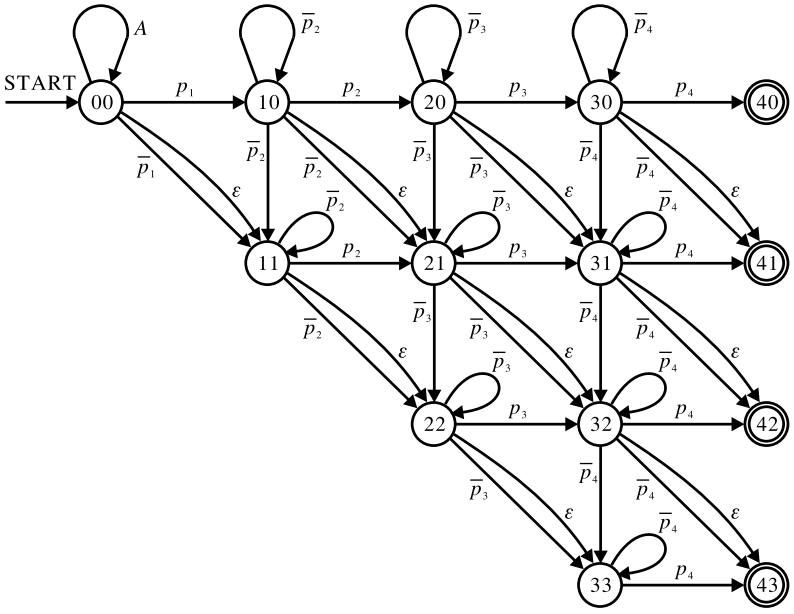
Formálně tento algoritmus zapíšeme následujícím způsobem:

Algoritmus 7. Transformace vyhledávacího automatu pro jeden vzor na automat pro vyhledávání jedné sekvence.
Vstup: Vyhledávací automat pro jeden vzor $M' = (Q', \Sigma, \delta', q'_0, F')$
Výstup: Vyhledávací automat M pro vyhledání jedné sekvence.
Metoda: Sestav automat $M = (Q, \Sigma, \delta, q_0, F)$ následujícím způsobem:

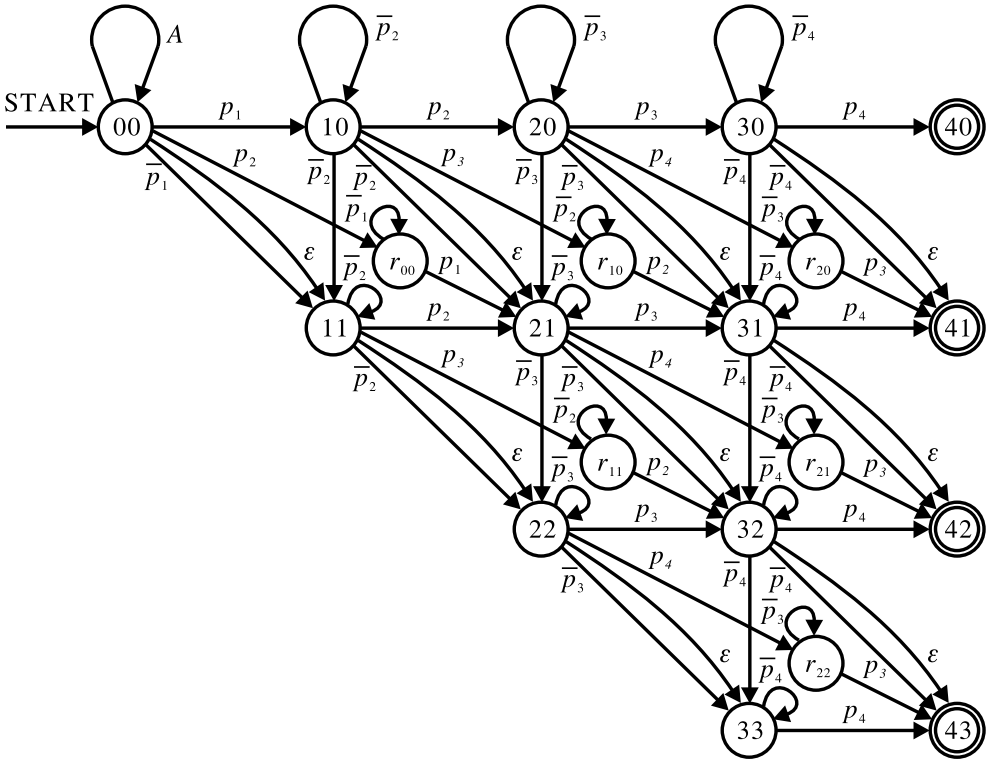
1. $Q = Q'$.
2. $\delta(q, a) = \delta'(q, a)$ pro všechna $q \in Q, a \in \Sigma \cup \{\varepsilon\}$.
3. $\delta(q, a) = \delta'(q, a) \cup \{q\}$ pro taková $q \in Q$, že $\delta(q, a) \neq \emptyset, a \in \overline{p_i}$, kde p_i je symbol na odchozí hraně ze stavu q do dalšího stavu na stejné úrovni.
4. $q_0 = q'_0$
5. $F = F'$



Obrázek 3.6: Přechodová funkce vyhledávacího automatu, který vyhledává sekvence s Hammingovou vzdáleností nejvýše 3 pro vzor $P = p_1p_2p_3p_4$ [1].



Obrázek 3.7: Přechodová funkce vyhledávacího automatu, který vyhledává sekvence s Levenshteinovou vzdálesností nejvýše 3, pro vzor $P = p_1p_2p_3p_4$ [1].

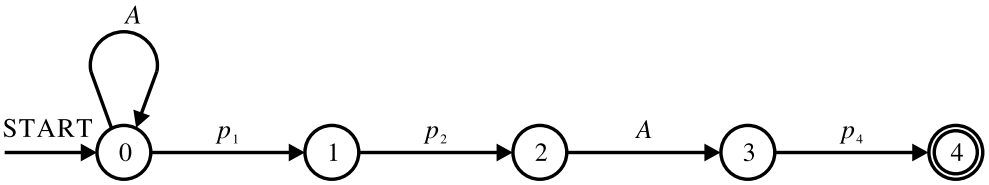


Obrázek 3.8: Přechodová funkce vyhledávacího automatu, který vyhledává sekvence se Zobecněnou Levenshteinovou vzdáleností nejvýše 3, pro vzor $P = p_1p_2p_3p_4$ [1].

