Sem vložte zadání Vaší práce.

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra Teoretické informatiky

Bakalářská práce

Experimenty s algoritmy na vyhledávání jader v řetězcích

Martin Jůna

Vedoucí práce: Ing. Ondřej Guth, Ph.D.

10. května 2015

Poděkování

Rád bych poděkoval panu Ing. Ondřeji Guthovi, Ph.D., za odborné vedení, četné konzultace a cenné rady. Dále bych chtěl poděkovat své rodině za morální pomoc a neutuchající podporování.

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 10. května 2015 .....

Odkaz na tuto práci

Jůna, Martin. *Experimenty s algoritmy na vyhledávání jader v řetězcích.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2015.

Abstrakt

V této práci jsem se zaměřil na dva algoritmy hledající vlastní přibližná jádra řetězců. Na řešení tohoto problému existují dva odlišné algoritmy. Za účelem jejich srovnání jsem provedl sadu experimentů s těmito algoritmy. Experimenty byly zaměřeny na vliv jednotlivých parametrů tohoto problému na vlastnosti těchto algoritmů, a také na jejich srovnání. Proto abych nemluvil pouze o černých skříňkách, nachází se v této práci i popis těchto algoritmů a popis problému, které řeší.

Klíčová slova Pravidelnosti v řetězcích, přibližná jádra řetězců, Hammingova vzdálenost, experimenty, experimentální porovnání

Abstract

I focused on two algorithms for finding all restricted smallest distance k-approximate seeds, in this work. Two different algorithms exist for solving the problem. I performed a lot of experiments with these algorithms to compare it. Experiments were focused on impact of all parametrs of algorithms and their comparison. In order not to talk just about black boxes, there are descriptions of algorithms and problems that solve them.

Keywords Regularities of string, aproximate seed of string, Hamming distance, experiments, experimental comparison

Obsah

Seznam obrázků

Úvod

Pravidelnosti v řetězcích, mezi které patří periody, pokrytí (cover) a také jádra (seed), mají využití v mnohých oborech lidské činnosti. Příklady takových odvětvích jsou molekulární biologie, počítačová analýza hudby a komprese dat. Tato problematika je studována po několik let a výsledkem jsou algoritmy řešící nejrůznější problémy týkající se pravidelností.

V této bakalářské práci se zaměřím na hledání jader. S jádry je spojena řada problémů, například nejzákladnějším je zjištění, zda je dané jádro opravdu jádrem daného textu. Další variantou je nalezení minimální vzdálenosti pro dané jádro, text a vzdálenostní funkci. Toto se dá dále zesložitit na hledání množiny přibližných jader daného textu při dané vzdálenostní funkci a maximální dovolené vzdálenosti. Tato množina, která je zkoušena, může být omezena na konkrétní řetězce nebo podřetězce daného textu nebo nemusí mít žádné omezení. Poslední uvedený případ, tedy hledání bez omezení, je nejobecnější problém, pro který je dokázáno, že při použití vážené vzdálenostní funkce (weighted edit distance) jde o NP-úplný problém (viz [1]).

V této práci se zabývám algoritmy řešícími problém hledání všech přibližných jader textu, které jsou podřetězci textu a při použití Hammingovy vzdálenosti jakožto vzdálenostní funkce. Na řešení tohoto problému existují dva algoritmy používající různé přístupy. Jeden postup používá dynamického programování a druhý používá konečný automat. V této práci řádně zadefinuji výše zmíněné pojmy a popíši oba algoritmy. Provedu implementaci těchto algoritmů, abych mohl experimentovat. Experimenty budou zaměřeny na ověření složitosti těchto algoritmů a jejich porovnání.

Kapitola 1

Cíl práce

V této bakalářské práci se zabývám algoritmy na hledání všech přibližných jader řetězců. Na řešení tohoto problému existují dva odlišné algoritmy. Jeden je založený na dynamickém programování a ten druhý na použití konečných automatů. Autorem algoritmu, který využívá konečné automaty, je vedoucí této práce. Cílem této práce je experimentální testování těchto vlastností:

- ověření vlastností algoritmu, který používá konečné automaty,
- porovnání náročností obou přístupů,
- prozkoumání jak závisí potřebný čas a paměť na délce vstupních dat,
- prozkoumání jak změna maximální dovolené vzdálenosti ovlivňuje časovou a paměťovou náročnost,
- prozkoumání vlivu různých datových sad na časovou a paměťovou náročnost,
- prozkoumání jak časovou a paměťovou náročnost ovlivňuje prohledávání řetězců s různými abecedami.

Kapitola 2

Definice pojmů a problémů

2.1 Řetězců

Definice 2.1.1 (Abeceda)**.** Abeceda je konečná a neprázdná množina symbolů.

Definice 2.1.2 (Řetězec)**.** Řetězec nad abecedou A je konečná posloupnost symbolů z A.

Značení 2.1.3. Symbolem $x[i]$ značíme i-tý symbol řetězce x (počítáno od jedné).

Definice 2.1.4 (Délka řetězce)**.** Délka řetězce je počet symbolů v něm obsažených. Délku řetězce w značíme $|w|$.

Definice 2.1.5 (Prázdný řetězec)**.** Prázdný řetězec ε je prázdná posloupnost symbolů a jeho délka je 0.

Definice 2.1.6 (Efektivní abeceda řetězce)**.** Efektivní abeceda A_w řetězce w nad abecedou A obsahuje pouze symboly obsažené v řetězci w.

Definice 2.1.7 (Množina všech řetězců)**.** Množina všech řetězců nad abecedou A je množina všech možných řetězců, které lze zkonstruovat ze symbolů v A a nebo prázdný řetězec. Množinu všech řetězců nad abecedou A značíme A^*.

Definice 2.1.8 (Množina všech neprázdných řetězců)**.** Množina všech neprázdných řetězců nad abecedou A je množina všech možných řetězců, které lze zkonstruovat ze symbolů v A. Množinu všech neprázdných řetězců nad abecedou A značíme A^+ a platí $A^* = A^+ \cup \{\varepsilon\}$.

Definice 2.1.9 (Formální jazyk)**.** Formální jazyk $L \subseteq A^*$ je množina řetězců nad abecedou A.

Definice 2.1.10 (Zřetězení)**.** Zřetězení dvou řetězců $v, w \in A^*$ nazýváme řetězec, který vznikl zapsáním sekvence symbolů z w za sekvenci v. Zřetězení značíme $v \cdot w$, nebo vw. Pro zřetězení platí:

- asociativita: $\forall u, v, w \in A^*$ platí $(uv)w = u(vw)$,
- nekomutativita: neplatí $\forall v, w\ vw = wv$,
- prázdný řetězec je neutrální prvek: $\forall x$ platí $x\varepsilon = \varepsilon x = x$.

Značení 2.1.11. Zápisem w^k, pro $w \in A^*$ a k celé kladné číslo, uvažujeme zřetězení k řetězců w za sebou. Pro upřesnění:

- $w^0 = \varepsilon$,
- $w^1 = w$,
- $w^k = w \cdot w^{k-1}$, pro $k \geq 2$.

Definice 2.1.12 (Překrytí)**.** Pokud máme řetězce $v = xp, w = py$ nad abecedou A, potom řetězec $z = xpy$ vznikl překrytím v a w.

Definice 2.1.13 (Podřetězec)**.** Řetězec $w \in A^*$ je podřetězec $x \in A^*$ právě tehdy, když $x = uwv$ kde $u, v \in A^*$.

Definice 2.1.14 (Konečná pozice podřetězce)**.** Uvažujme řetězce $u, w \in A^*$, kde u je podřetězec w. Potom konečnou pozicí u myslíme index v řetězci w, na kterém u končí. Tedy pro konečnou pozici i a $\forall j \in \{1, 2, \ldots, |u|\}$ platí, že $u[j] = w[i - |u| + j]$. Můžeme říci, že u okupuje i-tou pozici ve w.

Definice 2.1.15 (Množina konečných pozic)**.** Množina konečných pozic podřetězce u řetězce w je uspořádaná množina všech možných konečných pozic u v řetězci w.

Definice 2.1.16 (k-aproximovaný podřetězec)**.** Nechť $u, v, w \in A^*$ a $k \geq 0$, potom v je k-aproximovaný podřetězec w při použití vzdálenostní funkce D právě tehdy, když u je podřetězec w a platí $D(u, v) \leq k$.

Definice 2.1.17 (Konečná pozice k-aproximovaného podřetězce)**.** Uvažujme řetězce $u, v, w \in A^*$, kde u je k-aproximovaný podřetězec w při použití vzd. funkce D. Potom konečnou pozicí u je konečná pozice podřetězce v, pro který platí $D(u, v) \leq k$.

Definice 2.1.18 (Množina konečných pozic k-aproximovaného podřetězce)**.** Množina konečných pozic k-aproximovaného podřetězce u řetězce w je uspořádaná množina všech možných konečných pozic u v řetězci w.

Značení 2.1.19. Symbolem $x[i..j]$ značíme podřetězec x, který začíná i-tým a končí j-tým symbolem x. Platí $x = x[1..n]$, kde n je délka řetězce x.

Definice 2.1.20 (Levé rozšíření)**.** Nechť $l, x, w \in A^*$, potom řetězec $x = lw$ je levé rozšíření řetězce w.

Definice 2.1.21 (Pravé rozšíření)**.** Nechť $r, x, w \in A^*$, potom řetězec $x = wr$ je pravé rozšíření řetězce w.

Definice 2.1.22 (Nadřetězec)**.** Řetězec $w \in A^*$ je nadřetězec $x \in A^*$ právě tehdy, když $w = uxv$, kde $u, v \in A^*$.

Definice 2.1.23 (Reverz řetězce)**.** Reverz řetězce $w \in A^*$ je řetězec obsahující sekvenci symbolů z w zapsanou v opačném pořadí. Reverz řetězce w značíme w^R.

Definice 2.1.24 (Prefix)**.** Řetězec $p \in A^*$ je prefix řetězce $x \in A^*$ právě tehdy, když $x = pu$ kde $u \in A^*$.

Značení 2.1.25. Symbolem **pref**(w) značíme množinu všech prefixů řetězce $w \in A^*$.

Definice 2.1.26 (Sufix)**.** Řetězec $s \in A^*$ je sufix řetězce $x \in A^*$ právě tehdy, když $x = us$ kde $u \in A^*$.

Značení 2.1.27. Symbolem **suff**(w) značíme množinu všech sufixů řetězce $w \in A^*$.

Definice 2.1.28 (k-aproximovaný prefix)**.** Nechť přirozené číslo k, řetězce $v, w, u, p \in A^*$ a vzdálenostní funkce D, potom řetězec v je k-aproximovaný prefix řetězce w při použití vzdálenostní funkce D právě tehdy, když w lze psát jako pu a platí $D(p, v) \leq k$.

Definice 2.1.29 (k-aproximovaný sufix)**.** Nechť přirozené číslo k, řetězce $v, w, u, s \in A^*$ a vzdálenostní funkce D, potom řetězec v je k-aproximovaný sufix řetězce w při použití vzdálenostní funkce D právě tehdy, když w lze psát jako us a platí $D(s, v) \leq k$.

2.2 Vzdálenostní funkce

Definice 2.2.1 (Vzdálenostní funkce)**.** Vzdálenostní funkce D je zobrazení přiřazující dvěma řetězcům $x, y \in A^*$ číslo. $D : A^* \times A^* \to \mathbb{R}$. Použití značíme $D(x, y)$.

Definice 2.2.2 (Hammingova vzdálenost)**.** Hammingova vzdálenost mezi řetězci x a y je minimální počet změn symbolů v řetězci x tak, aby vznikl řetězec y. Hammingova vzdálenost je definována pouze pro řetězce stejné délky. Její hodnota je celé číslo v intervalu $\langle 0; n \rangle$, kde n je délka řetězců.

Definice 2.2.3 (Levenshteinova vzdálenost)**.** Levenshteinova vzdálenost je minimální počet změn, přidání a odebrání symbolů z řetězce x tak, aby vznikl řetězec y. Levenshteinova vzdálenost je také nazývána editační vzdálenost (edit distance).

Definice 2.2.4 (Vážená editační vzdálenost)**.** Vážená editační vzdálenost (weighted edit distance) je minimální cena změn, přidání a odebrání symbolů z řetězce x tak, aby vznikl řetězec y. Cena jednotlivých změn je dána penalizační maticí. Vážená editační vzdálenost je zobecnění Levenshteinovy vzdálenosti.

2.3 Pravidelností

Definice 2.3.1 (Perioda řetězce)**.** Řetězec r je periodou řetězce w právě tehdy, když existuje pravé rozšíření wp řetězce w, které lze zkonstruovat zřetězením kopií řetězce r, tedy $\exists k : wp = r^k$.

Definice 2.3.2 (k-aproximovaná perioda řetězce)**.** Řetězec r je k-aproximovaná perioda řetězce w při použití vzd. funkce D právě tehdy, když existuje množina $L = \{r_1, r_2, \ldots, r_{|L|}\}$ taková, že:

- pro $\forall r_i \in L$, platí $D(r, r_i) \leq k$,
- vhodným zřetězením prvků z L vznikne pravé rozšíření wp řetězce w.

Definice 2.3.3 (Pokrytí řetězce)**.** Řetězec s je pokrytí řetězce w právě tehdy, když vhodným překrytím a zřetězením kopií s vznikne právě řetězec w.

Definice 2.3.4 (k-aproximované pokrytí řetězce)**.** Řetězec s je k-aproximované pokrytí řetězce w při použití vzdálenostní funkce D právě tehdy, když existuje množina $L = \{s_1, s_2, \ldots, s_{|L|}\}$ taková, že:

- pro $\forall s_i \in L$, platí $D(s, s_i) \leq k$,
- vhodným překrytím a zřetězením kopií řetězců z L vznikne řetězec w.

Definice 2.3.5 (Jádro řetězce)**.** Řetězec s je jádro řetězce w právě tehdy, když vhodným překrytím a zřetězením kopií s vznikne nadřetězec $z = xwy$ řetězce w.

Poznámka: Speciálním případem jádra je pokrytí, což nastane, když levé a pravé rozšíření je prázdné.

Definice 2.3.6 (k-aproximované jádro)**.** Řetězec s je k-aproximované jádro řetězce w při použití vzdálenostní funkce D právě tehdy, když existuje množina $L = \{s_1, s_2, \ldots, s_{|L|}\}$ taková, že:

- pro $\forall s_i \in L$, platí $D(s, s_i) \leq k$,

- vhodným překrytím a zřetězením kopií řetězců z L vznikne nadřetězec $z = xwy$ řetězce w.

Poznámka: k-aproximované jádro také nazýváme k-přibližné jádro.

Definice 2.3.7 (Vlastní k-aproximované jádro)**.** Řetězec s je vlastní k-aproximované jádro řetězce w při použití vzdálenostní funkce D právě tehdy, když s je podřetězec w a s je k-aproximované jádro řetězce w při použití vzdálenostní funkce D.

2.4 Problémů

Problém 2.4.1 (Nalezení všech pokrytí řetězce)**.** Pro daný řetězec w najdi množinu $L^C(w)$ takovou, že každý řetězec $u \in L^C(w)$ je pokrytí řetězce w.

Problém 2.4.2 (Nalezení všech vlastních k-aproximovaných pokrytí řetězce)**.** Pro daný řetězec w, vzdálenostní funkci D a celé kladné číslo k, najdi množinu $L^C_{D^k}(w)$ takovou, že pro každou dvojici $(u, l) \in L^C_{D^k}(w)$ platí:

- u je podřetězec w,
- u je k-aproximované pokrytí řetězce w při použití vzdálenostní funkce D,
- l je nejmenší takové číslo, pro které platí, že u je l-aproximované pokrytí řetězce w při použití vzdálenostní funkce D.

Problém 2.4.3 (Nalezení všech jader řetězce)**.** Pro daný řetězec w najdi množinu $L^S(w)$ takovou, že každý řetězec $u \in L^S(w)$ je jádrem řetězce w.

Problém 2.4.4 (Nalezení nejmenší vzdálenosti aproximovaného jádra)**.** Uvažujme řetězec w a s a vzdálenostní funkci D najdi nejmenší celé kladné číslo k takové, že k je k-aproximované jádro řetězce w při použití vzdálenostní funkce D.

Problém 2.4.5 (Nalezení všech vlastních k-aproximovaných jader řetězce)**.** Pro daný řetězec w, vzdálenostní funkci D a celé kladné číslo k, najdi množinu $L^S_{D^k}(w)$ takovou, že pro každou dvojici $(u, l) \in L^S_{D^k}(w)$ platí:

- u je podřetězec w,
- u je k-aproximované jádro řetězce w při použití vzdálenostní funkce D,
- l je nejmenší takové číslo pro které platí, že u je l-aproximované jádro řetězce w při použití vzdálenostní funkce D.

Problém 2.4.6 (Nalezení všech přibližných jader řetězce)**.** Pro daný řetězec w, vzdálenostní funkci D a celé kladné číslo k, najdi množinu $L^S_{D^k}(w)$ takovou, že pro každou dvojici $(u, l) \in L^S_{D^k}(w)$ platí:

- u je k-aproximované jádro řetězce w při použití vzdálenostní funkce D,
- l je nejmenší takové číslo, pro které platí, že u je l-aproximované jádro řetězce w při použití vzdálenostní funkce D.