3.4 Vyhledávání s „don't care“ symbolem

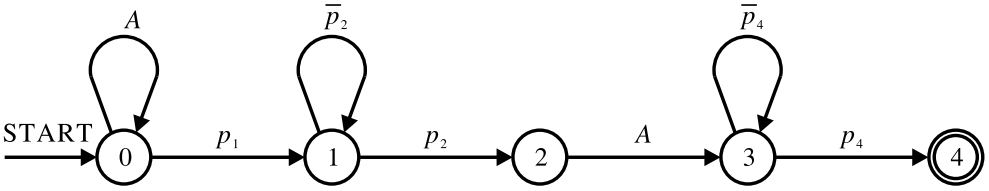
Pro vyhledávání řetězců s „don't care“ symbolem je zapotřebí provést lehkou modifikaci libovolného z popsaných algoritmů.

Na obrázku 3.9 se nachází přechodová funcke automatu, který vyhledává vzor $P = p_1 p_2 \circ p_4$. Všimněme si, že automat ve stavu 2 přechází do stavu 3 na celou abecedu. Toto je žádané chování, kterého lze docílit jednoduchou modifikací algoritmu 2. Stačí pouze detekovat, zda-li se zpracovávaný symbol rovná „don't care“ symbolu a pokud ano, vložit přechody pro všechny symboly z dané abecedy. Z důvodu jednoduchosti neuvádím formální zápis této úpravy.



Obrázek 3.9: Přechodová funkce pro vyhledání vzoru $P = p_1 p_2 \circ p_4$ [1].

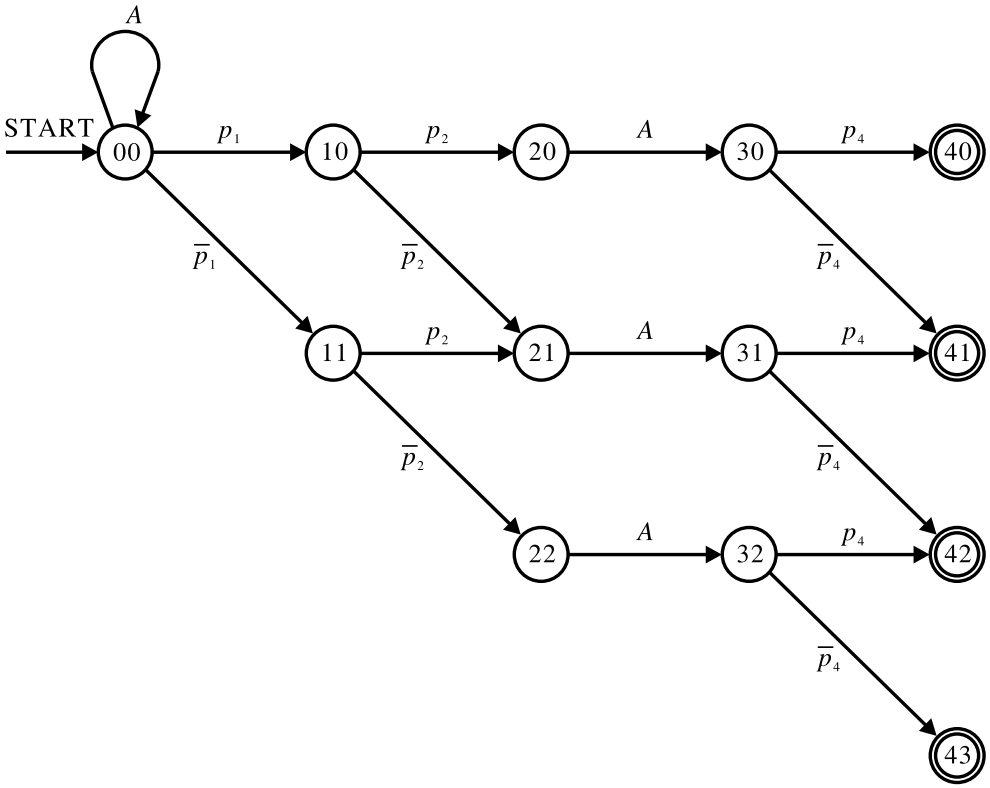
Podobným způsobem se postupuje i v případě vyhledávání sekvencí. Na obrázku 3.10 se nachází přechodová funkce automatu, který vyhledává sekvenci $P = p_1 p_2 \circ p_4$. Všimněme si, že ve stavu 2 se nenachází smyčka, která by zajistila přijetí symbolu, který se nenachází ve vzoru. Tak ale přesně chceme, aby se daný automat choval. Opět vidíme, že tato modifikace chování je velmi triviální a proto zde není uveden formální zápis této úpravy.



Obrázek 3.10: Přechodová funkce vyhledávacího automatu, který vyhledává sekvenci $P = p_1 p_2 \circ p_4$ [1].

Obdobným způsobem se postupuje i v případě přibližného vyhledávání. Na obrázku 3.11 se nachází ukázka takového automatu. Zde je navíc zapotřebí vzít v úvahu ještě jednu změnu. „Diagonální“ přechody stejného typu, jako je přechod ze stavu 20 do stavu 31, budou přecházet na prázdnou množinu symbolů a mohou tedy být odstraněny. Opět se jedná o triviální změnu a proto není uveden formální zápis této úpravy.

Ve zbytku probíraných automatů již nedochází k žádným netriviálním změnám, které by předchozí odstavce nepokrývaly.



Obrázek 3.11: Přechodová funkce vyhledávacího automatu, který vyhledává vzor $P = p_1 p_2 \circ p_4$ s Hammingovou vzdáleností nejvýše 3 [1].

Kapitola 4

Simulace vyhledávacích automatů

V této kapitole je probrána teorie, na základě které je následně naimplementována simulační část této bakalářské práce.

První sekce obecně rozebere problematiku simulací vyhledávacích automatů. V další sekci je probrána metoda simulace pomocí dynamického programování. V poslední sekci je prodiskutována metoda simulace pomocí bitového paralelismu.

4.1 Úvod do problematiky

Při praktické aplikaci vyhledávání v textu, není příliš vhodné používat konečné automaty, jaké jsou v této práci konstruovány. Důvod je takový, že tyto automaty mohou při vykonávání své práce mít obrovské nároky na prostor [1]. Z tohoto důvodu vznikly metody, jak tyto automaty pouze simulovat a dosáhnout tak v praxi přijatelné složitosti. Mezi tyto metody se řadí:

1. použití funkce selhání
2. dynamické programování
3. bitový paralelismus

Funkce selhání je metoda, která používá speciální druh konečných automatů, které používají minimální množství smyček, s cílem, aby zůstaly deterministické. Dále pak používají zpětné přechody, během kterých není čten žádný vstup, které jsou použity v případě, že nelze použít žádný jiný přechod. Příkladem tohoto druhu automatu je například automat vznikající aplikací metody Knuth–Moris–Pratt (KMP). [1]

Po dohodě s vedoucím práce se touto metodou práce nezabývá. Důvodem je, že tato skupina algoritmů se již v Knihovně algoritmů nachází.

Zbylé z metod jsou popsány v následujících sekcích.

4.2 Dynamické programování

Dynamické programování je metoda, kterou lze aplikovat na velké množství problémů. Jedním z těchto problémů je právě vyhledávání vzoru v řetězci.

Algoritmy z této sekce jsou převzány z [1].

Mějme vzor $P = p_1p_2...p_m$ a řetězec $T = t_1t_2...t_n$. Algoritmus dále používá matici D o velikosti $(m+1) \times (n+1)$. Každý z prvků $d_{i,j}, 0 \leq j \leq m, 0 \leq i \leq n$ obsahuje aktuální vzdálenost vzoru od řetězce končícího na i-té pozici. Algoritmy pro přibližné vyhledávání pak na vstupu ještě potřebují parametr k, určující nejvyšší přijatelnou vzdálenost vyhledávaného vzoru.

Algoritmus funguje tak, že na základě pravidel (která jsou definována pro každý z problémů v následujících sekcích), vypočítává sloupec po sloupci hodnoty v matici D. Po dokončení této procedury pak jen stačí správně nahlédnout do výsledné matice a z ní je možné zjistit, na kterých indexech byl hledaný vzor nalezen. Dané místo poznáme podle toho, že hodnota daného prvku matice D, $d_{j,i}$ je menší nebo rovna k. Zároveň jeho hodnota určuje nejmenší možnou vzdálenost vyhledávaného vzoru od nalezené pozice v řetězci.

4.2.1 Přesné vyhledávání

Algorimus pro přesné vyhledání je stejný, jako pro přibližné vyhledání pomocí Hammingovy vzdálenosti. Jediný rozdíl je, že parametr k se nastaví na hodnotu 0.

4.2.2 Přibližné vyhledávání pomocí Hammingovy vzdálenosti

V tomto algoritmu se matice D vypočítá následujícím způsobem.

$$\begin{array}{lcll} d_{j,0} & := & k+1, & 0 \leq j \leq m \\ d_{0,i} & := & 0, & 0 \leq i \leq n \\ d_{j,i} & := & \text{if } t_{i-1} = p_{j-1} \text{ then } d_{j-1,i-1}, \text{ else } d_{j-1,i-1}+1 & 0 < i \leq n, \\ & & & 0 < j \leq m \end{array} \tag{4.1}$$

V předpisu 4.1 výraz $d_{j-1,i-1}$ reprezentuje přesné vyhledání - pozice i v řetězci T je zvýšena, pozice j ve vzoru P je zvýšena a vzdálenost řětězce zůstává stejná. Výraz $d_{j-1,i-1}+1$ reprezentuje operaci nahrazení - pozice i v textu T je zvýšena, pozice j ve vzoru P je zvýšena a vzdálenost řetězce je zvýšena o 1. Hodnota $d_{0,i}, 0 \leq i \leq n$ je nastavena na 0, protože Hammingova vzdálenost mezi dvěma prázdnými řetězci je rovna 0. Hodnota $d_{j,0}, 0 \leq j \leq m$ je nastavena na $k+1$. Toto zaručuje, že všechny $j \leq i$ překračují maximální přijatelnou hodnotu vzdálenosti.

4.2.3 Přibližné vyhledávání pomocí Levenshteinovy vzdálenosti

Pro tento algoritmus dochází k jedné zásadní změně. Díky tomu, že Levenshteinova vzdálenost umožňuje vkládání nových a mazání současných znaků ze vzoru, není v této metodě možné zjistit, kde nalezený vzor začíná a můžeme pouze říci, kde nalezený vzor končí.

Prvky matice D vypočítáme následujícím způsobem:

$$\begin{array}{lcll} d_{j,0} & := & j, & 0 \leq j \leq m \\ d_{0,i} & := 0 & 0 \leq i \leq n & \\ d_{j,i} & := \quad \min(& \text{if } t_{i-1} = p_{j-1} \text{ then } d_{j-1,i-1}, & \\ & & \text{else } d_{j-1,i-1} + 1, & \\ & & \text{if } j < m \text{ then } d_{j,i-1} + 1, & \\ & & d_{j-1,i} + 1), & 0 < i \leq n, \\ & & & 0 < j \leq m \end{array} \tag{4.2}$$

V předpisu 4.2 výraz $d_{j-1,i-1}$ reprezentuje přesné vyhledávání. Výraz $d_{j-1,i-1} + 1$ reprezentuje operaci nahrazení. Výraz $d_{j,i-1} + 1$ reprezentuje operaci vložení - pozice i v řetězci T je zvýšena, pozice j ve vzoru P zůstává stejná a celková vzdálenost je o 1 zvýšena. Výraz $d_{j-1,i}+1$ reprezentuje operaci smazání - pozice i v řetězci T zůstává stejná, pozice j ve vzoru P se zvyšuje a celková vzdálenost je o 1 vyšší.