2.5 Konečných automatů

Definice 2.5.1 (Deterministický konečný automat)**.** Deterministický konečný automat (DKA) $\mathcal{M}$ je pětice $\mathcal{M} = (Q, A, \delta, q_0, F)$, kde:

- Q je neprázdná množina stavů,
- A je vstupní abeceda,
- δ je přechodová funkce typu $\delta : Q \times A \to Q$,
- $q_0 \in Q$ je počáteční stav,
- $F \subseteq Q$ je množina koncových stavů.

Definice 2.5.2 (Částečně definovaný DKA)**.** DKA $\mathcal{M} = (Q, A, \delta, q_0, F)$ je částečně definovaný právě tehdy, když je možné, že existuje dvojice (q, a) $q \in Q, a \in A$, pro kterou hodnota $\delta(q, a)$ není definována.

Definice 2.5.3 (Rozšířená přechodová funkce)**.** Pro DKA $\mathcal{M} = (Q, A, \delta, q_0, F)$ definujeme rozšířenou přechodovou funkci δ^*, typu $\delta^* : Q \times A^* \to Q$, induktivně takto: pro $q \in Q, a \in A, u \in A^*$

- $\delta^*(q, \varepsilon) = q$,
- $\delta^*(q, ua) = \delta(\delta^*(q, u), a)$.

Definice 2.5.4 (Řetězec přijímaný DKA)**.** Nechť DKA $\mathcal{M} = (Q, A, \delta, q_0, F)$ a řetězec $w \in A^*$, potom DKA $\mathcal{M}$ přijímá řetězec w právě tehdy, když $\delta^*(q_0, w) \in F$.

Definice 2.5.5 (Jazyk přijímaný DKA)**.** Nechť DKA $\mathcal{M} = (Q, A, \delta, q_0, F)$, potom jazyk L je přijímaný $\mathcal{M}$ právě tehdy, když obsahuje pouze řetězce $w \in L$ takové, že $\mathcal{M}$ přijímá w.

Definice 2.5.6 (Předchůdce stavu DKA)**.** Uvažujme DKA $\mathcal{M} = (Q, A, \delta, q_0, F)$, potom předchůdce stavu $q \in Q$ je jakýkoli stav $q_p \in Q$, pro který platí, že $\delta(q_p, a) = q$ pro některé $a \in A$.

Definice 2.5.7 (Stromový automat)**.** DKA $\mathcal{M} = (Q, A, \delta, q_0, F)$ je stromový automat právě tehdy, když pro každý stav $q \in Q \setminus \{q_0\}$ existuje pouze jeden předchůdce tohoto stavu.

Definice 2.5.8 (Acyklický DKA)**.** Uvažujme DKA $\mathcal{M} = (Q, A, \delta, q_0, F)$, potom je tento automat acyklický právě tehdy, když pro každý stav $q \in Q$ a všechny řetězce $w \in A^+$ platí $\delta^*(q, w) \neq q$, tzn. neexistuje neprázdný řetězec, pro který se dostaneme z jednoho stavu do toho samého.

Definice 2.5.9 (Levý jazyk stavu DKA)**.** Uvažujme DKA $\mathcal{M} = (Q, A, \delta, q_0, F)$ a jeho stav $q \in Q$, potom levý jazyk stavu q definujeme jako $L_q = \{w : w \in A^* \wedge \delta^*(q_0, w) = q\}$.

Definice 2.5.10 (Hloubka stavu)**.** Hloubka stavu $q \in Q$ acyklického automatu $\mathcal{M} = (Q, A, \delta, q_0, F)$ je délka nejdelšího řetězce z levého jazyka stavu q.

Značení 2.5.11 (lfactor). Levý jazyk stavu q stromového automatu obsahuje pouze jeden řetězec a ten značíme $\mathbf{lfactor}(q)$ a říkáme mu *lfactor*.

Definice 2.5.12 (Nedeterministický konečný automat)**.** Nedeterministický konečný automat (NKA) je pětice $\mathcal{M} = (Q, A, \delta, q_0, F)$, kde:

- Q je neprázdná množina stavů,
- A je vstupní abeceda,
- δ je přechodová funkce typu $\delta : Q \times A \to \mathcal{P}(Q)$,
- $q_0 \in Q$ je počáteční stav,
- $F \subseteq Q$ je množina koncových stavů.

Definice 2.5.13 (Rozšířená přechodová funkce pro NKA)**.** Nechť NKA $\mathcal{M} = (Q, A, \delta, q_0, F)$, potom rozšířená přechodová funkce δ^*, typu $\delta^* : Q \times A^* \to \mathcal{P}(Q)$ je, pro $q_1, q_2 \in Q$, $a \in A$ a $u \in A^*$, definována induktivně takto:

- $\delta^*(q_1, \varepsilon) = \{q_1\}$,
- $\delta^*(q_1, ua) = \bigcup_{q_2 \in \delta^*(q_1, u)} \delta(q_2, a)$.

Definice 2.5.14 (NKA s ε přechody)**.** NKA s ε přechody (ε-NKA) je pětice $\mathcal{M} = (Q, A, \delta, q_0, F)$, kde:

- Q je neprázdná množina stavů,
- A je vstupní abeceda,
- δ je přechodová funkce typu $\delta : Q \times (A \cup \{\varepsilon\}) \to \mathcal{P}(Q)$,
- $q_0 \in Q$ je počáteční stav,
- $F \subseteq Q$ je množina koncových stavů.

Definice 2.5.15 (ε-uzávěr)**.** Nechť ε-NKA $\mathcal{M} = (Q, A, \delta, q_0, F)$ a stavy $q_1, q_2 \in Q$, potom ε-uzávěr stavu q_1 značíme ε**-closure**(q_1) a definujeme takto:

- $q_1 \in \varepsilon$**-closure**(q_1),
- $q_2 \in \varepsilon$**-closure**(q_1) právě tehdy, když $q_2 \in \delta^*(q_1, \varepsilon)$.

Definice 2.5.16 (Rozšířená přechodová funkce pro ε-NKA)**.** Nechť ε-NKA $\mathcal{M} = (Q, A, \delta, q_0, F)$, potom rozšířená přechodová funkce δ^*, typu $\delta^* : Q \times A^* \to \mathcal{P}(Q)$ je, pro $q_1, q_2 \in Q$, $a \in A$ a $u \in A^*$, definována induktivně takto:

- $\delta^*(q_1, \varepsilon) = \varepsilon$**-closure**$\{q_1\}$,
- $\delta^*(q_1, ua) = \bigcup_{q_2 \in \delta^*(q_1, u)} \delta(q_2, a)$.

Definice 2.5.17 (Řetězec přijímaný NKA)**.** Řetězec w je přijímaný NKA $\mathcal{M} = (Q, A, \delta, q_0, F)$ bez nebo s ε přechody právě tehdy, když $\delta^*(q_0, w) \cap F \neq \emptyset$.

Definice 2.5.18 (Jazyk přijímaný NKA)**.** Jazyk L přijímaný NKA $\mathcal{M} = (Q, A, \delta, q_0, F)$ bez nebo s ε přechody obsahuje pouze řetězce, které jsou přijímány NKA $\mathcal{M}$.

Definice 2.5.19 (Konečný automat)**.** Konečný automat (KA) je buď DKA nebo NKA nebo ε-NKA.

Značení 2.5.20 (Jazyk přijímaný konečným automatem)*.* Jazyk přijímaný KA $\mathcal{M}$ značíme $L(\mathcal{M})$.

Definice 2.5.21 (Rovnost KA)**.** Dva KA $\mathcal{M}_1, \mathcal{M}_2$ jsou si rovny právě tehdy, když $L(\mathcal{M}_1) = L(\mathcal{M}_2)$, tedy když přijímají stejný jazyk.

Definice 2.5.22 (Prefixový automat)**.** Nechť řetězec $w \in A^*$, potom KA přijímající množinu všech prefixů řetězce w nazýváme prefixový automat.

Definice 2.5.23 (Sufixový automat)**.** Nechť řetězec $w \in A^*$, potom KA přijímající množinu všech sufixů řetězce w nazýváme sufixový automat.

Definice 2.5.24 (k-aproximovaný prefixový automat)**.** Nechť řetězec $w \in A^*$, celé číslo $k \geq 0$ a vzd. funkce D, potom KA přijímající všechny k-aproximované prefixy řetězce w při použití vzd. funkce D nazýváme k-aproximovaný prefixový automat pro w, D a k.

Definice 2.5.25 (k-aproximovaný sufixový automat)**.** Nechť řetězec $w \in A^*$, celé číslo $k \geq 0$ a vzd. funkce D, potom KA přijímající všechny k-aproximované sufixy řetězce w při použití vzd. funkce D nazýváme k-aproximovaný sufixový automat pro w, D a k.

Značení 2.5.26. Nedeterministický k-aproximovaný sufixový automat pro řetězec w a vzd. funkci D značíme $\mathcal{M}_{S^N}^{D,k}(w)$ a jeho deterministický ekvivalent značíme $\mathcal{M}_{S^D}^{D,k}(w)$.

Definice 2.5.27 (Podmnožinová konstrukce DKA)**.** Uvedena v [2]. Uvažujme KA $\mathcal{M} = (Q, A, \delta, q_0, F)$, potom DKA $\mathcal{M}' = (Q', A, \delta', q_0', F')$ vzniklý podmnožinovou konstrukcí definujeme takto:

1. Definuj $Q' = \{\{q_0\}\}$, stav $\{q_0\}$ bude neoznačený.
2. Jestliže v Q' jsou jen označené stavy, pokračuj krokem 4.
3. Vyber z Q' neoznačený stav q' a proveď:
   a) urči $\delta'(q', a) = \cup_{p \in q'} \delta(p, a)$, pro všechny $a \in A$,
   b) $Q' = Q' \cup \{\delta'(q', a)\}$, pro všechny $a \in A$,
   c) stav q' označ,
   d) pokračuj krokem 2.
4. Definuj $q_0' = \{q_0\}$.
5. Definuj $F' = \{q' : q' \in Q', q' \cap F \neq \emptyset\}$.

Definice 2.5.28 (d-subset)**.** Uvažujme DKA $\mathcal{M} = (Q, A, \delta, q_0, F)$ vzniklý pomocí podmnožinové konstrukce (viz 2.5.27) z NKA $\mathcal{M}_N$. Potom každý stav $q \in Q$ automatu $\mathcal{M}$ představuje množinu stavů z $\mathcal{M}_N$. Tuto množinu nazýváme d-subset a značíme jej $\mathbf{d}(q)$.

Kapitola 3

Testované algoritmy

V této kapitole představím algoritmy, kterými se zabývám. Prvně popíši fungování slovně. Dále uvedu časovou a paměťovou asymptotickou složitost algoritmu. Slovní popis shrnu tak, že jej také popíši formou pseudokódu.

3.1 Algoritmus používající dynamického programování

3.1.1 Popis

Prvním použitým algoritmem je algo. popsaný v [3]. Tento algo. má velkou variabilitu co se týče použití. Lze vybrat hledanou pravidelnost (periodu, pokrytí nebo jádro) pouhou změnou hodnot parametrů. Dále lze použít Hammingovou, editační nebo váženou editační vzdálenost. Výběr vzd. funkce způsobí tentokrát větší změnu algoritmu, ale pouze na jednom místě a to počítání tabulky d (viz níže).

Tento algoritmus je primárně konstruován na řešení problému nalezení nejmenší vzdálenosti aproximovaného jádra (2.4.4). a algoritmus je posléze použit na řešení mého cílového problému 2.4.5, tedy nalezení všech vlastních k-aproximovaných jader řetězců. Proto první popíši algoritmus 3.1.2 řešící problém 2.4.4 a poté algoritmus 3.1.1 řešící problém 2.4.5 za použití algo. 3.1.2.

Algoritmus 3.1.2 řeší svůj problém tak, že jej rozdělí do dvou částí, dvou podproblémů. První částí je nalezení vzdáleností mezi všemi podřetězci řetězce w a testovaného jádra s. Tyto vzdálenosti jsou uloženy do tabulky d (v původní práci značeno w_{ij}). Tato tabulka má rozměry $n \times n$, kde n je délka řetězce w. Prvek d_{ij} tabulky d představuje vzdálenost mezi podřetězcem $w[i..j]$ a testovaným jádrem s, tedy $d_{ij} = D(w[i..j], s)$. Prvky této tabulky jsou plněny podle použité vzdálenostní funkce D a hledané pravidelnosti. Jelikož se zabývám pouze jádry s Hammingovou vzdáleností, popíši pouze postup pro tuto konfiguraci. Nejprve naleznu vzdálenost mezi testovaným jádrem a podřetězci

w, které mají stejnou délku, protože Hammingova vzdálenost je definovaná pouze pro řetězce stejné délky. Takové podřetězce jsou $w[1..m], w[2..m+1]$ až $w[n-m+1..n]$. Dále je potřeba spočítat vzdálenost mezi všemi sufixy jádra s s příslušně dlouhými prefixy w. Jsou to vzdálenosti mezi $w[1..m-1]$ a $s[2..m]$, $w[1..m-2]$ a $s[3..m]$ až $w[1..1]$ a $s[m..m]$. Tyto vzdálenosti představují možnost, že nadřetězec vzniklý pomocí zřetězení a překrytí jader má neprázdné levé rozšíření. Z obdobného důvodu, tedy neprázdného pravého rozšíření, je potřeba dále spočítat vzdálenost mezi všemi prefixy jádra s a příslušně dlouhými sufixy řetězce w.

Když máme takto vyplněnou tabulku d můžeme přejít k nalezení nejmenšího k. To je prováděno tak, že spočteme hodnoty posloupnosti t_i pro i od 0 do n. Posloupnost t_i představuje minimální vzdálenost, se kterou je možné dosáhnout pokrytí na podřetězci $w[1..i]$. Posloupnost t_i je při použití Hammingovy vzdálenosti definovaná takto:

- $t_0 = 0$,
- $t_i = \max\left\{\min_{h \le j \le i-1}\{t_j\}, d_{h+1,i}\right\}$, kde $h = \max\{0, i-m\}$, pro $i \in \{1, 2, \ldots, n-1\}$,
- $t_n = \min_{n-m \le h < n}\left\{\max\left\{\min_{h \le j \le n-1}\{t_j\}, d_{h+1,n}\right\}\right\}$.

Takovýto výpočet i-tého prvku představuje to, že na zkonstruování podřetězce $w[1..i]$ použijeme patřičně upravené jádro s, které vložíme na konec námi vytvářeného řetězce. Potřebnou úpravu jádra s představuje hodnota $d_{h+1,i}$. Jako začátek použijeme nejlepší konstrukci, kterou jsme dostali v předchozích krocích. Začátek vybíráme tak, aby nevznikla v řetězci díra (např. $w[1..4]$ a $w[6..8]$, takové části nelze použít, protože by chyběl symbol na pozici pět) a zároveň, abychom alespoň jeden symbol přidali. Důvodem proč se výpočet posledního prvku posloupnosti t_i liší, je to, že uvnitř řetězce musíme použít celé délky jádra s, ale na konci je možná volba, jak moc vzniklý řetězec rozšíříme. Podobnou možnost rozšíření máme i na začátku řetězce, ta je zprostředkována výběrem nejvhodnějšího minulého t_i.

U posledního prvku t bychom museli počítat $\mathcal{O}(m^2)$ operací pro jeho určení, kdybychom použili naivní přístup. Nicméně lze použít zlepšení, které je podobné zlepšení počítání $\min_{h \le j \le j_max}\{t_j\}$ v [3], ale které nelze použít při Hammingově vzdálenosti. Toto zlepšení počítá t_n ve zpětném pořadí od $n-1$ do $n-m$. V tomto pořadí se interval, ve kterém počítáme nejvnitřnější minimum, zvětšuje pouze o jedna, proto jej můžeme mít uložen v jedné proměnné. Tuto proměnnou poté při zvětšení intervalu v důsledku změny h pouze aktualizujeme se složitostí $\mathcal{O}(1)$. A proto místo $\mathcal{O}(m^2)$ máme složitost $\mathcal{O}(m)$ na spočítání poslední položky t_n.

Výsledkem algo. je hodnota t_n, která představuje nejmenší takové k, pro které je s k-aproximovaným jádrem w při použití vzd. funkce D.

Pro řešení problému 2.4.5 za použití algo. 3.1.2 postupujeme následovně. Postupně pro každý podřetězec řetězce w počítáme hodnotu l pomocí algo. 3.1.2. Hodnota l představuje nejmenší vzdálenost zkoumaného jádra, tedy je to hodnota k z algo. 3.1.2. Pokud hodnota l splňuje kritérium $l \leq k$, poté ji přidáme do výsledné množiny.