4.2.4 Přibližné vyhledávání pomocí Zobecněné Levenshteinovy vzdálenosti

Pro tuto metodu rozšíříme formuli 4.2 tak, abychom přidali možnost prohození sousedících symbolů. Výsledné rozšíření matice D vypadá následovně:

$$\begin{array}{lcll} d_{j,0} & := & j, & 0 \leq j \leq m \\ d_{0,i} & := & 0 \leq i \leq n & \\ d_{j,i} & := \quad \min(& \text{if } t_{i-1} = p_{j-1} \text{ then } d_{j-1,i-1}, & \\ & & \text{else } d_{j-1,i-1} + 1, & \\ & & \text{if } j < m \text{ then } d_{j,i-1} + 1, & \\ & & d_{j-1,i} + 1, & \\ & & \text{if } i > 1 \text{ and } j > 1 & \\ & & \text{and } t_{i-2} = p_{j-1} \text{ and } t_{i-1} = p_{j-2} & \\ & & \text{then } d_{j-2,i-2} + 1), & 0 < i \leq n, \\ & & & 0 < j \leq m \end{array} \tag{4.3}$$

V předpisu 4.3 výraz $d_{j-1,i-1}$ reprezentuje přesné vyhledávání. Výraz $d_{j-1,i-1} + 1$ reprezentuje operaci nahrazení. Výraz $d_{j,i-1} + 1$ reprezentuje operaci vložení. Výraz $d_{j-1,i} + 1$ reprezentuje operaci smazání. Výraz

$d_{j-2,i-2} + 1$ reprezentuje operaci prohození sousedících symbolů - pozice i v řetězci T je zvýšena o 2, pozice j ve vzoru P je zvýšena o 2 a celková vzdálenost je zvýšena o 1.

4.3 Bitový paralelismus

Tato metoda čerpá z výhod seskupení bitů ve vektoru a použití paralelních bitových operací. Jedná se tudíž o metodu vhodnou pro implementaci přímo nad hardware [24]. Na základě toho, jaké operace tyto algoritmy používají, je lze rozdělit do tří skupin:

1. Shift-Or algoritmy,
2. Shift-And algoritmy,
3. Shift-Add algoritmy.

Tato práce se ale zabývá pouze Shitf-Or algoritmy. Zdrojem těchto algoritmů je [1].

Tyto algoritmy pro svou práci potřebují dvě věci. Vzor $P = p_1p_2...p_m$ a řetězec $T = t_1t_2...t_n$. Algoritmy pro přibližné vyhledávání pak ještě potřebují parametr k určující maximální přípustný počet chyb.

Samotný algoritmus pak lze rozdělit na dvě části.

V první části probíhá příprava. V rámci té se vytvoří matice D velikosti $m \times |\Sigma|$, kde m je velikost vzoru. Každý prvek této matice $d_{j,x}, 0 < j \leq m, x \in \Sigma$, obsahuje 0, právě tehdy když $p_j = x$. V opačném případě obsahuje hodnotu 1.

Druhá část tohoto druhu algoritmů je již různá pro každý z problémů. Všechny z algoritmů však obsahují matici $R^l, 0 < l \leq$ k velikosti $m \times (n + 1)$.

Obsah této matice je ale pro každou z nich odlišný a je tedy popsán v následujících kapitolách.

4.3.1 Přesné vyhledávání

Pro přesné vyhledání použijeme Shift-Or algoritmus. Tento algoritmus potřebuje pouze matici R^0. Její hodnoty vypočítáme následovně:

$$\begin{array}{rcll} r^0_{j,0} & := & 1, & 0 < j \leq m \\ R^0_i & := & \text{shl}(R^0_{i-1}) \text{ OR } D[t_i], & 0 < i \leq n \end{array} \tag{4.4}$$

V druhé formuli shl() značí operaci bitový posun vlevo, která vloží 0 na začátek bitového vektoru. OR značí bitovou operaci binárního sčítání.

Výraz $\text{shl}(R^0_{i-1})$ OR $D[t_i]$ reprezentuje operaci nalezení znaku na pozici i v řetězci T.

Vzor P je nalezen na pozici $t_{i-m+1}...t_i$, pokud $r^0_{m,i} = 0,$ pro $0 < i \leq n$.

4.3.2 Vyhledávání s Hammingovou vzdáleností

Pro přibližné vyhledání s pomocí Hammingovy vzdálenosti vypočítáme vektory $R_i^l, 0 \leq l \leq k, 0 \leq i \leq n$ následujícím způsobem:

$$\begin{array}{rcll} r_{j,0}^l & := & 1, & 0 < j \leq m, 0 \leq l \leq k \\ R_i^0 & := & \text{shl}(R_{i-1}^0) \text{ OR } D[t_i], & 0 < i \leq n \\ R_i^l & := & (\text{shl}(R_{i-1}^l) \text{ OR } D[t_i]) \text{ AND shl}(R_{i-1}^{l-1}), & 0 < i \leq n, 0 < l \leq k \end{array} \tag{4.5}$$

Ve třetí formuli operace AND značí binární násobení.

Výraz $\text{shl}(R_{i-1}^0)$ OR $D[t_i]$ reprezentuje přesné vyhledávání daného vzoru a výraz $\text{shl}(R_{i-1}^{l-1})$ reprezentuje operaci nahrazení - pozice i v řetězci T je zvýšena, pozice ve vzoru P je zvýšena a vzdálenost l se zvyšuje.

Vzor P je nalezen s nejvýše k chybami na pozici $t_{i-m+1}...t_i$, pokud $r_{m,i}^k = 0, 0 < i \leq n$. Nejmenší počet chyb l, pro daný nález vzoru nalezneme tak, že najdeme nejmenší možné l, pro které platí $r_{m,i}^l = 0$.

4.3.3 Vyhledávání s Levenshtenovou vzáleností

Pro přibližné vyhledání s pomocí Levenshteinovy vzdálenosti vypočítáme vektory $R_i^l, 0 \leq l \leq k, 0 \leq i \leq n$ níže uvedeným způsobem. Pro zamezení vkládání znaků v koncových stavech musíme ještě navíc použít vektor V.

$$\begin{array}{rcll} r_{j,0}^l & := & 0, & 0 < j \leq m, 0 < l \leq k \\ r_{j,0}^l & := & 1, & l < j \leq m, 0 \leq l \leq k \\ R_i^0 & := & \text{shl}(R_{i-1}^0) \text{ OR } D[t_i], & 0 < i \leq n \\ R_i^l & := & (\text{shl}(R_{i-1}^l) \text{ OR } D[t_i]) & \\ & & \text{AND shl}(R_{i-1}^{l-1} \text{ AND } R_i^{l-1}) & \\ & & \text{AND } (R_{i-1}^{l-1} \text{ OR } V), & 0 < i \leq n, 0 < l \leq k \end{array} \tag{4.6}$$

Vektor V sestrojíme následovně:

$$V = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_m \end{bmatrix}, \text{ kde } v_m = + \text{ a } v_j = 0, \forall j, 1 \leq j < m.$$

Výraz $\text{shl}(R_{i-1}^0)$ OR $D[t_i]$ reprezentuje přesné vyhledávání daného vzoru a výraz $\text{shl}(R_{i-1}^{l-1})$ reprezentuje operaci nahrazení. Výraz $\text{shl}(R_i^{l-1})$ reprezentuje operaci smazání - pozice ve vzoru P je zvýšena, pozice v řetězci T je zvýšena a vzdálenost l je zvýšena. Výraz R_{i-1}^{l-1} reprezentuje operaci vložení. Matice V pak zamezuje tomu, aby se operace vložení neprovedla nad žádným z koncových stavů.

Vzor P je nalezen se vzdáleností nejvýše k a končí na pozici i, pokud $r^k_{m,i} = 0, 0 < i \leq n$. Maximální počet chyb zjistíme tak, že nalezneme nejmenší l takové, pro které platí, že $r^l_{m,i} = 0$.

4.3.4 Vyhledávání se Zobecněnou Levenshteinovou vzdáleností

Pro vyhledávání se Zobecněnou Levenshteinovou vzdáleností je nutné rozšířit metodu z předchozí sekce. Rozšíříme ji přidáním vektorů $S^l_i, 0 \leq l < k, 0 \leq i < n$, které budou ve tvaru:

$$S^l_i = \begin{bmatrix} s^l_{1,i} \\ s^l_{2,i} \\ \vdots \\ s^l_{m,i} \end{bmatrix}, 0 \leq l < k, 0 \leq i < n$$

Vektory $R^l_i, 0 \leq l \leq k, 0 \leq i \leq n$ a $S^l_i, 0 \leq l \leq k, 0 \leq i \leq n$ pak vypočítáme následujícím způsobem:

$$\begin{array}{rcll} r^l_{j,0} & := & 0, & 0 < j \leq m, \\ & & & 0 < l \leq k \\ r^l_{j,0} & := & 1, & l < j \leq m, \\ & & & 0 \leq l \leq k \\ R^0_i & := & \text{shl}(R^0_{i-1}) \text{ OR } D[t_i], & 0 < i \leq n \\ R^l_i & := & (\text{ shl}(R^l_{i-1}) \text{ OR } D[t_i]) & \\ & & \text{AND shl}(R^{l-1}_{i-1} \text{ AND } R^{l-1}_i \text{ AND } (S^{l-1}_{i-1} \text{ OR } D[t_i])) & \\ & & \text{AND } (R^{l-1}_{i-1} \text{ OR } V), & 0 < i \leq n, \\ & & & 0 < l \leq k \\ s^l_{j,0} & := & 1 & 0 < j \leq m, \\ & & & 0 \leq l < k \\ S^l_i & := & \text{shl}(R^l_{i-1}) \text{ OR shr}(D[t_i]), & 0 < i < n, \\ & & & 0 \leq l < k \end{array} \tag{4.7}$$

V poslední formuli značí výraz shr(...) operaci bitového posunu vpravo.

Výraz $\text{shl}(R^0_{i-1})$ OR $D[t_i]$ reprezentuje přesné vyhledávání daného vzoru, výraz $\text{shl}(R^{l-1}_{i-1})$ reprezentuje operaci nahrazení, výraz $\text{shl}(R^{l-1}_i)$ reprezentuje operaci smazání a výraz R^{l-1}_{i-1} reprezentuje operaci vložení.

Výraz $(S^{l-1}_{i-1}$ OR $D[t_i])$ slouží k implementaci operace prohození sousedních symbolů - pozice ve vzoru je posunuta o 2, stejně jako pozice v řetězci, nicméně vzdálenost je pousnuta pouze o 1. Toto zvýšení je zajištěno použitím vektoru S^l_i. Vzor P je nalezen s nejvýše k chybami a končí na indexu i, pokud $r^k_{m,i} = 0, 0 < i \leq n$. Maximální počet chyb je zjištěn tak, že nalezneme nejmenší l takové, pro které platí, že $r^l_{m,i} = 0$.

Kapitola 5

Implementace a testování

V této kapitole je nejprve provedena stručná analýza současného stavu řešení v Algoritmové knihovně. Následně je probrána zhotovená implementace vyhledávacích automatů. Poté je prodiskutována implementace simulací vyhledávacích automatů. Na závěr jsou popsány metody testování.

5.1 Analýza současného stavu Algoritmové knihovny

5.1.1 Vyhledávávací automaty

Z vyhledávacích automatů byl v knihovně naimplementován jednoduchý vyhledávací automat z algoritmu 2. Ostatní algoritmy bylo nutné vytvořit.

5.1.2 Simulace vyhledávacích automatů

V Algoritmové knihovně již byla vytvořena nejzákladnější metoda, tedy přímá simulace běhu konečného automatu. Dále pak v knihovně jsou v dostatečném množství i kvalitě naimplementovány metody simulace pomocí fuknce selhání a proto se jimi tato práce nezabývala. Metody dynamického programování a bitového paralelismu bylo nutné naimplementovat.

5.2 Vyhledávání

Nutnou podmínkou pro implementaci algoritmů, které konstruují konečné automaty, je mít připravenou vhodnou vnitřní reprezentaci. V této práci jsem se tímto problémem nemusel zabývat, protože v Algoritmové knihovně již taková implementace existuje. Jedná se o třídy `automaton::NFA`, `automaton::DFA` a `automaton::EpsilonNFA` z modulu *alib2data*. Další nutností bylo mít možnost převádět výsledek z vnitřní reprezentace do nějakého čitelného formátu.