3.1.2 Pseudokód

Algoritmus 3.1.1: Algoritmus hledající všechny vlastní k-aproximované jádra řetězce w při použití Hammingovy vzdálenosti za použití dynamického programování

Input: Řetězec w délky n a číslo k
Output: Množina $L^S_{D^k}(w)$ taková, že řeší problém 2.4.5

```
1  Vytvoř prázdnou množinu L^S_{D^k} ;
2  for i ← 1 to n do
3  |   for j ← i to n do
4  |   |   Spočítej l za použití algoritmu 3.1.2, kde w ← w a s ← w[i..j];
5  |   |   if l ≤ k then
6  |   |   |   L^S_{D^k}(w) ← L^S_{D^k}(w) ∪ {(l, w[i..j])} ;
7  |   |   end
8  |   end
9  end
10 return L^S_{D^k}
```

Algoritmus 3.1.2: Algoritmus hledající nejmenší vzdálenost aproximovaného jádra za použití dynamického programování

Input: Řetězec w délky n a s délky m
Output: Nejmenší k takové, že řeší problém 2.4.4

Vytvoř tabulku d, velikosti $n \times n$;
`// Vyplnění tabulky` d
for $i \leftarrow 1$ **to** $n - m + 1$ **do** `// Použití celých podřetězců`
  $j \leftarrow i + m - 1$;
  $d_{ij} =$ **HammingDistance**$(w[i..j], s)$;
end
for $i \leftarrow 2$ **to** m **do** `// Použití sufixů jádra`
  $j \leftarrow m - i + 1$;
  $d_{ij} =$ **HammingDistance**$(w[1..j], s[i..m])$;
end
for $i \leftarrow 1$ **to** $m - 1$ **do** `// Použití prefixů jádra`
  $j \leftarrow n - i + 1$;
  $d_{ij} =$ **HammingDistance**$(w[j..n], s[1..i])$;
end

Vytvoř posloupnost t, délky $n + 1$;
$t_0 \leftarrow 0$;
for $i \leftarrow 1$ **to** $n - 1$ **do** `// Spočítání minimálního pokrytí od 1 do` $n - 1$
  $h \leftarrow \max(0, i - m)$;
  $tmp \leftarrow$ Najdi minimum z t_h až t_{i-1} ;
  $t_i \leftarrow \max(tmp, d_{h+1,i})$
end
$t_n \leftarrow \infty$;
$min\_t_j \leftarrow \infty$;
for $i \leftarrow n - 1$ **downTo** $n - m$ **do** `// Spočítání minimálního pokrytí pro poslední prvek` t
  $min\_t_j \leftarrow \min(min\_t_j, t_i)$;
  $tmp \leftarrow \max(min\_t_j, d_{i+1,n})$;
  $t_n \leftarrow \min(t_n, tmp)$;
end

return t_n

3.2 Algoritmus používající konečných automatů

3.2.1 Popis

Druhý zkoumaný algoritmus používá konečných automatů. Jeho autorem je můj vedoucí bakalářské práce Ondřej Guth, který jej prezentoval ve své dizertační práci [4]. V této práci se zaměřil na řešení problémů nalezení všech vlastních pokrytí a jader řetězců.

Základem tohoto řešení je KA přijímající všechny řetězce, které mají od daného řetězce w vzdálenost maximálně k. Konstrukce tohoto KA je popsána v algo. 3.2.3. Stavy tohoto automatu jsou značeny q_i^j, kde i představuje pozici v řetězci w a j představuje vzdálenost od prefixu $w[1..i]$. Proto levý jazyk (viz definice 2.5.9) stavu q_i^j obsahuje pouze řetězce nad efektivní abecedou A_w, které mají od řetězce $w[1..i]$ maximální vzdálenost j. Tohoto faktu dále využijeme k tomu, abychom byli schopni určit konečné pozice nějakého řetězce v řetězci w.

Tohoto efektu docílíme tak, že automat přetvoříme na k-aproximovaný sufixový automat. Transformace spočívá ve spojení počátečního stavu q_0^0 se všemi ostatními stavy s přibližností 0, tedy q_x^0 ε přechody (viz algo. 3.2.1 řádka 2). Zásah způsobí nedeterminističnost automatu. Pokud pro řetězec v nyní projdeme automatem, dostaneme množinu stavů (tzn. $\delta^*(q_0^0, v)$). Tyto stavy vzniklé v předchozím kroku stále nesou informaci o pozici a přibližnosti. Informace ale změnila podstatu, nyní stav q_i^j říká, že řetězec v okupuje pozici i se vzdáleností j.

Aby se s automatem lépe pracovalo, odstraníme z automatu ε přechody (algo. 3.2.4) a poté jej převedeme na DKA. Převod provedeme pomocí množinové konstrukce popsané v [2]. Při této konstrukci nový stav DKA představuje množinu stavů z NKA, a ta se nazývá d-subset $\mathbf{d}(q_N)$. Postup převodu nedodržíme doslovně. Prvně, pokud nový stav neobsahuje ve svém d-subsetu stav s nulovou přibližností, tak jej nezahrneme do DKA. Takový stav by nereprezentoval podřetězec řetězce w. Vytváření nového stavu popisuje algo. 3.2.6. Druhou změnou je průběžné testování, zda řetězec reprezentovaný stavem je jádrem (viz algo. 3.2.5 řádka 25 až 32).

Pro testování, zda je řetězec reprezentovaný stavem opravdu jádrem, potřebujeme d-subset testovaného stavu, maximální dovolenou vzdálenost l a sufixové automaty pro řetězce w a w^R. Prvním testem je ověření, zda kandidát na jádro pokryje vnitřek řetězce w. Tuto skutečnost ověříme tím, že rozdíl mezi indexy u dvou po sobě jdoucích stavů v d-subsetu je maximálně l (viz řádek 1 v algo. 3.2.8). Dále hledáme pokrytí konce řetězce w. K tomu poslouží aproximovaný sufixový automat pro řetězec w, který postupně vytváříme. Postupně tedy zkoušíme prefixy kandidáta, a pokud je prefix dostatečně dlouhý, je přijímán automatem a má dostatečně malou přibližnost, potom pokrývá konec řetězce. Dostatečná délka kandidáta znamená, že rozdíl délky w a poslední pozice v d-subsetu je menší nebo roven délce prefixu. Stav, do kterého se

dostaneme průchodem automatu podle testovaného prefixu, musí být koncový a přibližnost posledního stavu v d-subsetu musí být menší nebo rovna k. Tato podmínka je v algo. 3.2.8 na řádce 2. Pokrytí začátku řetězce w nalezneme podobně jako konec. Místo sufixového automatu pro w použijeme sufixový automat pro w^R. V něm testujeme reverz sufixu kandidáta. Ten musí být přijat a mít příslušnou vzdálenost. Jeho délka musí být větší nebo rovna rozdílu první pozice v d-subsetu a délky kandidáta (viz řádka 3 v algo. 3.2.8).

Když kandidát splňuje předchozí tři podmínky, je jádrem. Pro nalezené jádro nyní potřebujeme nalézt nejmenší možnou vzdálenost pro jádro. Na to slouží algo. 3.2.7. Ten pracuje tak, že odebere prvky se vzdáleností rovnou startovní a otestuje, zda je jádro jádrem při o jedna menší vzdálenosti. Pokud ano, pokračuje s odebíráním prvků se vzdáleností o jedna menší a testuje nižší vzd. jádra. Pokud testem jádro neprojde, algo. skončil a minimální vzd. je vzd., která poslední úspěšně prošla testy.

Na závěr to shrneme. Jako první vytvoříme NKA pro přijímání k-aproximovaných sufixů řetězce w a jeho reverzu podle algo. 3.2.1. Nejprve provedeme determinizaci automatu pro w^R, řádky 11 až 19 algo. 3.2.5. Pokračujeme provedením determinizace automatu pro řetězec w. V té, ale při přidání nového stavu, zároveň testujeme, zda levý faktor tohoto stavu je jádrem s příslušnou aproximací. Test provede algo. 3.2.8. Pokud je jádrem, hledáme jeho minimální vzdálenost pomocí algo. 3.2.7. Výsledkem celého algo. je množina dvojic skládající se z jádra a jeho minimální vzdálenosti.

Algoritmus 3.2.7, který hledá nejmenší vzdálenost aproximovaného jádra, byl uveden v [4, str. 82]. Nicméně v původní verzi je chyba, která způsobuje chybné určení min. vzdálenosti ve specifických případech. Tuto chybu jsem objevil, když jsem porovnával výstupy obou algo. proti sobě. Nalezená jádra se shodovala i jejich počet, ale minimální vzdálenosti se u některých jader lišily. Po prozkoumání příčiny jsem tuto chybu objevil.

Původní verze na řádce 2 algo. 3.2.7 obsahuje výraz $l \leftarrow \max_{r \in C} (\mathbf{level}(r))$, tedy nalezení maximální vzdálenosti elementů v d-subsetu předaného této proceduře. Poté stejně jako v algo. 3.2.7 autor pokračoval postupným odebíráním prvků a testováním, zda kandidát je pořád jádrem. Tento způsob výběru startovní hodnoty způsobuje problém. Připomínám, že prvky d-subsetu popisují, jak jádro vyplní střed zkoumaného řetězce. Začátek a konec, který není pokryt podle d-subsetu, je potřeba pokrýt jinak. Jelikož pokrytí začátku a konce nevyplývá z d-subsetu, je možné, že pokrytí začátku nebo konce má ostře vetší vzdálenost než pokrytí středu. Pokud takovéto jádro testujeme, hned v první iteraci dostaneme negativní výsledek. Načež naše nalezená nejmenší vzdálenost je první testovaná, což nemusí být správný výsledek. Mojí opravou této chyby je určení první testované vzdálenosti jako jeden z předaných parametrů této procedury. Hodnotou parametru je vždy k, tj. maximální dovolená vzdálenost pro jádra. Při tomto testu je maximální možná vzdálenost k, která již předtím byla úspěšně otestována.

3.2.2 Pseudokód

Algoritmus 3.2.1: Algoritmus na vytvoření nedeterministického k-aproximovaného sufixového automatu $\mathcal{M}_{S^N}^{H,k}(w) = (Q, A, \delta, q_0, F)$ pro řetězec $w \in A^+$ a Hammingovu vzdálenost.

Input: Řetězec $w \in A^+$, celé číslo $k \geq 0$
Output: Nedeterministický k-aproximovaný sufixový automat bez ε přechodů $\mathcal{M}_{S^N}^{H,k}(w) = (Q, A, \delta, q_0, F)$

1 Vytvoř k-aproximovaný prefixový automat $\mathcal{M}_P = (Q_P, A, \delta_P, q_{0P}, F_P)$ pro w, Hammingovu vzd. a k, podle algo. 3.2.2 ;
2 Definuj $Q = Q_P$, $\delta = \delta_P$, $q_0 = q_{0P}$;
3 **for** $i \leftarrow 0$ **to** $|w|$ **do**
4 | Pro stav q_i^0 vytvořený v algo. 3.2.3 definuj $\delta(q_0, \varepsilon) = q_i^0$;
5 **end**
6 **for** $j \leftarrow 0$ **to** k **do**
7 | Pro stav $q_{|w|}^j$ vytvořený v algo. 3.2.3 přiřaď $F = F \cup \left\{q_{|w|}^j\right\}$;
8 **end**
9 Odstraň ε přechody z $\mathcal{M}_{S^N}^{H,k}(w)$ podle algo. 3.2.4 ;
10 **return** $\mathcal{M}_{S^N}^{H,k}(w)$

Algoritmus 3.2.2: Algoritmus na vytvoření nedeterministického k-aproximovaného prefixového automatu $\mathcal{M} = (Q, A, \delta, q_0^0, F)$ pro řetězec $w \in A^+$ a Hammingovu vzdálenost.

Input: Řetězec $w \in A^+$ a celé číslo $k \geq 0$
Output: Nedeterministický k-aproximovavý prefixový automat $\mathcal{M} = (Q, A, \delta, q_0, F)$

1 Vytvoř automat $\mathcal{M}_B = (Q_B, A, \delta_B, q_{0B}, F_B)$ podle algo. 3.2.3 ;
2 Definuj $Q = Q_B$, $\delta = \delta_B$, $q_0 = q_{0B}$;
3 Definuj $F = Q$;
4 **return** $\mathcal{M} = (Q, A, \delta, q_0, F)$

Algoritmus 3.2.3: Algoritmus na vytvoření KA $\mathcal{M} = (Q, A, \delta, q_0^0, F)$ pro příjem všech řetězců s maximální vzdáleností k od řetězce w při použití Hammingovy vzd. funkce

Input: Řetězec $w \in A^+$ a celé číslo $k \geq 0$
Output: Nedeterministický automat $\mathcal{M} = (Q, A, \delta, q_0^0, F)$

1 Definuj $Q = \left\{ q_i^j : \ 0 \leq i \leq |w|, 0 \leq j \leq k \right\}$;
2 Definuj $F = \left\{ q_{|w|}^j : 0 \leq j \leq k \right\}$;
3 **for** *i, j takové, že $1 \leq i \leq |w|, 0 \leq j \leq k$* **do**
4 | Definuj $\delta\left(q_{i-1}^j, w[i]\right) = q_i^j$;
5 **end**
6 **for** *i, j, a takové, že $1 \leq i \leq |w|, 0 \leq j \leq k, a \in A \setminus \{w[i]\}$* **do**
7 | Definuj $\delta\left(q_{i-1}^{j-1}, a\right) = q_i^j$;
8 **end**
9 Odeber všechny stavy q_i^j, pro které neexistuje řetězec $u \in A^*$ takový, že $q_i^j \in \delta^*(q_0^0, u)$;
10 **return** $\mathcal{M} = (Q, A, \delta, q_0^0, F)$

Algoritmus 3.2.4: Algoritmus na odstranění ε přechodů.

Input: KA s ε přechody $\mathcal{M}_E = (Q_E, A, \delta_E, q_{0E}, F_E)$
Output: KA bez ε přechodů $\mathcal{M} = (Q, A, \delta, q_0, F)$

1 Definuj $Q = Q_E$;
2 Definuj $q_0 = q_{0E}$;
3 **forall the** $q_1 \in Q$, $a \in A$ **do**
4 | Definuj $\delta(q_1, a) = \bigcup_{q_2 \in \varepsilon - \text{closure}(q_1)} \delta_E(q_2, a)$;
5 **end**
6 Definuj $F = \{q : \varepsilon - \text{closure}(q) \cap F_E \neq \emptyset, q \in Q\}$;
7 **return** $\mathcal{M} = (Q, A, \delta, q_0, F)$

Algoritmus 3.2.5: Algoritmus hledající všechna vlastní k-aproximovaná jádra řetězce w při použití Hammingovy vzdálenosti, za použití konečných automatů.

Input: Řetězec $w \in A^+$ a maximální dovolená vzdálenost k
Output: Množina $L^S_{D^k}(w)$ taková, že řeší problém 2.4.5

Vytvoř prázdnou množinu $L^S_{D^k}(w)$;
Spočítej efektivní abecedu A_w řetězce w;
Vytvoř automat $\mathcal{M}^{H,k}_{S^N}(w)$ podle algo. 3.2.1 za použití w a k;
Vytvoř automat $\mathcal{M}^{H,k}_{S^N}(w^R)$ podle algo. 3.2.1 za použití w^R a k;
Vytvoř prázdné fronty stavů F a F^R;
Vytvoř stav q_{0D} a nastav ho jako počáteční stav automatu $\mathcal{M}^{H,k}_{S^D}(w)$;
Vytvoř stav q^R_{0D} a nastav ho jako počáteční stav automatu $\mathcal{M}^{H,k}_{S^D}(w^R)$;
lfactor$(q_{0D}) \leftarrow \varepsilon$;
depth$(q_{0D}) \leftarrow 0$; **depth**$(q^R_{0D}) \leftarrow 0$;
zařaď(F, q_{0D}); **zařaď**(F^R, q^R_{0D});
while *F^R není prázdná* **do**
    $q^R_t \leftarrow$ **vyřaď**(F^R);
    if ***depth***$(q^R_t) < |w|$ **then**
        forall the $a \in A_w$ **do**
            Vytvoř nový stav pomocí algo. 3.2.6 pro $\mathcal{M}^{H,k}_{S^D}(w^R)$ a F^R;
        end
    end
    Odstraň všechny prvky z $\mathbf{d}(q^R_t)$ kromě posledního;
end
while *F není prázdná* **do**
    $q_t \leftarrow$ **vyřaď**(F);
    if ***depth***$(q_t) < |w|$ **then**
        forall the $a \in A_w$ **do**
            Vytvoř stav q_u pomocí algo. 3.2.6 pro $\mathcal{M}^{H,k}_{S^D}(w)$ a F;
            if *$\exists r \in \boldsymbol{d}(q_u)$ takový, že* ***level***$(r) = 0$ **then**
                $y \leftarrow$ **lfactor**$(q_t) \cdot a$;
                lfactor$(q_u) \leftarrow y$;
                if *Rozhodni podle algo. 3.2.8 pro y, $\boldsymbol{d}(q_u)$, k* **then**
                    Spočítej l za pomoci algo. 3.2.7 pro $\mathbf{d}(q_u)$, y, k;
                    $L^S_{D^k}(w) \leftarrow L^S_{D^k}(w) \cup \{(l, y)\}$;
                end
            end
        end
    end
    Odstraň všechny prvky z $\mathbf{d}(q_t)$ kromě posledního;
end
return $L^S_{D^k}(w)$

Algoritmus 3.2.6: Vytvoř nový stav q_u pro postupně konstruovaný k-aproximovaný sufixový automat $\mathcal{M} = (Q_D, A, \delta_D, q_{0D}, F_D)$

Input: Konstruovaný automat $\mathcal{M}$, nedeterministická verze automatu $\mathcal{M}_N = (Q_N, A, \delta_N, q_{0N}, F_N)$, stav $q_t \in Q_D$, symbol $a \in A$ a fronta F

Output: Nový stav q_u. Upravený automat $\mathcal{M}$, změněná fronta F

```
1  Vytvoř nový stav q_u;
2  depth(q_u) ← depth(q_t) + 1;
3  δ_D(q_t, a) ← q_u;
4  forall the r_i ∈ d(q_t) (v pořadí přidání do d(q_t)) do
5  |  přidej na konec ∀r_j ∈ δ_N(r_i, a) do d(q_u);
6  end
7  if |d(q_u)| > 0 then
8  |  if ∃r ∈ d(q_u) takový, že level(r) = 0 then
9  |  |  Q_D ← Q_D ∪ {q_u};
10 |  |  zařaď(F, q_u);
11 |  |  if poslední prvek v d(q_u) leží v F_N then
12 |  |  |  F_D ← F_D ∪ {q_u};
13 |  |  end
14 |  end
15 end
16 return q_u
```

Algoritmus 3.2.7: Algoritmus určující nejmenší vzdálenost k-aproximovaného jádra s řetězce w. Upravená verze algoritmu popsaném v [4].