K tomu jsem použil nástroj `convert::DotConverter` z modulu *alib2aux*, který zajišťuje převod z vnitřního formátu do formátu Dot.

V práci jsem se snažil klást důraz na znovupoužitelnost již napsaného kódu, protože většina algoritmů je pouhým rozšířením nějakého jiného algoritmu. Tento krok si vyžádal některé ústupky. Algoritmus 4 vyžaduje v posledním kroku provést operaci odstranění nedostupných stavů. Důvodem je, že algoritmus vytvoří k kopií automatu pro přesné vyhledávní a následně do něj přidá přechody, které zabezpečí přijetí s chybou. Pokud nejsou odstraněny, automat bude stále perfektně fungovat, jen nebude tak hezký. Po jejich odstranění ale mohou při znovupoužití nastat problémy.

Jako příklad si vezmeme libovolný řetězec s abecedou velikosti 1. Vyhledávací automat, který využívá Hammingovu vzdálenost větší než 1 pak bude vypadat úplně stejně, jako automat pro přesné vyhledávání. Problém nastane, pokud tento automat budeme chtít rozšířit, například postavením automatu, který využívá Levenshteinovu vzdálenost. Takový algoritmus pak bude předpokládat existenci paralelních vyhledávacích automatů pro přesné vyhledání, které ale díky odstranění nedostažitelných stavů nebudou existovat. Algoritmus by na tento fakt musel být připraven a jeho složitost by tak značně narostla. Z tohoto důvodu jsem se po konzultaci s vedoucím práce rozhodl tento krok z algoritmů vypustit a značně je tím zjednodušit a hlavně zpřehlednit.

5.2.1 Konstrukce automatu používající Hammingovu vzdálenost

Jedná se o implementaci algoritmu 4.

V první fázi algoritmus vytvoří $k + 1$ automatů pro přesné vyhledávání. Stavy jsou pojmenovávány systematicky, tak, aby bylo možné je snadno využít i při stavbě dalších automatů, založených na tomto vyhledávacím automatu. Systém pojmenování je jednoduchý, využil jsem dvojici čísel, kde první z dvojice určuje pořadí stavu v jednoduchém vyhledávacím automatu a druhý určuje aktuální úroveň.

Další fáze přidává chybové přechody, přesně jak je popsáno v 4.

Poslední fáze algoritmu je ale záměrně vynechána, jak jsem již popsal o několik odstavců výše.

5.2.2 Konstrukce automatu používající Levenshteinovu vzdálenost

Tento algoritmus je rozšířením algoritmu pro konstrukci automatu používající Hammingovu vzdálenost. Implementuji ho tedy tím, že nejprve postavím Hammingův automat. Následně přidám přechody pro odstranění starých a vložení nových symbolů, přesně jako je uvedeno v 5.

5.2.3 Konstrukce automatu používající Zobecněnou Levenshteinovu vzdálenost

Tento algoritmus je rozšířením předchozího algoritmu. Algoritmus tedy začne konstrukcí Levenshteinova vyhledávacího automatu.

Následně algoritmus 6 vyžaduje vytvoření nových stavů, které slouží pro uchování informace během prohození symbolů. Zachoval jsem podobný systém pojmenování stavů, jaký jsem vytvořil v algoritmu pro vytvoření automatu, který používá Hammingovu vzdálenost. První z dvojice reprezentuje pořadí v jednoduchém vyhledávacím automatu. Aby nedošlo ke kolizi názvů, je toto číslo zvýšeno o velikost vzoru + 1. Druhý z dvojice je pak číslo úrovně, ze které se do daného stavu lze dostat.

Poslední operací je přidání požadovaných přechodů, které připojí právě vytvořené stavy.

5.2.4 Modifikace pro vyhledávání sekvencí

Ačkoliv algoritmus je univerzální a stačilo by jej naimplementovat pouze jednou, rozhodl jsem se kvůli přehlednosti vždy implementovat zvlášť pro každou z metod vyhledávání. Algoritmus je implementován přesně, jako je popsáno v 7.

5.2.5 Modifikace pro vyhledávání řetězců a sekvencí s „don't care“ symbolem

Tato modifikace již byla o něco komplexnější. Nejprve bylo zapotřebí vytvořit obecnou reprezentaci symbolu „don't care“. Ta již v Algoritmové knihovně byla vytvořena pro využití stromy. Stačilo ji tedy jen vhodně přejmenovat, aby použití dané třídy dávalo smysl. Jednalo se o třídu `alphabet::SubtreeWildcardSymbol` v modulu *alib2data*, kterou jsem přejmenoval na `alphabet::WildcardSymbol`.

Další potřebnou změnou bylo vytvoření reprezentace řetězce, který obsahuje symbol „don't care“. Do modulu *alib2data* jsem tedy přidal třídu `string::WildcardLinearString`, která po vzoru třídy `string::LinearString` obsahuje všechny náležitosti, jaké Algoritmová knihovna požaduje.

Dále jsem vytvořil specializace všech předchozích algoritmů, kterými se tato část práce zabývala. Změny, které bylo nutné provést, jsou popsány v sekci 3.4.

5.3 Simulace

Pro implementaci simulací není nutně zapotřebí žádná z funkcí implementovaných v Algoritmové knihovně. Přesto jsem po jejím vzoru použil rozšíření

standartních kontejnerů z jmenného prostoru `ext`. K odůvodnění tohoto kroku se dostanu později.

5.3.1 Simulace pomocí dynamického programování

V návrhu jsem přistoupil k rozdělení algoritmů do dvou procesů - vypočtení tabulky a vyhledání výskytů ve vypočtené tabulce. Důvodem k tomuto kroku byl fakt, že implementace se měla zaměřit na přehlednost a né na efektivitu.

Z přechozí kapitoly vyplývá, že algoritmus výpočtu tabulky je možné rozdělit do dvou částí. V první části se inicializuje tabulka, nad kterou se daný algoritmus provádí. V druhé pak dochází k samotnému výpočtu.