Input: d-subset $\mathbf{d}(q) = \{r_1, r_2, \ldots, r_{|\mathbf{d}(q)|}\}$, jádro $s = \mathbf{lfactor}(q)$ a počáteční vzdálenost k

Output: Nejmenší l takové, že s je l-aproximované jádro řetězce w

```
1  C ← d(q);
2  l ← k;
3  repeat
4  |  forall the r ∈ C : level(r) = l do
5  |  |  Odeber r z C;
6  |  end
7  |  l ← l − 1;
8  until algo. 3.2.8 pro s, C a l, vrací ano;
9  return l + 1
```

Algoritmus 3.2.8: Rozhodni, zda řetězec s je k-aproximované jádro řetězce w.

Input: Kandidát na jádro řetězce s, d-subset $C = \{r_1, r_2, \ldots, r_{|C|}\}$ pro řetězec s v KA $\mathcal{M}(w)$, vzdálenost k a k-aproximované sufixové automaty $\mathcal{M}(w)$ a $\mathcal{M}(w^R)$

Output: Odpověď, zda je s k-aproximovaným jádrem w

```
1 if pro všechny i = 2,3,...,|C| platí r_i − r_{i−1} ≤ |s| then
2   if ∃p ∈ pref(w), |p| ≥ (|w| − r_{|C|}) ; δ(q_0, p) = q_1 ; r_{|d(q_1)|} ∈ d(q_1)
    a q_1 ∈ F a level(r_{|d(q_1)|}) ≤ k then     // Automat přijme prefix
    a prefix vyplní zbytek řetězce
3     if
      ∃s ∈ suff(w), |s| ≥ (r_1 − |s|) ; δ*_R(q_{0R}, s^R) = q_2 ; r_{|d(q_2)|} ∈ d(q_2)
      a q_2 ∈ F_R a level(r_{|d(q_2)|}) ≤ k then // Automat přijme sufix
      a sufix vyplní začátek řetězce
4       return Ano
5     end
6   end
7 end
8 return Ne
```

3.3 Asymptotické složitosti

V předchozích podkapitolách jsem popsal algoritmy na hledání vlastních k-aproximovaných jader řetězce. To, co jsem u nich neuvedl, jsou jejich složitosti. Ty uvádím v tabulce 3.1 pro jejich snadnější srovnání.

Při uvádění složitostí uvažuji hledání všech vlastních k-aproximovaných jader řetězce w. Řetězec w má délku $n = |w|$. Řetězec w je konstruován nad abecedou A a efektivní abeceda řetězce w je A_w. Maximální vzdálenost jádra je k.

Tabulka 3.1: Srovnání složitostí

Použitý algoritmus	Časová složitost	Paměťová složitost
Konečné automaty	$\mathcal{O}\left(k \cdot \lvert A_w \rvert \cdot n^3\right)$	$\mathcal{O}\left(n^2\right)$
Dynamické programování	$\mathcal{O}\left(n^3\right)$	$\mathcal{O}\left(n^2\right)$

Uvedené asymptotické složitosti algoritmu používající KA lze najít v [4]. Časovou asymptotickou složitost algo. s dyn. programováním lze najít v [3], ale paměťovou složitost zde nenalezneme. Proto jsem ji určil sám.

Paměťová složitost algo. založeného na dynamickém programování je $\mathcal{O}\left(n^2\right)$. Důvod: V tomto algo. používáme tabulku o velikosti $\mathcal{O}(n^2)$ a pole o velikosti $\mathcal{O}(n)$. Z toho plyne, že algo. má paměťovou složitost $\mathcal{O}\left(n^2\right)$.

3.4 Generování řetězců pro testování

3.4.1 Popis

Při testování vlastností výše popsaných algoritmů jsem testoval i vliv velikosti efektivní abecedy. Abych mohl tento vliv testovat, potřeboval jsem data s různými abecedami. Nějaká takováto data je obtížné nalézt. A proto, abych mohl vliv abeced otestovat, rozhodl jsem se generovat vlastní data.

Způsob, který jsem zvolil, generuje řetězce s ohledem na to, aby v nich byla velká míra opakování. Dále pro uplatnění přibližnosti se opakující části změní. Jelikož jádra mohou vytvořit zadaný text pomocí překrytí, algo. mění délku opakující se sekvence.

Algoritmus tedy jako svůj vstup bere délku základní sekvence l_s, počet opakování n_r, počet změn m_d a příslušnou abecedu A. Algoritmus nejprve vygeneruje náhodný řetězec s nad zadanou abecedou A o délce l_s. S tímto řetězcem následně n_r-krát provede následující. Odebere náhodný počet prvků ze začátku a z konce. V tomto řetězci náhodně změní několik znaků a přidá ho

na konec generovaného textu w. Součet odebraných znaků ze začátku a z konce a počet změněných znaků je roven počtu změn m_d.

3.4.2 Pseudokód

Algoritmus 3.4.1: Algoritmus generování řetězců pro testování

Input: Abeceda A, celá čísla $l_s, n_r > 0$ a $m_d \leq 0$
Output: Řetězec w

```
1  Definuj w = ε;
2  Nad abecedou A vygeneruj řetězec s délky l_s;
3  for i ← 1 to n_r do
4  |   Přiřaď do i náhodné číslo mezi 0 a m_d;
5  |   Přiřaď do j náhodné číslo mezi 0 a m_d − i;
6  |   v ← s[i..j];
7  |   Náhodně změň m_d − i − j znaků v řetězci v;
8  |   w ← w · v;
9  end
10 return w
```

Kapitola 4

Použité softwarové vybavení

Při realizaci této bakalářské práce jsem potřeboval softwarové vybavení, jednak pro realizaci experimentů, tak i pro zpracování, vizualizaci a popis výsledků.

Pro zpracování výsledků jsem použil již existující programy. Pro vizualizaci naměřených dat jsem použil program *gnuplot*. A v poslední řadě LaTeX pro napsání této práce.

Ale pro realizaci experimentů byla potřeba vlastní invence. V první řadě implementace výše zmíněných algoritmů (Kapitola 3) pro řešení problému hledání všech vlastních k-aproximovaných jader zadaného řetězce (2.4.5). Jejich implementace je popsána v následující podkapitole.

Za druhé pro jednodušší spouštění testů jsem si napsal pomocný program. Tento program bere jako parametry z příkazové řádky jména souborů, které obsahují jednoduchý popis spouštěného testu. Popis se skládá z jména souboru s testovanou implementací, jména souboru, ve kterém jsou uložena testovaná data, intervalů počátku sekvence v souboru, délek sekvence, maximálních dovolených přibližností a souboru pro uchování (logování) dat. Intervaly jsou zadány jako počáteční hodnota, maximální hodnota a velikost kroku. Program je realizován v programovacím jazyce C++ a pro spouštění testů používá standardní funkce *int system(const char * command)*.

4.1 Implementace algoritmu

Implementaci všech algoritmů jsem provedl v jazyce C++, normě C++11. Implementace lze nalézt v adresáři /programs na přiloženém CD.

4.1.1 Algoritmus využívající dynamického programování

Implementace algoritmů 3.1.1 a 3.1.2 je přímočará bez nutnosti zásadních změn nebo voleb struktur.

Jedinou změnou v implementaci oproti pseudokódu je omezení alokace dynamické paměti potřebné na tabulku d. Tato tabulka by se měla vytvářet $\mathcal{O}(n^2)$, což je zbytečné a především to způsobí dvojnásobné prodloužení doby běhu.

Realizaci algo. 3.1.2 lze nalézt na přiloženém CD v souboru /programs/dynamic/smallestApproximateSeed.cpp. Realizaci algo. 3.1.1 lze nalézt na přiloženém CD v souboru /programs/dynamic/allRestrictedApproximateSeed.cpp.

4.1.2 Algoritmus využívající konečných automatů

Implementace algoritmu 3.2.5 používající KA je poněkud rozsáhlejší. Důvodem je velké množství dílčích kroků, jako je konstrukce automatů, odstranění ε-přechodů a determinizace. Nicméně jejich realizace je přímočará.

K tomu, abych dosáhl této přímočarosti, jsem použil třídy pro reprezentaci konečných automatů a jejich vnitřních stavů. Stav nedeterministického automatu reprezentuje struktura *NDState*, která obsahuje pouze informaci o pozici a přibližnosti, která přísluší danému stavu. Mohl by dále obsahovat informaci o přechodech vedoucích z tohoto stavu, nebo zda je koncový. Ale ty v mé implementaci nejsou uloženy ve stavu, protože tyto stavy jsou následně uloženy v d-subsetech a je dobré, když je tato struktura co nejmenší. Třída reprezentující celý NKA (*NDMachine*) obsahuje informace o počátečním stavu automatu, koncových stavech a přechodové funkci. Přechodová funkce je reprezentovaná jako mapa, kde klíčem je stav, pro který se přechod hledá a hodnotou je mapa. Vnitřní mapa má za klíč znak, to je druhý argument přechodové funkce a hodnotou je množina stavů, která je výsledkem funkce. Pro vytvoření ε-přechodů jsem přidal do třídy položku reprezentující ε-uzávěr daného stavu. Tato položka neobsahuje celý ε-uzávěr, ale jakousi jeho část, to však nevadí vzhledem k tomu, že v algo. vznikne pouze jeden stav s ε-přechody. Položka je realizována jako mapa, kde klíčem je stav, pro který hledáme ε-uzávěr a hodnota je množina stavů. Třída *NDMachine* dále obsahuje metody pro přidávání nových stavů, koncových stavů, rozšiřování přechodové funkce, testování, zda je stav koncový a získávání hodnot přechodové funkce.

U implementace deterministických konečných automatů jsem také použil dělení na stav a automat. Nicméně třída *DMachine* reprezentující DKA je pouhá obálka pro počáteční stav a identifikaci koncových stavů automatu. Proto jediné metody v této třídě jsou na změnu a zpřístupnění těchto položek. Třída *DState* reprezentující stav DKA je obsáhlá. Obsahuje přechodovou funkci, která přechody z tohoto stavu ukládá jako pole párů obsahující znak a stav. Důvodem této volby je úspora paměti. Další položkou je hloubka stavu podle definice 2.5.10. Poslední položkou je množina stavů NKA, ze kterých daný stav vznikl, tzv. d-subset.

Když se podíváme do algo. 3.2.5, vidíme že u stavů DKA je atribut *lfactor*. Tento atribut se i v algo. 3.2.5 počítá, a tak by se mohlo zdát, že by měl

být u stavu i uložen. Když jsem tak učinil, způsobilo to významný nárůst potřebné paměti. To lze obejít, a to tak, že si *lfactor* dopočítáme. Dopočítání je možné, jelikož v každém d-subsetu nalezneme alespoň jeden stav NKA s nulovou vzdáleností. Pozice tohoto stavu je pozicí, kde daný *lfactor* v testovaném textu končí. Díky hloubce stavu jsme schopni určit délku, kterou má *lfactor*. Díky tomu si nemusím *lfactor* pamatovat.

Pro realizaci map jsem použil mapu *std::map* z STL. Pro realizaci množin jsem použil množinu *std::set* z STL.

Implementaci algo. 3.2.1, 3.2.2, 3.2.3 a 3.2.4 lze nalézt na přiloženém CD v souboru /programs/fsm/constructMachine.cpp. Zde jsou algo. 3.2.2 a 3.2.3 spojeny do jedné funkce. V souboru /programs/fsm/NDMachine.cpp je implementace tříd reprezentující NKA a v souboru /programs/fsm/DMachine.cpp jsou třídy reprezentující DKA. Implementaci algo. 3.2.5, 3.2.6, 3.2.7 a 3.2.8 lze nalézt na přiloženém CD v souboru
/programs/fsm/allRestrictedApproximateSeed.cpp.

4.1.3 Generování řetězců

Implementace algoritmu 3.4.1, generující řetězce s mírou opakování, je realizována jako program beroucí vstupní parametry z příkazové řádky a vygenerovaný text vypisuje na standardní výstup. Vstupní parametry jsou shodné s popisem, tedy délka základní sekvence, počet opakování, počet změn a použitá abeceda. Použitá abeceda se zadává jako první a poslední znak z abecedy. Abecedu tvoří znaky ležící mezi prvním a posledním znakem včetně. Pořadím se myslí pořadí tak, jak je v ASCII tabulce.

Plus k potřebným parametrům je přidán jeden navíc. Tím je „seed“ pro náhodný generátor. Tento generátor slouží k provedení náhodných operací. Jako generátor je použit generátor ze standardní knihovny cstdlib (neboli stdlib.h). Tento generátor není příliš kvalitní co se týče náhodnosti, protože je předvídatelný, dá se popsat matematickou formulí. Pro mě je tato vlastnost spíše výhodou, protože výsledky lze replikovat při znalosti počátečního nastavení.

Realizace algo. 3.4.1 lze nalézt na přiloženém CD v souboru
/programs/generator/genSeed.cpp.

4.2 Způsob měření

Měření jednotlivých metrik zajišťuje samotný spustitelný program, zapouzdřující implementaci algoritmu. Ten program načte příslušnou část datového souboru do operační paměti. Na té potom pustí testovaný algo.

Algoritmy ukládají výsledky do společné struktury, která uchovává nalezené výsledky. Ty můžeme po dokončení výpočtu zobrazit nebo spočítat. Ale pokud nás konkrétní výsledky nezajímají, lze struktuře říct, aby si nic nepamatovala.

Další metriky, tedy dobu běhu a použitou paměť, program měří pomocí systémového volání *int getrusage(int who, struct rusage *usage);* popsanou v manuálové stránce [5]. Volání se provede před a po dokončení výpočtu jader. Tím získáme čas doby běhu pouze pro výpočet jader. Toto měření má problém s měřením vícevláknových programů, kde se časy běhů vláken sčítají, ale to není náš případ.

Při měření použité paměti pomocí tohoto volání dostaneme maximální velikost použité paměti. Tedy při volání na začátku výpočtu dostaneme paměť potřebnou pro uložení vstupních dat. Voláním po dokončení dostaneme paměť potřebnou pro výpočet a uložení vstupních dat. Pokud tedy odečteme počáteční od konečné, dostaneme paměť potřebnou pouze pro výpočet jader.

Kapitola 5

Experimenty

Všechny algoritmy, jak už bylo zmíněno, byly implementovány v programovacím jazyce C++. Byly použity standardní knihovny tohoto jazyka včetně knihovny šablon STL. Pro kompilaci byl použit překladač GCC ve verzi 4.7.2 s optimalizacemi O3. Jako operační systém sloužil Debian 7.1. Experimenty poháněl 64 bitový procesor Intel Celeron T3300 na 2,00 GHz. Velikost operační paměti byla omezena na 3,5 GB.

5.1 Zopakování měření, která byla publikována

Prvním experimentem, který jsem provedl, bylo zopakování měření, která byla uvedena v pracích popisujících použité algoritmy [3], [4]. Hlavním důvodem tohoto experimentu bylo zjistit, zda se má implementace nějak neodlišuje od těch publikovaných. Kdyby tomu tak nebylo, kdyby se výsledky zásadně lišily, odhalilo by to případnou chybu nebo nedokonalost mé implementace. Dalším hlavním výsledkem takovéhoto experimentu je porovnání obou přístupů.

Možností tohoto porovnání je fakt, že obě měření prezentovaná v pracích mých předchůdců, byla provedena na stejných datech. Jako data byla použita chromozom kvasinky (Saccharomyces cerevisiae S288c chromosome IV) dostupný z [6]. Tato data jsem použil i já, pro opětovné otestování. Jako délka vstupů při těchto experimentech, je počet znaků od počátku sekvence.

5.1.1 Konečný automat

V kapitole 4.1.2 jsem popsal zmenšení paměťových nároků a to tak, že si nebudeme pamatovat *lfactor* každého stavu, ale budeme jej dopočítávat. Jelikož má první implementace toto zlepšení neobsahovala a přidal jsem ji až po naměření sady dat, můžeme porovnat vliv tohoto zlepšení.

Jak lze vidět na obrázcích 5.1 a 5.2, tato změna výrazně pomůže se spotřebou paměti a spotřebuje pouze málo času navíc.

První srovnání je zaostřeno na časovou náročnost, tedy potřebný čas na vyřešení úlohy v závislosti na délce vstupu. Ty závislosti jsou vyneseny při maximální dovolené závislosti k pro hledaná jádra. Jak ukazují obrázky 5.3 a 5.4, tak mé časy jsou horší, ale to je způsobeno méně výkonným hardwarem použitým při mých experimentech.

Druhé srovnání potom zkoumá jak ovlivňuje potřebný čas změna dovolené maximální vzdálenosti. Tentokráte je tato závislost měřena při konstantní délce vstupu. Toto srovnání je poté na obr. 5.5 (původní), a na obr. 5.6 (mé). Při tomto srovnání je propad ve výkonnosti ještě markantnější. Propad teď už nejspíš nezpůsobuje jenom slabší hardware, ale i nedokonalost mého řešení, která je popsána níže.

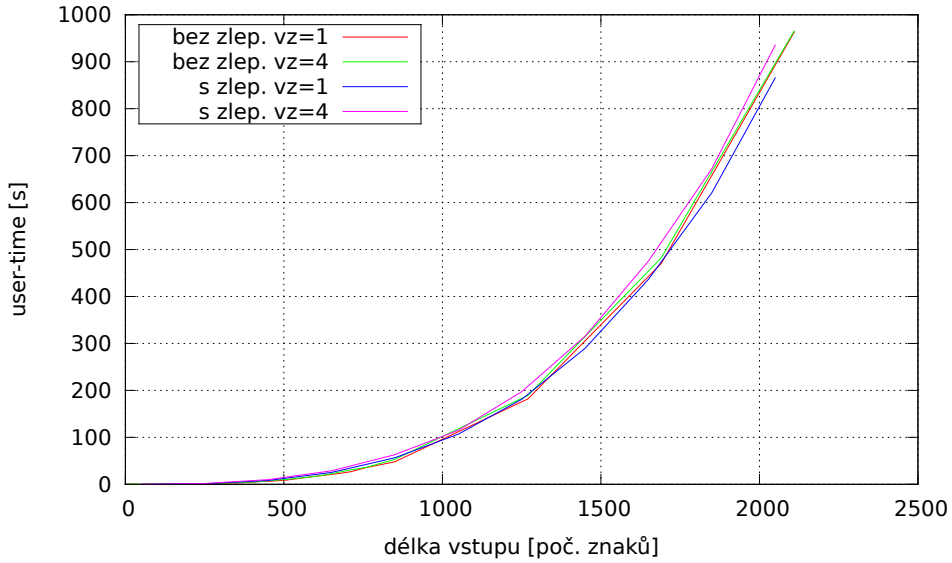
Poslední srovnání porovnává potřebnou paměť v závislosti na délce vstupních dat a konstantní maximální dovolené vzdálenosti. Když tedy srovnáme původní výsledky, obr. 5.7 a můj obr. 5.8, potom vidíme, že obě implementace jsou po paměťové stránce srovnatelné.

Poslední obrázek 5.9 v této sekci zobrazuje závislost využité paměti na změně max. dovolené vzdálenosti. Ta, jak je popsána v tabulce 3.1, je na této vlastnosti nezávislá. Ale jak ukazuje obr. 5.9 s rostoucí max. dovolenou vzdáleností, roste i použitá paměť. To je v rozporu s předpokladem, měla by tam být konstantní přímka v určité hladině. To je způsobeno tím, že si ve své implementaci pamatuji nedeterministické verze automatů, ale to není potřeba. Jejich existenci lze nahradit funkcí. Bohužel na toto zlepšení jsem přišel později a už nezbyl čas ji realizovat.

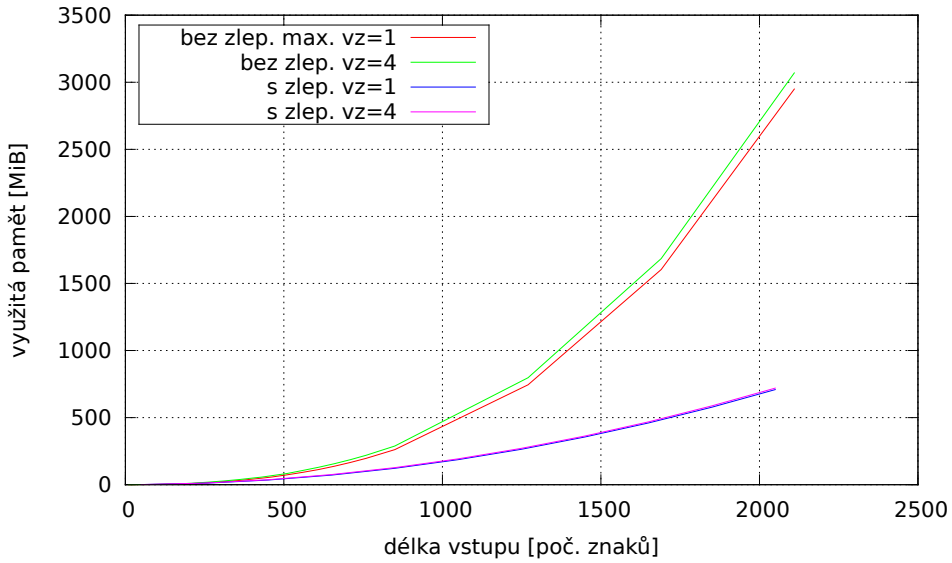
5.1.2 Algoritmus používající dynamického programování

Na rozdíl od práce, ve které byl popsán předchozí algoritmus, práce popisující tento algoritmus, je poněkud chudší co se týče naměřených závislostí. Nejspíše i proto, že ovlivnění tohoto algo. je možné pouze pomocí délky vstupu. Zde je uvedena pouze závislost potřebného času na délce vstupních dat, a to v rozsahu od 10 do 120 znaků.

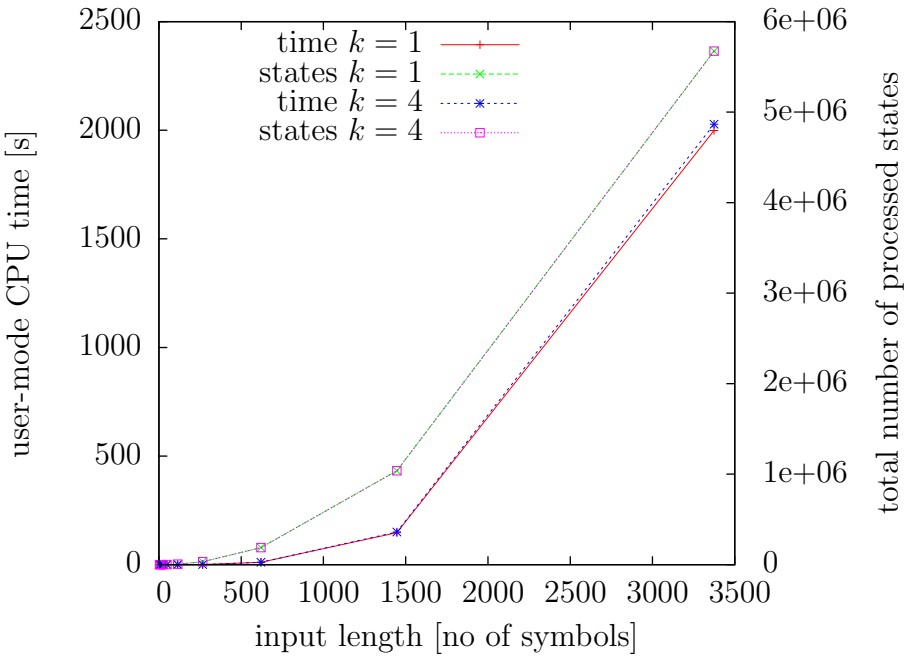
Obr. 5.10 (původní) a obr. 5.11 (můj) zobrazují tvarově podobnou křivku závislosti, ale trvání výpočtu je výrazně odlišné. Tato odlišnost je způsobena tím, že výkonnost použitých systémů je různá. Pro ilustraci moji předchůdci používali Pentium-4M 1,7GHz. Na obr. 5.11 není křivka příliš vyhlazená, obsahuje spoustu zubů. To je způsobeno tím, že měřené časy jsou zde malé a jsou ovlivněny činností systému.



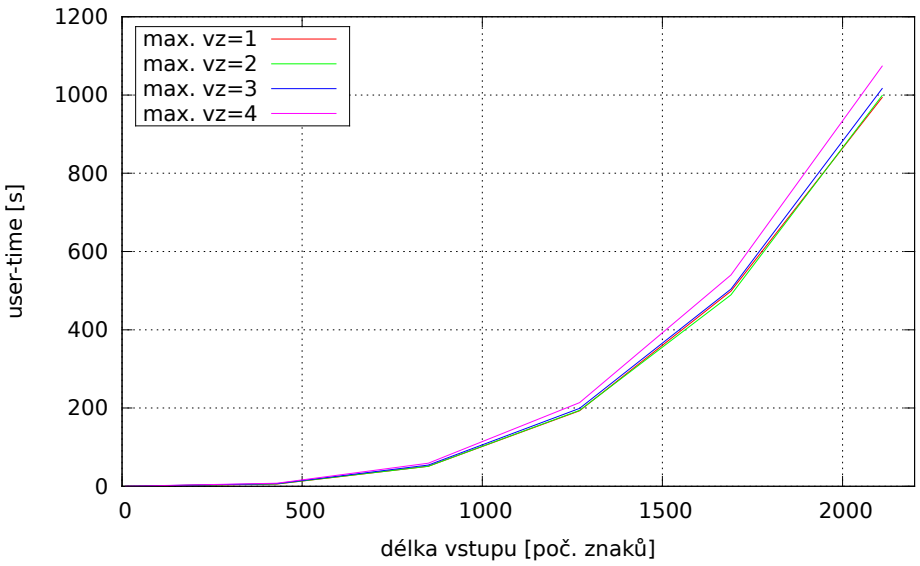
Obrázek 5.1: Demonstrace vlivu zlepšení práce s *lfactor*em na časovou složitost



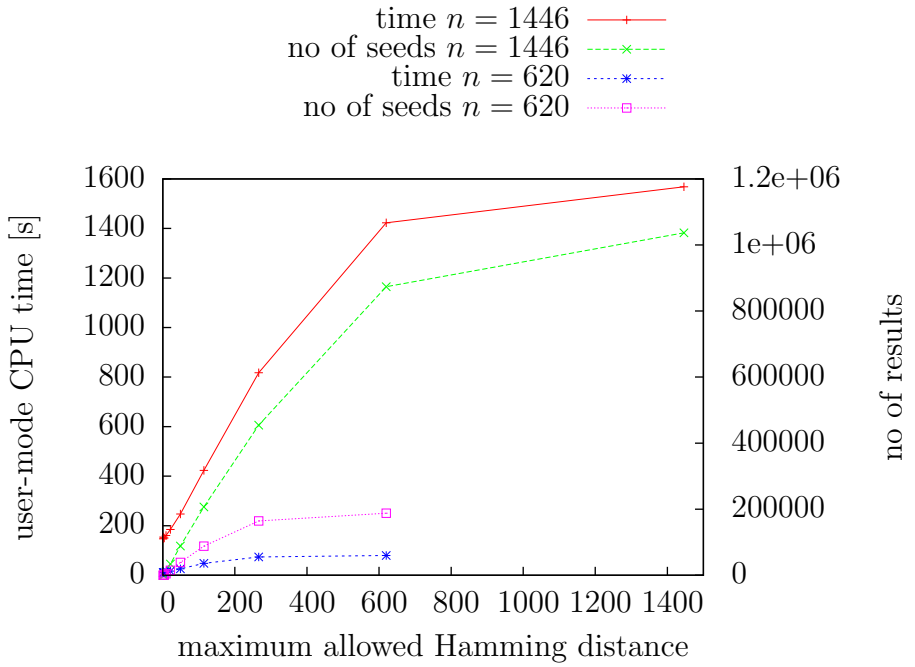
Obrázek 5.2: Demonstrace vlivu zlepšení práce s *lfactor*em na paměťovou složitost



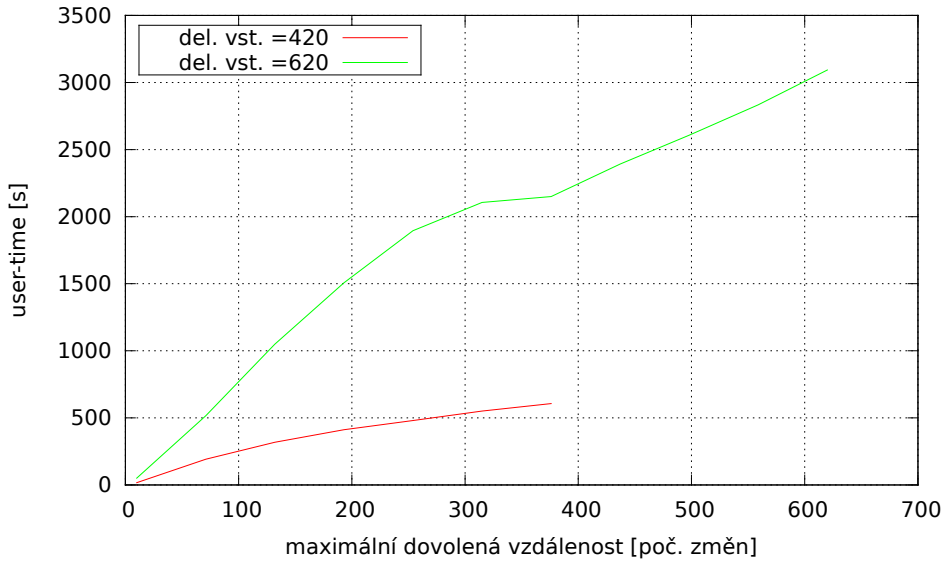
Obrázek 5.3: Závislost potřebného času na délce dat pro algo. s KA, prezentovaný v [4, str. 97]



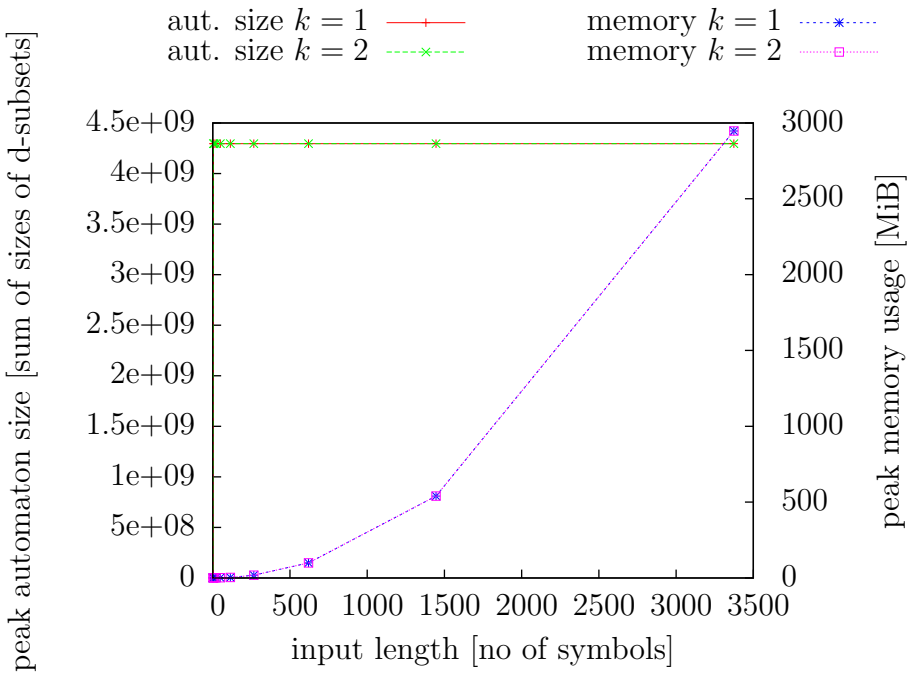
Obrázek 5.4: Závislost potřebného času na délce dat pro algo. s KA, implementovaný mnou



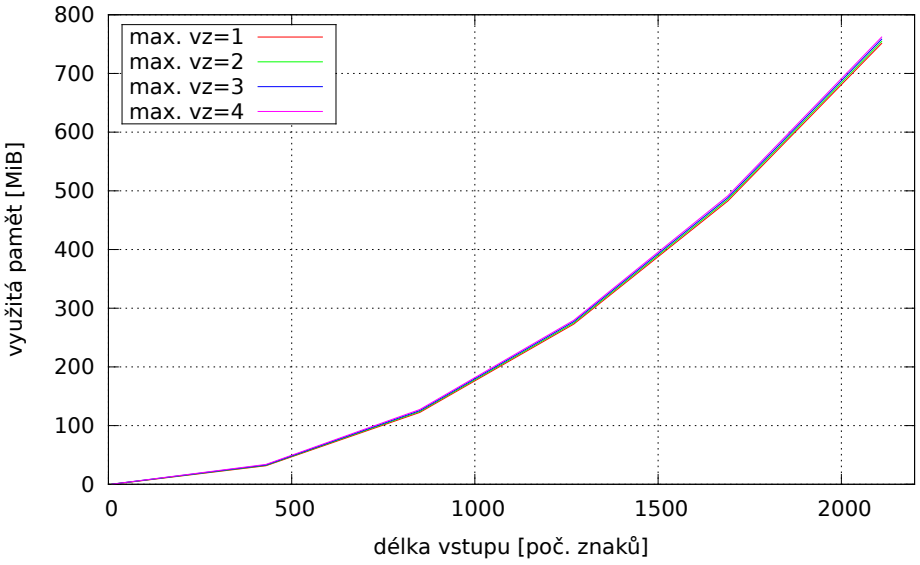
Obrázek 5.5: Závislost potřebného času na maximální dovolené vzdálenosti pro algo. s KA, prezentovaný v [4, str. 98]



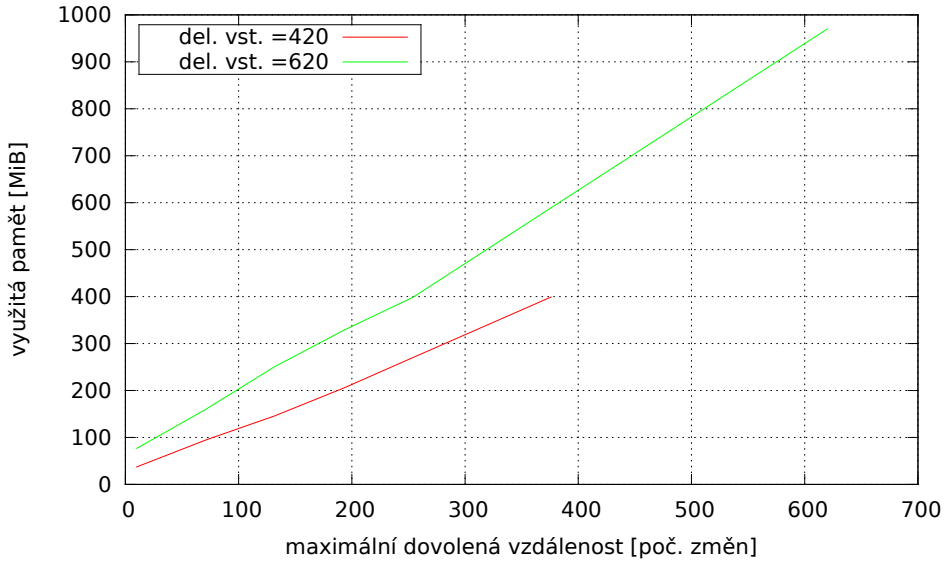
Obrázek 5.6: Závislost potřebného času na maximální dovolené vzdálenosti pro algo. s KA, implementovaný mnou



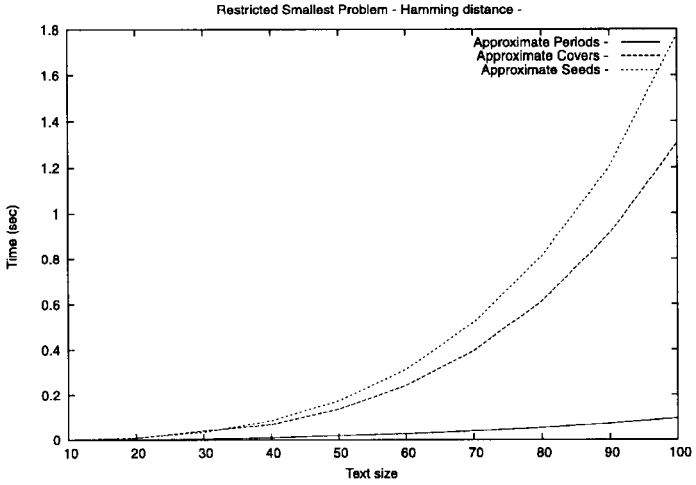
Obrázek 5.7: Závislost potřebné paměti na délce vstupu pro algo. s KA, prezentovaný v [4, str. 99]



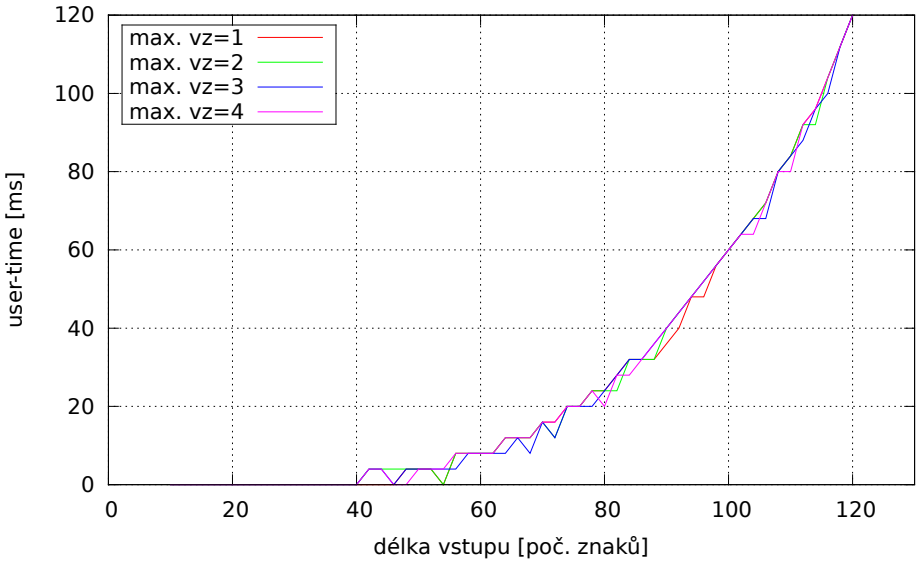
Obrázek 5.8: Závislost potřebné paměti na délce vstupu pro algo. s KA, implementovaný mnou



Obrázek 5.9: Závislost potřebné paměti na maximální dovolené vzdálenosti pro algo. s KA, implementovaný mnou



Obrázek 5.10: Závislost potřebného času na délce vstupu pro algo. používající dyn. programování, prezentovaný v [3, str. 864]



Obrázek 5.11: Závislost potřebného času na délce vstupu pro algo. používající dyn. programování, implementovaný mnou

5.2 Srovnání obou přístupů

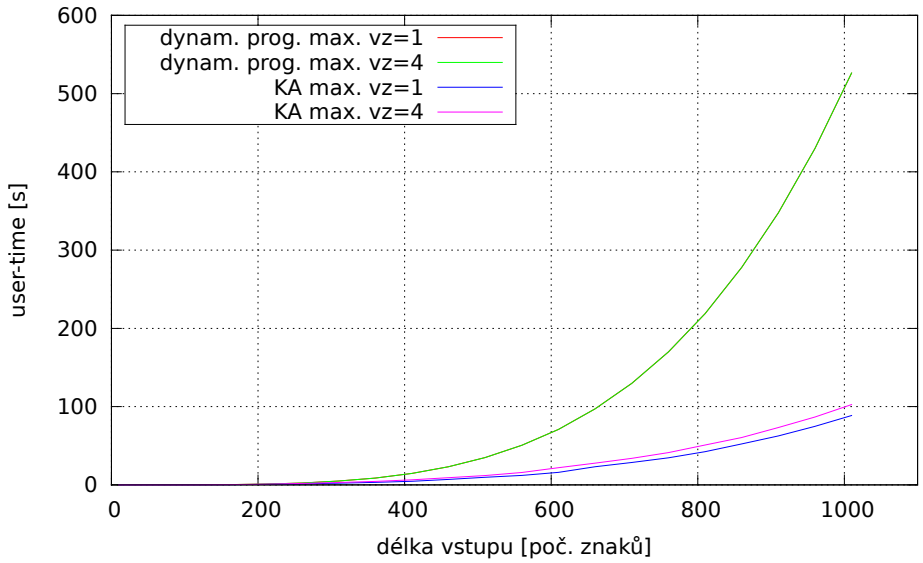
Nyní oba algoritmy porovnáme. Zde uvedené srovnání vychází z dat naměřených v předchozí kapitole. Jako zdroj dat byla použita sekvence DNA chromozomu kvasinky [6].

První sada dat pozoruje vztah potřebného času a paměti na měnící se délce dat při konstantní dovolené vzdálenosti. Délka dat se pohybuje od 10 do 1010 znaků od začátku dat. Maximální dovolená vzdálenost je 1 a 4. Druhá sada dat pozoruje vztah potřebného času a paměti na měnící se dovolené vzd. při konstantní délce. Délky jsou 420 a 620 znaků. Dovolená vzd. se pohybuje od 10 do délky vstupu.

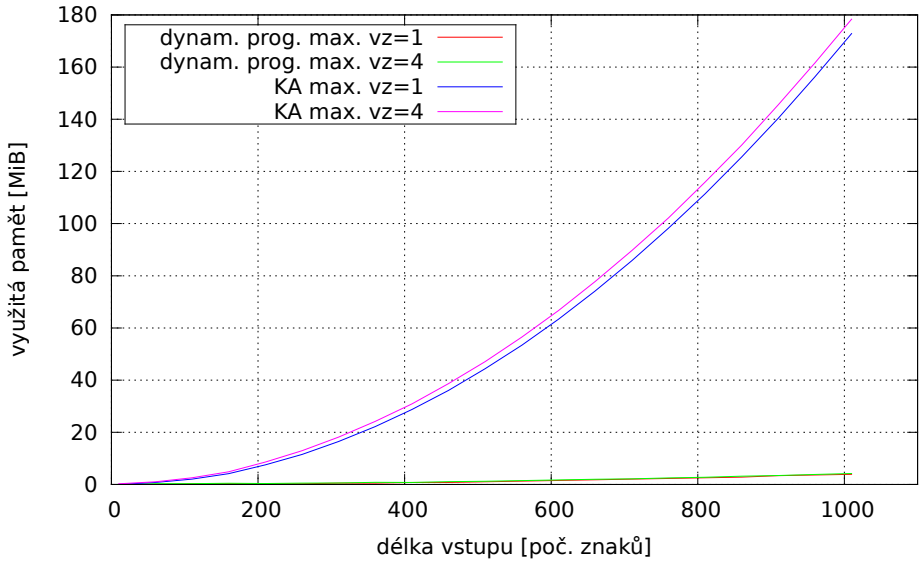
Účelem je postavit oba algo. vedle sebe, aby bylo jednoduché porovnat jejich vlastnosti. Asymptotické složitosti říkají, že algo. jsou stejně dobré, ale jak ukazuje srovnání, skutečnost je trochu jiná.

Na obrázcích 5.12 a 5.13 je dobře vidět, jak oba algoritmy vynikají v odlišných vlastnostech. Algo. používající KA je rychlejší než ten druhý. Rozdíl v rychlosti je asi pětinásobný ve prospěch algo. používající KA. Toto ilustruje obrázek 5.12. Oproti tomu, jak nám ilustruje obr. 5.13, ve stránce paměťové náročnosti vyniká algo. používající dyn. programování. Propad je zde mnohem markantnější, než v případě rychlosti. Rozdíl se pohybuje okolo 40 až 60 násobku ve prospěch dyn. programování. Důvodem tohoto je fakt, že dyn. programování potřebuje pouze matici znaků. Oproti tomu si přístup s KA pamatuje čtyři různé automaty.

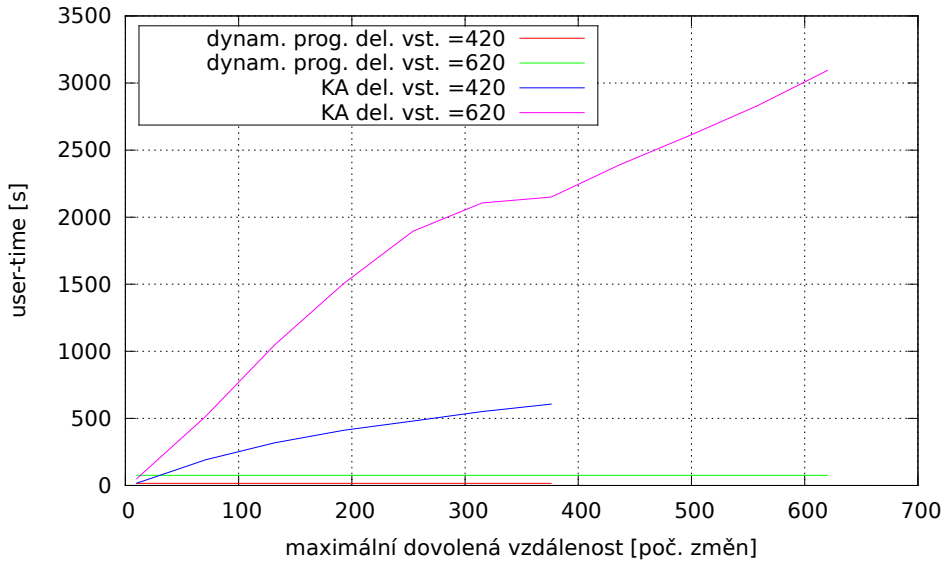
Rychlost přístupu s KA, pomíjí se zvětšující se dovolenou vzdáleností. Obr. 5.14 to ilustruje více něž názorně. Propad je rapidní, ale to je z toho důvodu, že složitost u algo. s KA je lineární v závislosti na dovolené vzdálenosti. Kdežto u algo. používající dyn. programování je složitost konstantní v závislosti na dovolené vzd. Obr. 5.15 jenom podtrhuje fakt, že paměťová náročnost u algo. s KA je vyšší než u algo. s dyn. programováním.



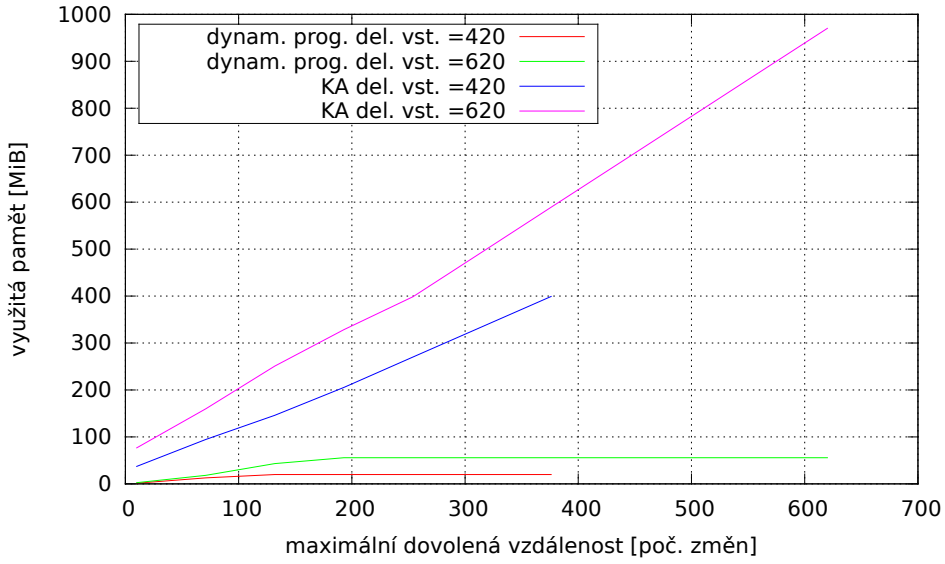
Obrázek 5.12: Srovnání obou přístupů na závislosti délky vstupu a potřebného času



Obrázek 5.13: Srovnání obou přístupů na závislosti délky vstupu a potřebné paměti



Obrázek 5.14: Srovnání obou přístupů na závislosti maximální dovolené vzdálenosti a potřebného času



Obrázek 5.15: Srovnání obou přístupů na závislosti maximální dovolené vzdálenosti a potřebné paměti

5.3 Datová citlivost

Další podstatnou vlastností, kterou algo. mají, je citlivost či necitlivost na zkoumaná data. Tím se myslí, že lze pro dva různé vstupy dostat odlišné složitosti. Tato změna složitosti asi nenastane na úrovni asymptotické složitosti, taková změna by byla uvedena v literatuře. Nějaké kolísání na úrovni multiplikativní konstanty by bylo možné.

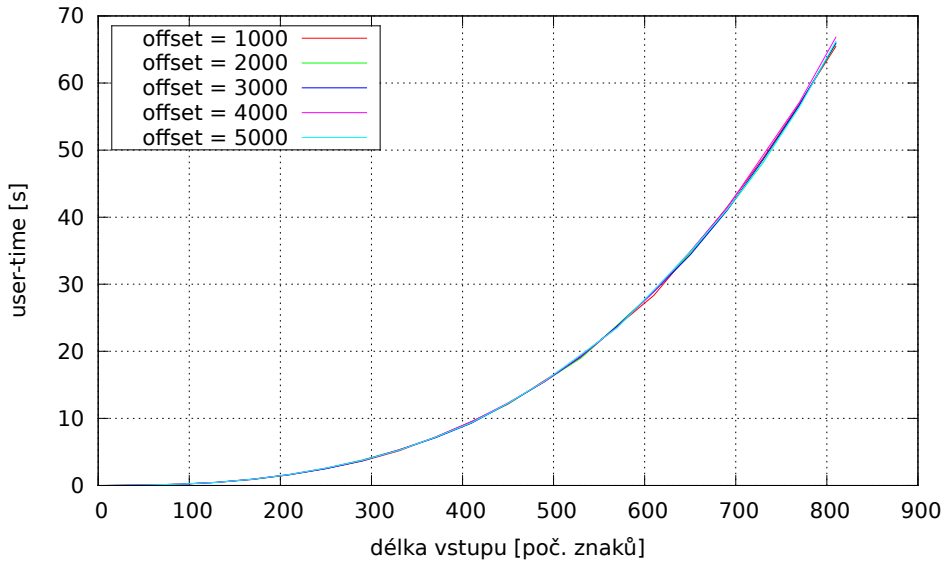
K měření jsem použil opět DNA, ale nyní jsem použil různé začátky dat. Takto dostanu data se stejnou abecedou, ale s různými daty. Měřil jsem spotřebovaný čas a paměť v závislosti na délce vstupu v intervalu od 10 do 810 znaků. A to pro data začínající na pozici 1000, 2000, 3000, 4000, 5000 a konstantní maximální dovolenou vzdálenost 5.

Obr. 5.16, 5.18 velmi pěkně demonstrují, že časová složitost obou algoritmů je datově necitlivá. Křivky naměřených hodnot leží přímo na sobě.

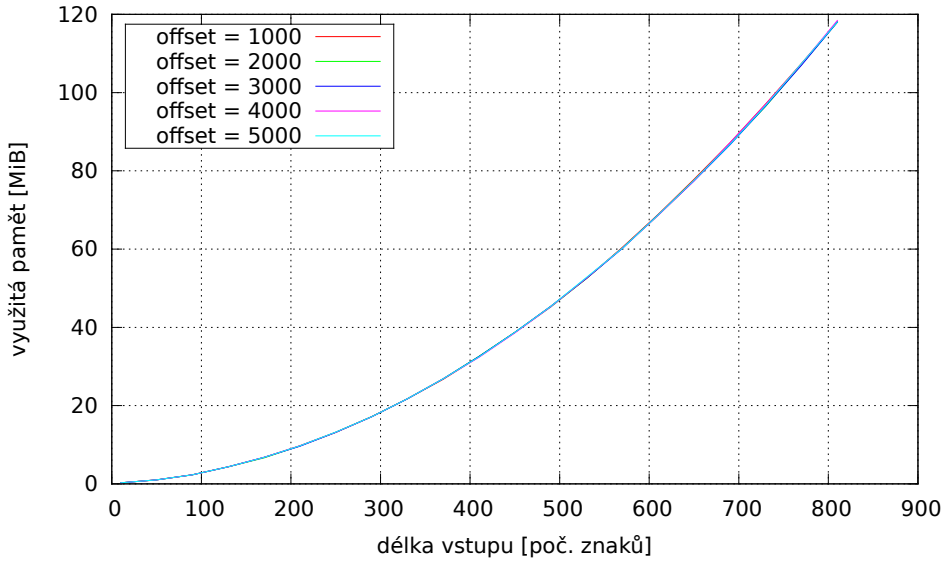
Obr. 5.17 a 5.19 poukazuje na to, že paměťová složitost je také datově nezávislá. To, co bych měl na obr. 5.19 vysvětlit, je schodovitost potřebné paměti. To je způsobeno tím, že operační systém nepřiděluje přesně požadovanou paměť, ale množství zaokrouhlené na nejbližší větší velikost bloku. Proto má závislost tvar schodů, potřebná paměť na dvě po sobě jdoucí vzdálenosti se vejde do stejného počtu bloků přidělených operačním systémem.

Ale když jsem toto tvrzení o datové necitlivosti otestoval na datech generovaných mým generátorem, výsledek se změnil. Pomocí generátoru jsem vytvořil tři sady dat. Generátor vytvářel řetězce nad abecedou číslic (0 až 9) s délkou základní sekvence 100 znaků, s dvaceti opakováními této sekvence a s deseti změnami v základní sekvenci. Počáteční nastavení náhodného generátoru byla: 2015 pro sadu 1, 456789 pro sadu 2 a 72164 pro sadu 3. Vynesené závislosti jsou měřeny pro délku vstupních dat od 10 do 610 znaků. Maximální dovolená vzdálenost byla během testů konstantní na hodnotě 5.

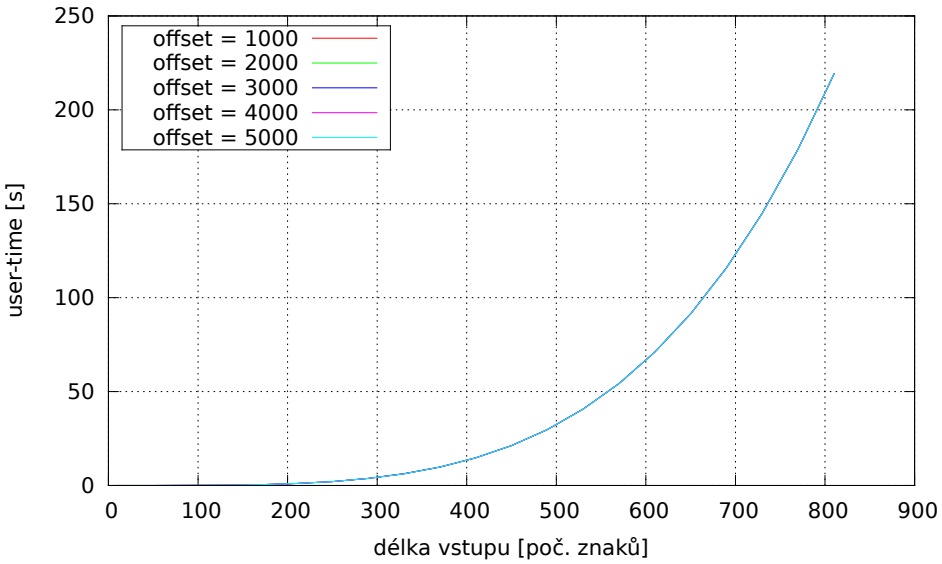
U algoritmu, který používá dynamické programování se výsledek potvrdil. To ilustruje obrázek 5.20 a 5.21. Ale u algoritmu, který používá KA je výsledek odlišný. Na obrázku 5.22, který zobrazuje závislost potřebného času na délce vstupu, je vidět rozdíl mezi sadami 1 a 2 oproti sadě 3. Rozdíl se projevuje po celou dobu experimentu a nepatrně roste. Ve výsledku jsem tedy vyvrátil předchozí tvrzení a algo. s KA je datově citlivý. Tato změna není na úrovni změny asymptotické složitosti, ale pouze na úrovni multiplikativní konstanty. Paměťová náročnost se nezměnila, přestože časová náročnost ano, jak ukazuje obr. 5.23.



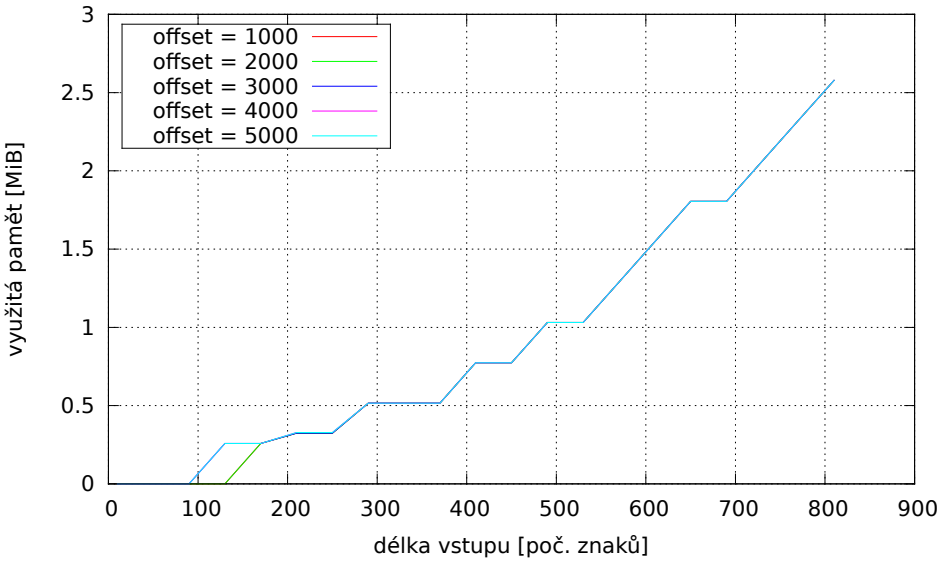
Obrázek 5.16: Algo. s KA: Závislost potřebného času na délce vstupu při konstantní maximální dovolené vzdálenosti a různých počátcích dat



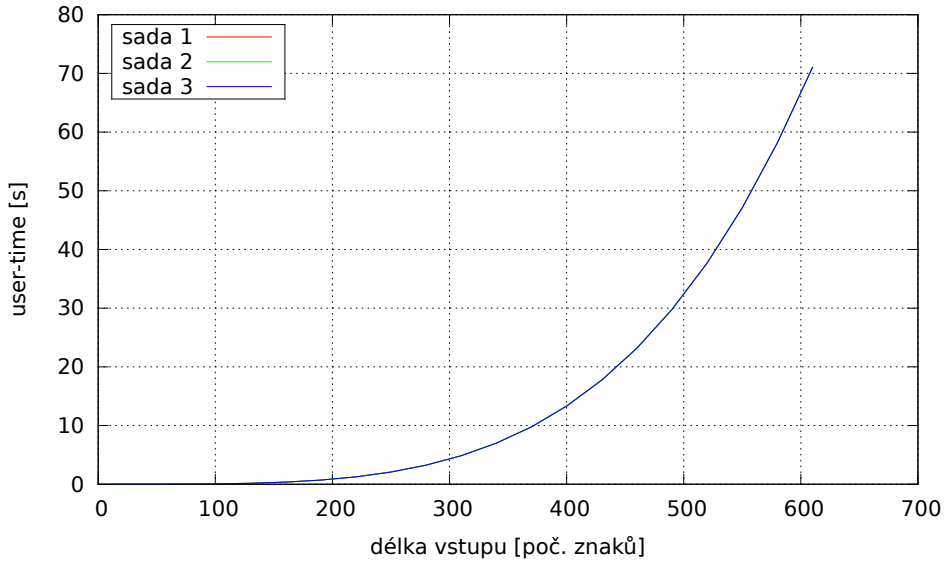
Obrázek 5.17: Algo. s KA: Závislost použité paměti na délce vstupu při konstantní maximální dovolené vzdálenosti a různých počátcích dat



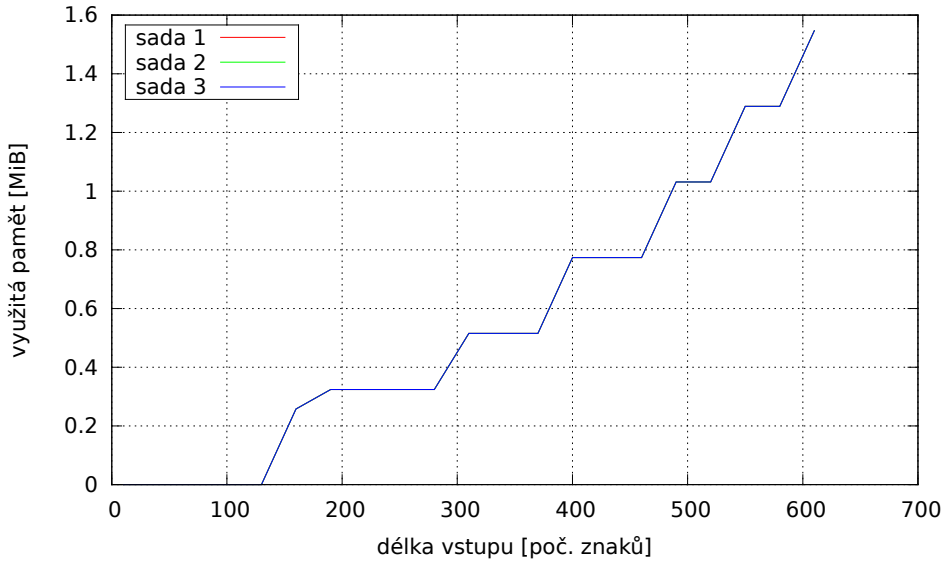
Obrázek 5.18: Algo. používající dyn. prog.: Závislost potřebného času na délce vstupu při konstantní maximální dovolené vzdálenosti a různých počátcích dat



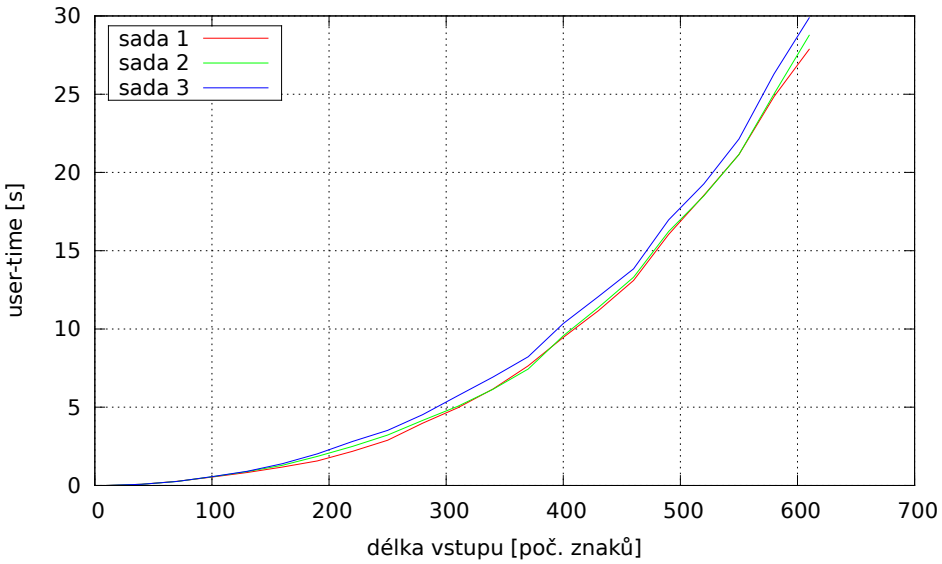
Obrázek 5.19: Algo. používající dyn. prog.: Závislost použité paměti na délce vstupu při konstantní maximální dovolené vzdálenosti a různých počátcích



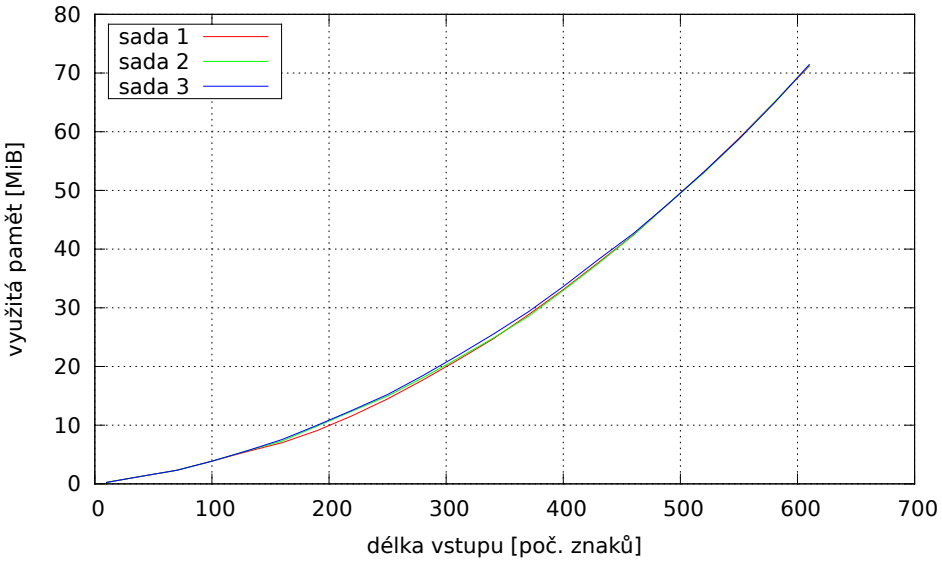
Obrázek 5.20: Algo. používající dyn. prog.: Závislost potřebného času na délce vstupu při konstantní maximální dovolené vzdálenosti a různých datových sadách



Obrázek 5.21: Algo. používající dyn. prog.: Závislost použité paměti na délce vstupu při konstantní maximální dovolené vzdálenosti a různých datových sadách



Obrázek 5.22: Algo. s KA.: Závislost potřebného času na délce vstupu při konstantní maximální dovolené vzdálenosti a různých datových sadách



Obrázek 5.23: Algo. a KA: Závislost použité paměti na délce vstupu při konstantní maximální dovolené vzdálenosti a různých datových sadách

5.4 Vliv velikosti vstupní abecedy

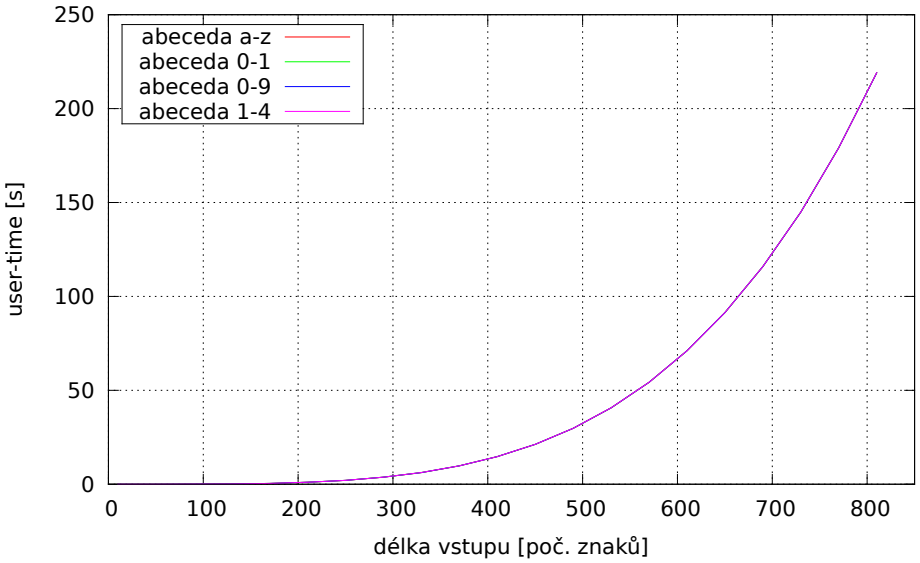
Dále jsem prozkoumal vliv různě velkých abeced na časovou a paměťovou náročnost obou sledovaných algo. Abych byl schopen testovat tento vliv na algo., použil jsem můj generátor řetězců. Důvodem byl nedostatek vhodných dat. Pro velikost abecedy 4 existuje dostatek datových sad, např. jakákoli sekvence DNA. Pro abecedu číslic (10 znaků) jich je také mnoho, např. iracionální konstanty jako π nebo e, ale hledat v nich pravidelnosti není úplně vhodné, protože už ze své podstaty opakování neobsahují. A další abecedy jsou na tom podobně, těžko se hledají.

V tomto experimentu jsem měřil závislost časové a paměťové složitosti na délce vstupních dat. Rozsah délek byl od 10 do 810 znaků a maximální dovolená vzdálenost byla 5.

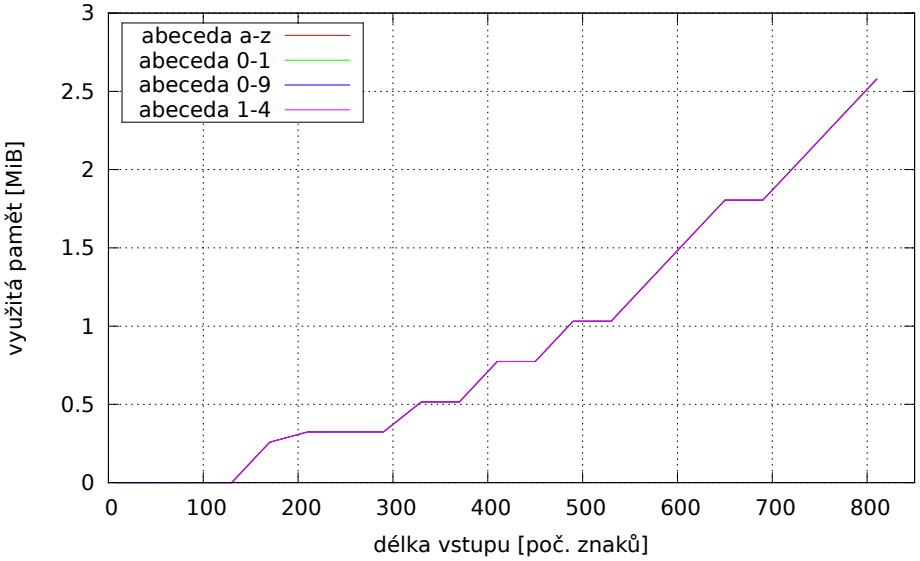
Jako použité abecedy jsem zvolil binární abecedu (0, 1), čtyř znakovou abecedu (1–4), číslice (0–9) a písmena anglické abecedy (a–z). Délka základní sekvence byla 30 znaků, která se 300krát opakovala a změn bylo 10. A také počáteční nastavení generátoru bylo pro všechny čtyři sady stejné na hodnotě 159874.

Algo. používající dyn. programování by podle tabulky 3.1 neměl být tímto parametrem ovlivněn. Proto, když se podíváme na obrázky 5.24, který zobrazuje vztah časové náročnosti na délce vstupu pro různé abecedy a 5.25, který zobrazuje vztah paměťové náročnosti na délce vstupních dat pro různé abecedy, vidíme tuto nezávislost. Ta se projevuje tím že vidíme pouze jednu křivku.

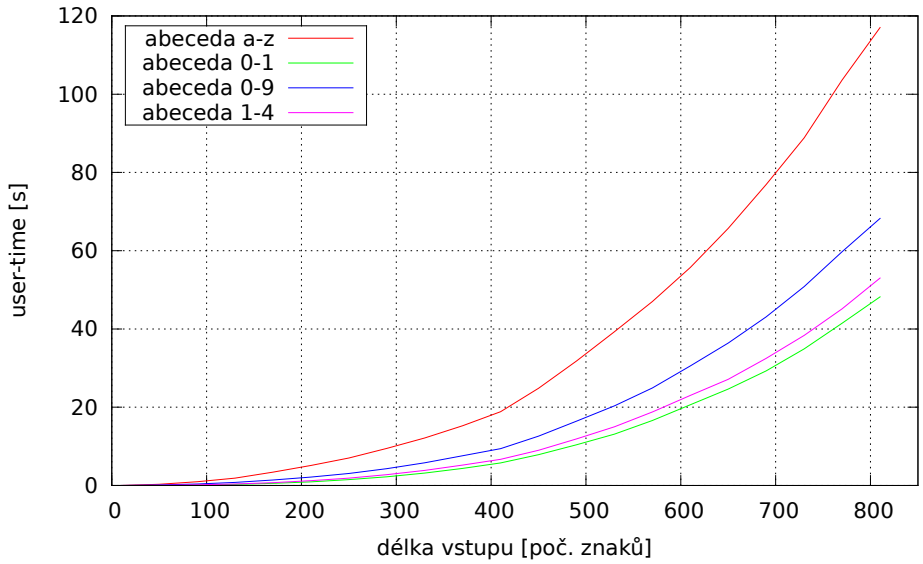
U algo. s KA je časová složitost závislá na velikosti vstupní abecedy lineárně. A paměťová složitost by měla být nezávislá na velikosti vstupních dat. Na obr. 5.26, který zobrazuje závislost potřebného času na vstupních datech, je patrný nárůst náročnosti spolu s rostoucí velikostí abecedy. Podobně tak jako roste potřebný čas, tak roste celková spotřebovaná paměť (viz obr. 5.27). To by bylo v rozporu s předpokladem, ale když se podíváme na závislost počtu stavů na délce vstupních dat, obr. 5.28, tak tuto závislost neukazuje. Tedy nárůst potřebné paměti je způsoben tím, že při zvětšení abecedy vzroste i počet přechodů v každém stavu. S nárůstem počtu přechodů rostou i paměťové nároky na každý stav, proto je tedy potřebná paměť závislá na velikosti abecedy.



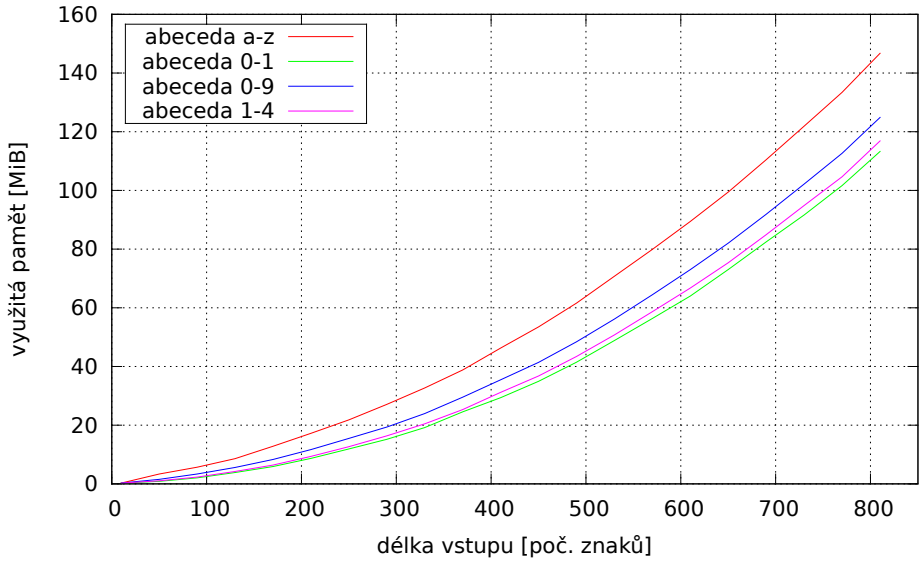
Obrázek 5.24: Algo. používající dyn. programování: Závislost potřebného času na délce vstupu při konstantní maximální dovolené vzdálenosti a různých abecedách



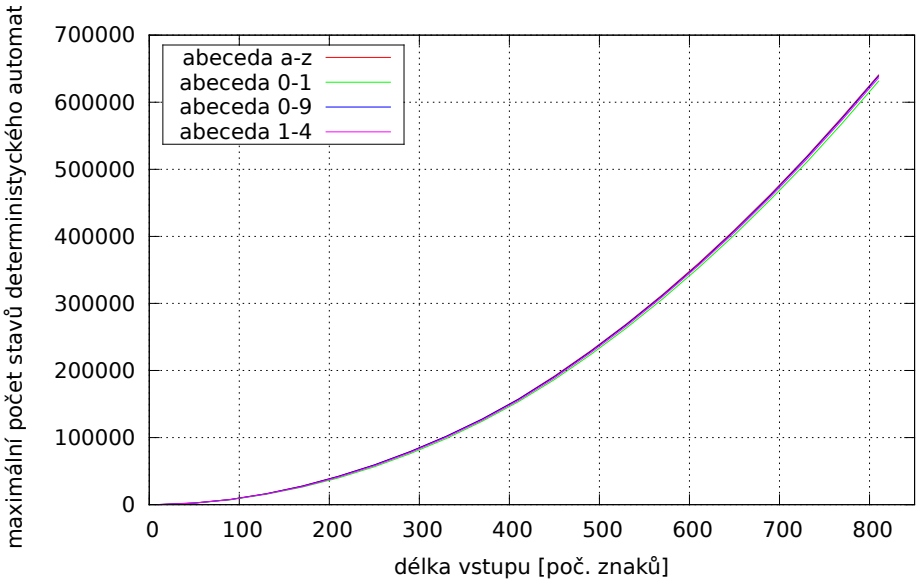
Obrázek 5.25: Algo. používající dyn. programování: Závislost potřebné paměti na délce vstupu při konstantní maximální dovolené vzdálenosti a různých abecedách



Obrázek 5.26: Algo. s KA: Závislost potřebného času na délce vstupu při konstantní maximální dovolené vzdálenosti a různých abecedách



Obrázek 5.27: Algo. s KA: Závislost potřebné paměti na délce vstupu při konstantní maximální dovolené vzdálenosti a různých abecedách



Obrázek 5.28: Algo. s KA: Závislost maximálního počtu stavů na délce vstupu při konstantní maximální dovolené vzdálenosti a různých abecedách

Závěr

V této bakalářské práci jsem se zabýval zkoumáním algoritmů na hledání jader řetězců. Jádro je jedna z pravidelností, dalšími jsou například periody nebo pokrytí. Přesnou formulaci problému, kterým jsem se zabýval naleznete v definici 2.4.5 na stránce 9. Na řešení tohoto problému existují dva algoritmy, které jsou velmi rozdílné. Jeden je založený na dynamickém programování a ten druhý používá konečné automaty.

Oba tyto algoritmy jsem ve své práci popsal. K popisu jsem použil přirozený jazyk, kterým jsem se snažil popsat činnost, fungování a myšlenky algoritmů. Tento popis jsem doplnil i pseudokódem, který formálně popisuje činnost jednotlivých algoritmů. Popis algo. s dynamickým programováním lze nalézt v kapitole 3.1 a popis algo. s konečnými automaty lze nalézt v kapitole 3.2. K popisu algoritmu patří také uvedení asymptotické složitosti, kterou v této práci naleznete v kapitole 3.3, a to pro oba algoritmy současně.

K tomu abych mohl zkoumat vlastnosti výše zmíněných algoritmů, oba jsem je implementoval. Popis implementace, tak jak jsem ji provedl já, naleznete v kapitole 4.1. Samotné programy realizující algoritmy lze potom nalézt na přiloženém CD. Při testování, zda obě implementace fungují stejně, jsem přišel na chybu v algoritmu s konečnými automaty. Tato chyba se projevovala určením špatné minimální vzdálenosti jádra. Navrhl jsem opravu této chyby, kterou jsem konzultoval s vedoucím práce a autorem algoritmu v jedné osobě, který s opravou souhlasil.

Další částí testování jsou data, která jsou použita pro testy. Použil jsem jednak sekvenci DNA chromozomu kvasinky a také data generovaná generátorem, jehož funkce je popsána v kapitole 3.4.

Po provedení experimentů můžeme konstatovat následující zjištění. Srovnáním obou přístupů se ukázalo, že každý vyniká v jiných parametrech. Algo. používající dyn. programování vyniká na poli potřebné paměti. Na druhou stranu v rychlosti vyniká algo. s konečnými automaty. Ale jak ukázal test zkoumající vliv maximální dovolené vzdálenosti, jeho výhoda se s rostoucí dovolenou vzdáleností zmenšuje. Datová citlivost se projevila pouze u algo.

s konečnými automaty. U algo. používající dynamické programování se tato citlivost během testování neprojevila. Při testování vlivu abecedy se oba algo. projevily obdobně jako při předchozím testování. Abeceda měla vliv pouze na algo. s konečnými automaty, a to jak na čas potřebný pro dokončení úlohy, tak i na potřebnou paměť. U algo. používající dynamické programování se citlivost na změnu abecedy nijak neprojevila.

Experimenty nám tedy odhalily výhody obou přístupů. Algoritmus používající dynamické programování ovlivňuje pouze délka vstupu jinak nic. Tato výhoda je vykoupena pomalejším výpočtem než v případě druhého algoritmu. Druhý algoritmus, ten s konečnými automaty, na druhou stranu spotřebuje velké množství paměti a je závislý i na ostatních parametrech, tak i na konkrétních datech.

Literatura

[1] Christodoulakis, M.; Iliopoulos, C. S.; Park, K.; aj.: Approximate Seeds of Strings. *Journal of Automata, Languages and Combinatorics*, ročník 10, č. 5/6, 2005: s. 609–626.

[2] Holub, J.: *Operace s konečnými automaty [Přednáška]*. Praha: FIT ČVUT, zimní semestr 2013/2014.

[3] Christodoulakis, M.; Iliopoulos, C. S.; Park, K.; aj.: Implementing Approximate Regularities. *Mathematical and computer Modelling*, ročník 42, 2005: s. 855–866.

[4] Guth, O.: *Searching Regularities in Strings using Finite Automata*. Dizertační práce, Czech Technical University in Prague, Faculty of Information Technology, Department of Theoretical Computer Science, 2014.

[5] *getrusage - get resource usage [Linuxová manuálová stránka]*. 09 2010 [cit. 2014-12-10].

[6] Saccharomyces cerevisiae S288c chromosome IV, complete sequence [Online]. National Center for Biotechnology Information, [cit. 2014-11-20]. Dostupné z: `http://www.ncbi.nlm.nih.gov/nuccore/NC_001136.10`

Příloha **A**

Seznam použitých zkratek

algo. Algoritmus

vzd. Vzdálenost

dyn. Dynamické

prog. Programování

DKA Deterministyský konečný aotomat (viz definice 2.5.1).

NKA Nedeterministický konečný automat (viz definice 2.5.12).

ε**-NKA** Nedeterministický konečný automat s ε přechody (viz definice 2.5.14).

KA Konečný automat (viz definice 2.5.19)

Příloha **B**

Obsah přiloženého CD

```
readme.txt ..... stručný popis obsahu CD
programs ..... adresář s implementací
data ..... adresář s použitými daty
result ..... adresář s výsledky měření
text ..... adresář s této práce
    BP_Juna_Martin.pdf ..... text práce ve formátu PDF
    src ..... adresář se zdrojovými kódy této práce
```