U všech implementovaných simulací jsem pak využil stejné kostry algoritmu. V té nejdříve vytvořím matici T rozměrů $(m + 1) \times (n + 1)$, kde m je velikost vzoru a n je velikost textu. Následně dochází k předpřípravě matice T. Každý z algoritmů dynamického programování potřebuje, aby některé hodnoty v tabulce byly přednastaveny. Bližší detaily jsou popsány v sekci 4.2. Následuje dvojitý cyklus, ve kterém postupně procházím tabulku, nejprve podle velikosti řetězce a následně podle velikosti vzoru. Uvnitř cyklu dochází k samotnému vyhledávání, kdy se vyplňují hodnoty v matici T. Detaily jsou opět popsány v sekci 4.2.

Implementoval jsem algoritmy přibližného vyhledání s následujícími vzdálenostmi:

- Hammingova vzdálenost, viz sekce 4.1,
- Levenshteinova vzdálenost, viz sekce 4.2,
- Zobecněná Levenshteinova vzdálenost, viz sekce 4.3.

Neimplementoval jsem simulaci pro přesné vyhledávání, protože její algoritmus je úplně stejný, jako když se k pro simulaci používající Hammingovu vzdálenost nastaví na 0.

5.3.2 Simulace pomocí bitového paralelismu

Nyní je vhodná chvíle odůvodnit použití kontejnerů z jmenného prostoru `ext`. Důvodem k tomuto kroku byl fakt, že v knihovně jsou implementované binární operátory pro operace nad `ext::vector<bool>`, které byly u implementace bitového paralelismu zapotřebí.

Ačkoliv rešerše této skupiny algoritmů nabádá k tomu, aby se při implementaci pracovalo s maticemi, rozhodl jsem se v souladu s [24] pracovat pouze s vektory. Proto není možné získat jako výstup tabulku, ale pouze indexy v řetězci. Díky tomu ale získáme o něco lepší paměťovou efektivitu.

Všechny algoritmy potřebují vypočítat matici D. Z toto důvodu jsem vytvořil třídu `stringology::simulations::BitParalelism`, která obsahuje statickou metodu `constructDVectors`, která se stará o její vytvoření.

Algoritmus lze opět rozdělit na dvě části.

V první dochází k přípravě. V této fázi algoritmus vytvoří matici D za pomocí `BitParalelism::constructDVectors`. Dále pak dojde k počáteční inicializaci všech dalších pomocných vektorů, jakými jsou vektor R, S a nebo V.

V druhé fázi dochází k samotnému výpočtu. Ten probíhá ve smyčce, jejíž délka odpovídá velikosti řetězce, ve kterém se provádí vyhledávání. Na začátku smyčky se vždy uloží stav vektorů R (a případně i S), protože algoritmy vyžadují rekurentní přístup ke stavu v předchozí iteraci. Tento krok by mohl být vynechán, protože algoritmus je možné provést in-place [24]. Od tohoto kroku jsem se ale rozhodl upustit, protože výsledný kód byl vysoce nepřehledný. Na závěr této smyčky dojde k samotnému výpočtu (vyhledávání) a upraví se hodnoty všech námi používaných vektorů.

Naimplementoval jsem simulace následujících vzdáleností:

- Hammingova vzdálenost, viz sekce 4.5,
- Levenshteinova vzdálenost, viz sekce 4.6,
- Zobecněná Levenshteinova vzdálenost, viz sekce 4.7.

Dále jsem pak naimplementoval simulaci pro přesné vyhledání vzoru, viz sekce 4.4.

5.4 Testování

Testování mnou vytvořených funkcionalit lze rozdělit do dvou skupin problémů. Prvním z nich jsou jednotkové testy, druhým pak testy náhodné.

5.4.1 Jednotkové testy

Každý z problémů, který jsem řešil, má přiřazený jednotkový test. Tyto testy byly připraveny na základě ukázek ze zdroje [1].

Pro konstrukci automatů se jedná o prosté konstrukce na základě předdefinovaných problémů. Automat vždy v testu zkonstruuji manuálně a ten porovnám s výstupem z algoritmu. Algoritmy, které konstruuji manuálně, ale neobsahují nedosažitelné stavy a tak před samotným porovnáním dojde k tomu, že nad výstupem z algoritmu zavolám proceduru `automaton::simplify::UnreachableStatesRemover::remove`, která odstraní nedostupné stavy.

U simulací metodou dynamického programování testuji odděleně dva problémy.

Prvním je samotná konstrukce tabulky. Zdroj [1] obsahoval jako příklady vypočtené tabulky, které jsem převzal a na základě nich jsem sestrojil jednotkové testy.

Druhým je vyhledání výskytů vzoru v předvypočtené tabulce. I v tomto případě jsem přistoupil k převzetí dat z [1] a na základě nich jsem setrojil jednotkové testy.

Pro simulace metodou bitového paralelismu jsem postupoval obdobně, jako u metody dynamickým programováním. Protože jsem se ale rozhodl nepracovat s maticemi, ale pouze s vektory, bylo by porovnávání celých tabulek obtížně proveditelné. Proto testuji pouze výstup ze simulace, tedy sadu nalezených vzorů, a pro důkladnější otestování se opírám o náhodné testy. Data pro testy jsem opět převzal z [1].

5.4.2 Náhodné testy

Algoritmová knihovna obsahuje shell skript, který náhodně generuje data, na základě kterých jsou testovány některé z algoritmů ze jmenného prostoru `stringology`. Pro otestování mých algoritmů jsem se rozhodl tento skript rozšířit.

Pro náhodné testy jsem zvolil přístup takový, že z trojice vyhledávací automat, simulace metodou dynamického programování a simulace metodou bitového paralelismu, vyberu vždy dvojici, která dostane na vstupu stejný vzor a řetězec. V dalším kroku pak zkontroluji, zda-li se jejich výstup rovná. Bohužel netestuji, zda-li se rovnají pozice výskytů, ale testuji pouze jejich počet. Důvod k tomu je, že by bylo zapotřebí vytvořit nástroj, který převede výstup ze standartní simulace konečného automatu, na sadu indexů, kde se nalezený vzor vyskytuje. Díky existenci jednotkových testů, které se ale právě na tuto funkcionalitu zaměřují, jsem usoudil, že se nejedná o žádný velký problém.

Závěr

Cílem této práce bylo nastudovat projekt Algoritmová knihovna a následně jej rozšířit o algoritmy, které konstruují automaty přesného a přibližného vyhledávání. Dalším cílem bylo vytvořit chybějící metody simulací tohoto druhu automatů a vše následně otestovat.

V práci se mi podařilo uspět a naimplementovat tři druhy vyhledávacích automatů, které používají Hammingovu, Levenshteinovu a Zobecněnou Levenshteinovu vzdálenost. Nad rámec zadání jsem přidal možnost vytvářet automaty pro vyhledávání sekvencí a dále i možnost vyhledávat řetězce, které obsahují „don't care“ symbol.

Do knihovny jsem dále přidal dva druhy simulací, a to za pomocí dynamického programování a bitového paralelismu. Oba druhy simulací podporují jak přesné, tak přibližné vyhledávání a to s Hammingovou, Levenshteinovou a i Zobecněnou Levenshteinovou vzdáleností.

Vše jsem na závěr otestoval pomocí jednotkových a náhodně generovaných testů.

V budoucnu je možné na základě této práce doimplementovat algoritmy konstrukce automatů pro přibližné vyhledávání s Γ a Δ vzdáleností. Dále pak chybí konstrukce automatů, které vyhledávají regulární výrazy.

Literatura

[1] Melichar, B.; Holub, J.; Polcar, T.: *Text searching algorithms volume I: Forward string matching*. Czech Technical Univerisity in Prague, Faculty of Electrical Engineering, Department of Computer Science and Engineering, 2005, [cit. 2018-04-19]. Dostupné z: `http://stringology.org/athens/TextSearchingAlgorithms/`

[2] Hopcroft, J.; Motwani, R.; Ullman, J.: *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley series in computer science, Addison-Wesley, 2001, ISBN 9780201441246.

[3] Sipser, M.: *Introduction to the Theory of Computation*. Cengage Learning, 2012, ISBN 9781133187790.

[4] Melichar, B.; Holub, J.; Mužátko, P.: *Languages and translations*. Vydavatelství ČVUT, 1997, ISBN 8001016927.

[5] Melichar, B.: *Jazyky a překlady*. Vydavatelství ČVUT, 2003, ISBN 8001027767.

[6] Crochemore, M.; Hancart, C.; Lecroq, T.: *Algorithms on Strings*. Cambridge University Press, 2007, ISBN 9781139463850.

[7] Žák, M.: *Automatová knihovna – vnitřní a komunikační formát*. Bakalářská práce, České vysoké učení technické v Praze, Fakulta informačních technologií, Praha, 2014.

[8] Pecka, T.: *Automatová knihovna – převody mezi regulárními výrazy, regulárními gramatikami a konečnými automaty*. Bakalářská práce, České vysoké učení technické v Praze, Fakulta informačních technologií, Praha, 2014.

[9] Veselý, J.: *Automatová knihovna – determinizace konečných a zásobníkových automatů*. Bakalářská práce, České vysoké učení technické v Praze, Fakulta informačních technologií, Praha, 2014.

[10] Plachý Š.: *Automatová knihovna - Stromové automaty a algoritmy nad stromy.* Bakalářská práce, České vysoké učení technické v Praze, Fakulta informačních technologií, Praha, 2015.

[11] Brož, J.: *Automatová knihovna - Grafy a grafové algoritmy.* Bakalářská práce, České vysoké učení technické v Praze, Fakulta informačních technologií, Praha, 2016.

[12] Rosca, D.: *Automatová knihovna - isomorfismus planárních grafů.* Bakalářská práce, České vysoké učení technické v Praze, Fakulta informačních technologií, Praha, 2016.

[13] Kočička, M.: *Automata library – LR parser construction.* Bachelor's thesis, Czech Technical University in Prague, Faculty of Information Technology, Prague, 2016.

[14] Pecka, T.: *Construction of a Pushdown Automaton Accepting a Language Given by Regular Tree Expression.* Master's thesis, Czech Technical University in Prague, Faculty of Information Technology, Prague, 2016.

[15] Obůrka, R.: *Dead zone tree pattern matching in trees.* Master's thesis, Czech Technical University in Prague, Faculty of Information Technology, Prague, 2016.

[16] Parma, J.: *Automatová knihovna - komprese dat.* Bakalářská práce, České vysoké učení technické v Praze, Fakulta informačních technologií, Praha, 2017.

[17] Mareš, V.: *GUI k automatové knihovně ALIB.* Bakalářská práce, České vysoké učení technické v Praze, Fakulta informačních technologií, Praha, 2017.

[18] Shatrovskii, A.: *Implementation of a Repetition Searching Algorithm in Trees.* Bachelor's thesis, Czech Technical University in Prague, Faculty of Information Technology, Prague, 2017.

[19] Plachý Š.: *Minimalizace stromových a zásobníkových automatů.* Magisterská práce, České vysoké učení technické v Praze, Fakulta informačních technologií, Praha, 2017.

[20] Neznámý: klawson88/LevenshteinAutomaton. [cit. 2018-04-02]. Dostupné z: `https://github.com/klawson88/LevenshteinAutomaton`

[21] Ayad, L. A.; Pissis, S. P.; Retha, A.: libFLASM: a software library for fixed-length approximate string matching. BMC Bioinformatics, vol. 17, no. 1, 2016, pp. 454. [cit. 2018-04-02]. Dostupné z: `https://github.com/webmasterar/libFLASM`

[22] byteseek. [cit. 2018-04-03]. Dostupné z: `https://github.com/nishihatapalmer/byteseek`

[23] Møller, A.: dk.brics.automaton – Finite-State Automata and Regular Expressions for Java. [cit. 2014-04-03]. Dostupné z: `http://www.brics.dk/automaton/`

[24] Gupta, S.; Rasool, A.: Bit Parallel String Matching Algorithms: A Survey. *International Journal of Computer Applications*, ročník 95, č. 10, 6 2014.

Příloha **A**

Seznam použitých zkratek

DKA deterministický konečný automat

NKA nedeterministický konečný automat

FIT ČVUT Fakulta informačních technologií, České vysoké učení technické v Praze

GUI graphical user interface - grafické uživatelské rozhraní

CLI command line interface - rozhraní příkazové řádky

Příloha **B**

Obsah přiložené SD karty

- `README.txt` ..... stručný popis obsahu SD karty
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF