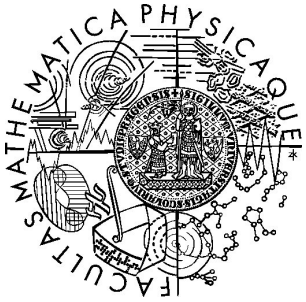
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Jindřich Šedek

Konstrukční algoritmy pro sufixové datové struktury

Kabinet software a výuky informatiky

Vedoucí bakalářské práce: RNDr. Tomáš Dvořák, CSc.
Studijní program: Informatika, programování

2007

Děkuji svému vedoucímu RNDr. Tomáši Dvořákovi, CSc. a Mgr. Martinu Senftovi za jejich ochotu a čas, který mi věnovali při analýze algoritmů.

Prohlašuji, že jsem svou bakalářskou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

V Praze dne Jindřich Šedek

Obsah

Název práce: Konstrukční algoritmy pro sufixové datové struktury
Autor: Jindřich Šedek
Katedra (ústav): Kabinet software a výuky informatiky
Vedoucí bakalářské práce: RNDr. Tomáš Dvořák, CSc.
e-mail vedoucího: Tomas.Dvorak@mff.cuni.cz

Abstrakt: Directed Acyclic Word Graph (DAWG) je prostorově úsporná datová struktura, která slouží k ukládání přípon řetězců. Compact Directed Acyclic Word Graph (CDAWG) je ještě úspornější variantou DAWG. Jejich hlavní uplatnění je v hledání vzorků uvnitř rozsáhlých řetezců. Tato práce je zaměřena na implementaci několika známých konstrukčních algoritmů těchto datových struktur. Otestoval jsem je na různé druhy vstupních dat a porovnal jejich vlastnosti. Konkrétně jsem se zajímal o Blumerův algoritmus na konstrukci DAWG [1], Crochemorův algoritmus na konstrukci CDAWG [2] a Inenagův algoritmus na konstrukci CDAWG [3].

Klíčová slova: datové struktury, DAWG, CDAWG, Blumerův algoritmus, Crochemorův algoritmus, Inenagův algoritmus

Title: Suffix data structures construction algorithms
Author: Jindřich Šedek
Department: Department of Software and Computer Science Education
Supervisor: RNDr. Tomáš Dvořák, CSc.
Supervisor's e-mail address: Tomas.Dvorak@mff.cuni.cz

Abstract: Directed Acyclic Word Graph (DAWG) is a space efficient data structure used for storing suffixes of strings. Compact Directed Acyclic Word Graph (CDAWG) is a more space efficient variant of DAWG. Their main use is in searching short patterns in a huge amount of data. This work is aimed at an implementation of few construction algorithms of these data structures. It compaires characteristics of Blumer et. al algorithm for DAWG construction [1], Crochemore algorithm for CDAWG construction [2] and Inenaga algorithm for CDAWG construction [3].

Keywords: data structures, DAWG, CDAWG, Blumer algorithm, Crochemore algorithm, Inenaga algorithm

Kapitola 1

Úvod

Při řešení klasického problému hledání výskytu podřetězce v textu se snažíme rozhodnout, zda se podslovo x vyskytuje v textu w. K řešení tohoto problému existuje velké množství algoritmů. Většina z nich si nejprve předpřipraví hledané slovo x a poté prochází text w. Tyto algoritmy často řeší daný problém v lineárním čase vzhledem k velikosti w, protože čas vynaložený na přípravu x je většinou zanedbatelný vzhledem k velikosti w.

Uvedená metoda je však zcela nevhodná pro hledání velkého množství krátkých vzorků v pevně daném textu. Opakované hledání zde vyžaduje opakované procházení celého textu a tím se zvyšuje čas hledání. Některé algoritmy dokáží hledat větší počet vzorků paralelně, ale vyžadují znalost všech hledaných vzorků předem, což není vždy splnitelné. V tomto případě je mnohem lepší pokusit se o předpřípravu daného textu do vhodné datové struktury a poté vyhledávat jednotlivé vzorky v lineárním čase vzhledem k velikostem hledaných vzorků.

Za tímto účelem byly prozkoumány datové struktury sufixový trie a sufixový strom. Nevýhodou obou těchto struktur je jejich redundance. O odstranění této redundance se snaží datová struktura nazývaná DAWG (Directed Acyclic Word Graph). Je to konečný automat, který přijímá všechny přípony slova w. DAWG je schopen rozhodnout o výskytu slova x v textu w v čase úměrném velikosti x a jeho velikost je shora omezena velikostí w. Modifikací DAWGu lze získat datovou strukturu zvanou CDAWG (Compact Directed Acyclic Word Graph), která ještě více snižuje velikost potřebné paměti.

Mimo hledání podřetězců nacházejí datové struktury DAWG a CDAWG své uplatnění v mnoha dalších aplikacích. Mezi další aplikace patří například určení počtu různých podslov daného slova nebo nalezení nejdelšího opakujícího se podřetězce v textu.

V bioinformatice jsou DNA sekvence velmi často chápány jako slova nad abecedou nukleotidů {a, c, d, t}. V tomto pojetí se stávají předmětem zájmu lingvistické a statistické analýzy. Pro tyto účely je sufixový strom velmi užitečnou datovou struk-

turou, ale jeho nevýhodou je poměrně vysoká paměťová náročnost. I na tomto místě nacházejí DAWG a CDAWG své uplatnění.

Tato práce se zabývá třemi nejvýznamnějšími algoritmy na konstrukci DAWGu a CDAWGu, které jsem implementoval ve dvou programovacích jazycích a následně testoval na různých vstupních datech. Na závěr uvádím jejich vzájemné srovnání a srovnání provedených implementací.

Kapitola 2

Základní pojmy

Nejprve zavedeme několik základních pojmů o poté na jejich základě zadefinujeme datové struktury DAWG a CDAWG.

Abeceda je neprázdná množina znaků.

Slovo a **text** jsou konečné posloupnosti znaků abecedy Σ. Pro přehlednost budeme pojmem text obvykle označovat delší posloupnost znaků než slovo a budeme ho používat ve významu vstupu pro konstrukci datových struktur. Slovo budeme používat ve významu kratší posloupnosti znaků uvnitř textu. Při indexaci jednotlivých znaků slova budeme i-tý znak slova x značit $x[i]$ a $|x|$ budeme označovat délku slova x.

Řekneme, že slovo y je **podslovo** slova x, pokud existuje index i takový, že $y = x[i] \dots x[i + |y| - 1]$.

Přípona slova x je posloupnost znaků $x[i] \dots x[n]$, kde n udává délku slova x.

End-set slova x v textu w budeme nazývat množinu

$$\text{end-set}_w(x) := \{i : x = w[i - |x| + 1] \dots w[i]\}.$$

Řekneme, že slova x a y jsou ekvivalentní vzhledem k w, pokud

$$\text{end-set}_w(x) = \text{end-set}_w(y).$$

Třídu ekvivalence podle této relace označujeme $[x]_w$.

w =	k	a	k	a	o
	1	2	3	4	5

$$\text{end-set}_w(ka) = \text{end-set}_w(a) = \{2, 4\} \Rightarrow [ka]_w = [a]_w$$

Tato definice ekvivalence na slovech splňuje následující vlastnosti:

- pokud jsou dvě slova ekvivalentní, pak jedno z nich je nutně příponou druhého
- slova xy a y jsou ekvivalentní, právě tehdy když se před každým výskytem slova y nachází slovo x

On-line nazveme takový konstrukční algoritmus, který je schopen převést datovou strukturu pro text w na datovou strukturu pro text wa, kde a je libovolný znak vstupní abecedy Σ. Tato vlastnost umožňuje vstupní text dále prodlužovat, aniž by bylo nutné provádět rekonstrukci celého automatu.

Directed Acyclic Word Graph (DAWG) je datová struktura tvořená orientovaným grafem. Vrcholy grafu reprezentují třídy ekvivalence všech podslov vstupního textu w. Množinu vrcholů definujeme:

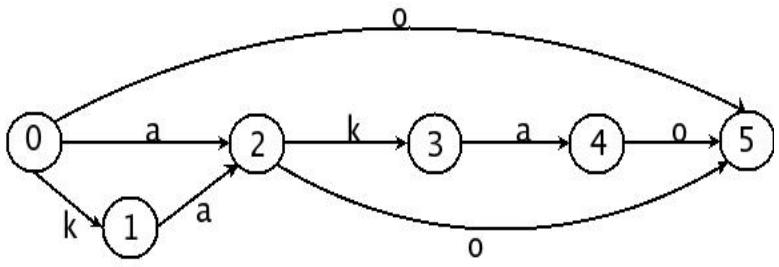
$$\{[x]_w \mid x \, je \, podslovem \, w\}.$$

Kořen (počáteční vrchol) grafu je reprezentant třídy ekvivalence $[\lambda]_w$, kde λ udává prázdné slovo. Množinu hran definujeme:

$$\{[x]_w \xrightarrow{a} [xa]_w \mid x, \; xa \, jsou \, podslova \, w, [x]_w \neq [xa]_w\}.$$

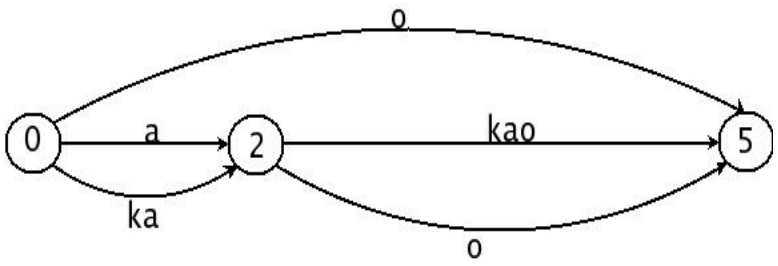
Hrana z vrcholu s do vrcholu t existuje, právě tehdy když s je reprezentantem nejdelší vlastní předpony slova reprezentovaného t.

Pro potřeby vyhledávání podslov lze DAWG charakterizovat jako částečný konečný automat nad danou vstupní abecedou Σ. Stavy odpovídají vrcholům grafu, počátečním stavem je kořen, všechny stavy jsou přijímací a přechodová funkce je dána hranami grafu.



Obrázek 2.1: DAWG pro slovo "kakao"

Počet stavů automatu je roven počtu tříd ekvivalence $[x]_w$ pro všechna podslova x slova w. Ten je nejvýše roven $2n-1$, kde n je délka slova w [1]. Protože počet hran je shora omezen $3n-4$, je velikost DAWGu lineární vzhledem k délce vstupního slova



Obrázek 2.2: CDAWG pro slovo "kakao"

w. Čas potřebný k jeho konstrukci je též lineární a navíc Blumerův algoritmus je schopen jej sestavovat on-line, tedy postupně bez znalosti celého vstupního slova na počátku konstrukce [1].

Compact Directed Acyclic Word Graph (CDAWG) vznikne z DAWGu vynecháním stavů výstupního stupně jedna a příslušným přeznačením hran z jednotlivých znaků na odpovídající řetězce. $|h|$ budeme označovat délku řetězce, který odpovídá hraně h.

Definice nám přímo dává návod, jak sestrojit CDAWG. Nejprve zkonstruujeme DAWG a poté odstraněním stavů stupně jedna provedeme kompakci na CDAWG [2]. Nepříjemností této konstrukce je skutečnost, že nejprve potřebujeme velké množství paměti na konstrukci DAWGu a až poté můžeme část paměti uvolnit při kompakci.

Prvním přímým konstrukčním algoritmem pro CDAWG je Crochemorův algoritmus [2]. Tento algoritmus však není schopen provádět konstrukci on-line a vyžaduje znalost celého vstupního řetězce na počátku konstrukce. Tuto nevýhodu odstraňuje až Inenagův algoritmus [3].

Tail(w) bude označovat nejdelší příponu slova w, která se ve w nachází více, než jedenkrát.

$$\text{tail(vodovod)} = \text{vod},\ \text{tail(kokos)} = \lambda,\ \text{tail(aaa)} = \text{aa}.$$

V textu budeme dále používat slovní označení pro dva význačné stavy. **Zdrojem** budeme nazývat stav reprezentující třídu ekvivalence $[\lambda]_w$, kde λ udává prázdné slovo. Zdroj označuje vstupní stav automatu. **Stokem** budeme nazývat stav reprezentující třídu $[w]_w$. Stok je jediný stav DAWGu i CDAWGu z něhož nevedou žádné hrany.

Kapitola 3

Blumerův algoritmus

3.1 Popis algoritmu

Blumerův algoritmus prezentovaný v [1] je prvním známým algoritmem pro přímou konstrukci DAWGu. Tento algoritmus je schopen konstruovat DAWG *on-line*. Tato vlastnost vyžaduje, aby byl automat v každém kroku konstrukce korektní pro již zpracovaný prefix textu w. Algoritmus je inicializován prázdným slovem λ, kterému odpovídá jednostavový automat. V i-tém kroku algoritmus předpokládá, že již má sestavený korektní DAWG pro text $w[1] \dots w[i-1]$ a provádí pouze změny nutné k přechodu na korektní DAWG pro text $w[1] \dots w[i]$.

Aby byl algoritmus schopen rozhodnout, jaké změny má při přidávání i-tého znaku provádět, používá při konstrukci dvě pomocné informace. První se týká hran a udává, zda je daná hrana *primární*, či *sekundární*. Hrana vedoucí do stavu x je *primární*, pokud leží na nejdelší cestě ze *zdroje* do stavu x, v opačném případě je hrana *sekundární*. Druhý údaj nezbytný k běhu algoritmu je funkce, která ke každému stavu udává jeho *předchůdce* v *podmnožinovém stromě stavů*.

Podmnožinový strom stavů je strom daný relací $\subseteq$ na množině stavů DAWGu. Stavy DAWGu reprezentují třídy ekvivalence podslov vstupního textu w, každý stav tedy reprezentuje množinu podslov w. Pokud je množina podslov reprezentovaná stavem p podmnožinou množiny reprezentované stavem q, pak vede v *podmnožinovém stromě stavů* hrana z p do q.

Důkaz, že relace $\subseteq$ tvoří strom na množině stavů DAWGu vychází z vlastností relace *end-set* a lze jej naleznout v [1]. Díky tomu, že třída $[\lambda]_w$ je podmnožinou $[x]_w$ pro libovolné x platí, že kořenem tohoto stromu je *zdroj*.

Přidávání nového znaku a začíná vytvořením nového stavu, který se *primární* hranou připojí za původní *stok* a stane se tak novým *stokem* reprezentujícím třídu ekvivalence $[wa]_{wa}$.

Dále algoritmus doplňuje znak a na konec všech přípon aktuálně reprezentovaných automatem. Pomocí funkce *předchůdce* prochází automatem z původního *stoku* směrem ke kořeni a doplňuje sekundární a-hrany do nového *stoku*. Toto provádí tak dlouho, dokud nedorazí do *zdroje*, nebo dokud nenalezne stav, který již a-hranu má. Taková hrana vede do stavu reprezentujícího třídu $[tail(wa)]_w$.

Je-li to hrana *primární*, její koncový stav reprezentantuje také třídu $[tail(wa)]_{wa}$. Je tedy označen jako *předchůdce* nově vzniklého stavu a vzniklý automat je korektní DAWG pro text wa.

Pokud je to hrana *sekundární*, je nutné její cílový stav duplikovat, převézt tuto hranu na *primární* a přesměrovat ji do duplikátu. Při duplikování je dále nutné vytvořit kopie všech hran vedoucích z původního stavu a nastavit jejich počáteční stav na nově vzniklý duplikát. Navíc je nutné nastavit *předchůdce* původního stavu jako *předchůdce* duplikátu, původní stav jako *předchůdce* nového *stoku* a duplikát jako *předchůce* původního stavu. Na závěr je nutné všechny *sekundární* hrany, které směřují do původního stavu, přesměrovat do duplikátu.

Speciální případ nastane, pokud algoritmus dorazí až do *zdroje*, ale nenalezne žádný stav mající a-hranu. V tomto případě je jako *předchůdce* nového *stoku* nastaven *zdroj*.

Přesný popis Blumerova algoritmu včetně velmi podrobného pseudokódu je možné naleznout v [1], a proto v této práci pseudokód neuvádím.

3.2 Implementace

Blumerův algoritmus jsem implementoval ve dvou variantách. První variantou je implementace v jazyce C bez použití složitějších datových struktur. Druhou implementaci jsem provedl v jazyce C++ s užitím Standard Template Library (STL) a kontejneru *map*. *Map* je asociativní datová struktura implementována červenočerným stromem, která dokáže v logaritmickém čase k danému klíči naleznout jemu přiřazená data. Modifikující operace insert a delete v této datové struktuře probíhají též v logaritmickém čase.

3.3 Implementace v jazyce C

Implementaci v jazyce C se snažím optimalizovat na vysokou rychlost pro malé abecedy. Z možných reprezenentací hran jsem mezi polem, spojovým seznamem, hašovací tabulkou apod. zvolil pole, protože umožňuje nejrychlejší přístup k jednotlivým prvkům a pro malé abecedy jeho redundance není příliš vysoká. Spojový seznam by způsobil zpomalení přístupu k prvkům a ukazatele použité k udržení spojových seznamů by též představovaly určité paměťové nároky. Hašování by bylo možné realizovat dvěma různými přístupy. Buď hašovat hrany pro jednotlivé stavy samostatně,

nebo hašovat hrany všech stavů do jedné společné tabulky. První přístup by vyžadoval hledání vhodných funkcí pro perfektní hašování. Druhý přístup by zase přinášel problémy s řešením kolizí.

Stavy automatu jsou ukládány v paměti v jejich pořadí vzniku a jejich adresace probíhá pomocí tohoto pořadí. Každý stav automatu je reprezentován polem o $1+|\Sigma|$ prvcích, kde každý prvek je n-bitové číslo. První prvek udává *předchůdce* daného stavu. V následujícím seznamu prvků vždy i-tý reprezentuje hranu označenou i-tým znakem abecedy Σ. Dolních $n-1$ bitů každého prvku udává pořadí stavu, do něhož hrana vede. Nejvyšší bit každého prvku zaznamenává, zda je jím reprezentovaná hrana *primární*, nebo *sekundární*. Pokud je i-tý prvek roven nule, pak ze stavu nevede žádná hrana označená i-tým písmenem Σ.

Jako prvky pole jsem se rozhodl použít datový typ *int*, který vzhledem k použité 32bitové architektuře odpovídá 32bitovému číslu. Bylo by jistě možné použít i menší čísla, pokud by byl jejich rozsah dostatečný pro adresaci všech stavů. Například použitím typu *small*, který na použité architektuře odpovídá 16bitovému číslu, by na indexaci zbývalo 15 bitů (první bit rozhoduje u každé hrany zda je primární či sekundární), což odpovídá 32 768 indexovaných stavů a délka vstupního řetězce by tedy nesměla přesáhnout 16 000 znaků (na jeden vstupní znak může mít DAWG až dva stavy viz. Kapitola 2). Použití vícebitových čísel by v tomto případě nemělo smysl, protože počet indexovaných stavů by výrazně převýšil velikost adresovatelné paměti. Pro 64bitové počítače by však i tato možnost přicházela v úvahu.

Použitá reprezentace umožňuje přístup k hraně v konstantním čase. Nalezení jejího cílového stavu je též v konstatním čase, protože vyžaduje pouze aritmetické operace přepočtení pořadí stavu na jeho adresu v paměti. Konstrukce této struktury vyžaduje v nejhorším případě čas $O(|w|)$, kde w je vstupní řetězec. Hledání v této struktuře vyžaduje čas nejvýše $O(|x|)$, kde x je hledaný řetězec. Obě operace jsou nezávislé na velikosti vstupní abecedy Σ, která však musí být předem známa.

Problémovým bodem této konstrukce je její paměťová složitost. Vzhledem k tomu, že počet stavů DAWGu je nejvýše $2 * |w|$, je jeho prostorová složitost v nejhorším případě $8*(1+|\Sigma|)*|w|$ bytů, což může být pro větší abecedy Σ nepříjemné. Druhou slabou stránkou této reprezentace je nutnost znát velikost abecedy Σ již na začátku konstrukce, aby bylo možné alokovat dostatečný prostor jednotlivým stavům. Oběma problémům se snažím předejít následující implementací v C++.

3.4 Implementace v jazyce C++

Při implementaci v jazyce C++ se snažím snížit požadavky na paměťovou náročnost reprezentace hran užitím složitějších datových struktur, než je pole. K reprezentaci hran je použit kontejner *map*, jehož klíčem je znak, který danou hranu popisuje a jeho hodnotou je ukazatel na stav, v němž daná hrana končí. Stav je reprezentován

třídou, která obsahuje ukazatel na *předchůdce* a dvě mapy hran. První mapa slouží pro hrany *primární* a druhá pro hrany *sekundární*. Použitím dvou různých map se vyhýbám nutnosti zaznamenávat u hrany příznak zda je *primární* či *sekundární*.

```
// mapa ukazatelů na stav reprezentující množinu hran
  typedef std::map<unsigned char, TStav*> THrany;

// třída reprezentující stav automatu
  class TStav{
  public:
    // inicializační konstruktor
      TStav();
    // ukazatel na předchůdce vrcholu
      TStav * Zpetny;
    // primární a sekundární hrany vycházející z vrcholu
      THrany Primarni, Sekundarni;
  };
```

Mezi kontejnery nabízenými STL je *map* jediný asociativní kontejner. Pro neasociativní kontejnery by bylo nutné k reprezentaci hran vytvářet třídu, která by obsahovala `znak` a `ukazatel` na cílový stav hrany, popřípadě ještě příznak *primární* či *sekundární* hrany. Kontejner *map* je pro danou situaci nejpřirozenější.

Použitá reprezentace už nedává možnost přistoupit k hraně v konstatním čase, protože hledání v kontejneru *map* probíhá v čase $O(\log |\Sigma|)$. Navíc je nutné zaplatit určitou prostorovou daň za implementační požadavky kontejneru způsobenou udržováním červeno-černého stromu.

Na druhou stranu použitá reprezentace nabízí možnost alokovat pro jednotlivé stavy pouze paměť úměrnou počtu hran z nich vycházejících a není potřebné alokovat prostor úměrný velikosti Σ. Tento fakt je poměrně významný, protože dle experimentálních výsledků (kapitola 6) z každého stavu vede v průměru jedna až dvě hrany. Další výhodou této reprezentace je, že nevyžaduje znát velikost vstupní abecedy Σ na počátku konstrukce, protože Blumerův algoritmus tuto skutečnost pro svou funkčnost nevyžaduje a použitý kontejner *map* dynamicky přizpůsobuje svou velikost aktuálním požadavkům.

Kapitola 4

Crochemorův algoritmus

4.1 Popis algoritmu

Crochemorův algoritmus prezentovaný v [2] je prvním přímým lineárním algoritmem na konstrukci CDAWGu. Vychází z McCreightova algoritmu [5] konstrukce sufixového stromu a využívá některé postupy používané při konstrukci DAWGu. Hlavní myšlenkou konstrukce CDAWG(w) je postupné vkládání jednotlivých přípon textu w od nejdelší (celé w) po nejkratší (poslední znak w). Z tohoto vyplývá, že již pro první krok algoritmu je nutné znát celý vstupní text w a algoritmus tedy není schopen pracovat *on-line*.

Pro potřeby konstrukce používá Crochemorův algoritmus, stejně jako Blumerův, dvě pomocné informace. Pro každý stav potřebuje uchovávat délku nejdelší cesty, která do něho vede ze *zdroje*. Tuto délku budu, stejně jako v [2], pro stav p označovat *length(p)*. Druhou informací je *přechůdce* stavu v *podmnožinovém stromě stavů*, který je definován stejně jako předcházející kapitole. Dále bude hrana α ze stavu p do stavu q označována **solid**, pokud $length(p) + |\alpha| = length(q)$.

Na počátku konstrukce je automat inicializován *zdrojem* a *stokem*. Algoritmus postupuje iterativně a v i-tém kroku přidává do již sestavené části automatu i-tou příponu w, tedy řetězec $w[i] \dots w[n]$. Tento hlavní cyklus algoritmu zachovává následující dva invarianty:

I1: Na počátku i-tého kroku jsou všechny přípony w, které jsou delší než i-tá přípona, cestami v dosud sestaveném automatu.

I2: Na počátku i-tého kroku všechny stavy automatu odpovídají nejdelším vlastním společným předponám dvojic přípon delších než i-tá přípona.

Na počátku i-tého kroku při vkládání přípony $w[i] \dots w[n]$ algoritmus nejprve nalezne, pomocí funkce *SlowFind*, nejdelší předponu slova $w[i] \dots w[n]$, která se již v automatu nachází. Funkce *SlowFind* postupně prochází automatem a znak po

znaku porovnává přidávanou příponu s řetězci na hranách. V případě, že *SlowFind* přichází do stavu p po hraně, která není *solid*, pak je nutné stav p duplikovat. Pokud by *SlowFind* vždy začínal ze *zdroje*, by byl nucen v nejhorším případě projít všechny přípony v celé jejich délce a časová složitost algoritmu by se tak stala kvadratickou. Pro zajištění linearity *SlowFind* nezačíná vždy ze *zdroje*, ale z *předchůdce* naposledy navštíveného stavu.

Pokud *SlowFind* skončí své hledání ve stavu, z něhož nevede žádná hrana, po které by mohl dále pokračovat, stačí z tohoto stavu přidat hranu do *stoku* a tím doplnit chybějící zbytek přípony. Jiná situace nastane, pokud algoritmus dorazí do stavu, ze kterého pokračuje po hraně α, ale do dalšího stavu již nedorazí. V tomto případě je nutné vytvořit nový stav, hranu α jím rozdělit a přidat hranu z nově vytvořeného stavu do *stoku*, čímž dojde k doplnění chybějícího zbytku přípony.

Pro každý nově vzniklý stav je nutné nalézt jeho *předchůdce*. Tato situace nastává, je-li stav ve *SlowFind* duplikován, nebo pokud dochází k rozdělení hrany při přidávání nové přípony. V obou situacích algoritmus používá funkci *FastFind*.

Nechť q je nově vzniklý stav a α je hrana z p vedoucí do q. *FastFind* začíná v *předchůdci* stavu p a nalezne hranu β začínající stejným písmenem jako hrana α. Dokud je α delší než β, *FastFind* pokračuje z koncového stavu hrany β a opakuje hledání α zkrácené o již nalezené β. Pokud cyklus skončí ve stavu a hledané α je prázdné slovo λ, pak je tento stav *předchůdcem* stavu q. Pokud se algoritmus dostane do stavu r, a β je delší než aktuálně hledané α, mohou nastat dvě situace.

Je-li hrana β *solid*, pak je nutné vytvořit nový stav s a hranu β jím rozdělit. Stav s se stane *předchůdcem* stavu q a algoritmus pokračuje hledáním *předchůdce* stavu s. Není-li hrana β solid, stačí hranu β zkrátit na délku α a přesměrovat do q. Algoritmus poté dále pokračuje v hledání *předchůdce* stavu q. Při dalším hledání začíná *FastFind* z *předchůdce* stavu r s hranou β.

Provádí-li *FastFind* přesměrování, nebo vytváří-li nový stav, vždy dochází k vkládání několika dalších přípon. Jejich počet je nutné určit již při volání funkce *FastFind*. *FastFind* je vždy volán z *předchůdce* nějakého stavu p a počet jím vložených přípon je dán rozdílem $length(p) - length(předchůdce(p))$.

V průběhu studia Crochemorova algoritmu jsem narazil na několik problémů, které nejsou v [2] dostatečně dobře vysvětleny. Příkladem je předcházející informace o určení počtu vložených přípon v průběhu funkce *FastFind*. Z tohoto důvodu jsem se rozhodl vypracovat podrobnější rozbor Crochemorova algoritmu v podobě pseudokódu, který je uveden v příloze B.

4.2 Implementace

Hrany datové struktury CDAWG jsou, na rozdíl od datové struktury DAWG, indexovány řetězci, ne pouze jediným znakem, a proto již není vhodné uchovávat popis

hran přímo v datových strukturách reprezentujících hranu. Výrazně lepším řešením je uložit celý vstupní řetězec v paměti a v hranách ukládat pouze index do tohoto řetězce na místa, ve kterých hrana začíná a končí.

Vzhledem k tomu, že všechny hrany vedoucí do jednoho stavu mají shodný index svého konce (končí na stejném místě ve vstupním řetězci), není nutné tento index ukládat jako součást reprezentace hrany, ale je možné jej ukládat jako součást reprezentace stavu. Protože datová struktura CDAWG má vždy více hran než stavů, bude tato informace uložena na méně místech. Jako součást reprezentace hrany je nutné ukládat pouze její délku. Začátek hrany je dán odečtením délky od konce hrany, který je uložen v jejím koncovém stavu. Index koncového stavu je také součástí reprezentace hrany.

4.3 Implementace v jazyce C

Implementaci v jazyce C jsem stejně jako při implementaci Blumerova algoritmu optimalizoval na rychlost pro malé abecedy. Z důvodů stejných s uvedenými v předchozí kapitole jsem se rozhodl pro použití pole tvořeného čísly typu *int*.

Stav p jsem tedy reprezentoval polem o $3 + |\Sigma| * 2$ prvcích. První prvek pole udává index do vstupního řetězce, na němž končí hrany vedoucí do p. Druhý prvek pole udává *předchůdce(p)* a třetí prvek pole udává hodnotu *length(p)*. Následující dvojice prvků vždy reprezentují jednu hranu vycházející z p. Pořadí dvojice je dáno abecedním pořadím prvního znaku hrany, čímž je možné naleznout hranu začínající písmenem α v konstantním čase. Z dvojice čísel reprezentujících hranu vždy první udává její délku a druhé index cílového stavu, do něhož hrana směřuje. Pokud ze stavu neexistuje hrana začínající daným písmenem, je její délka nastavena na 0.

Tato reprezentace umožňuje velmi rychlý běh funkce *FastFind*, protože přístup k hraně podle jejího prvního písmene znamená pouze jeden posun ukazatele. Na druhou stranu je tato reprezentace značně redundantní. Ne však tolik jako u Blumerova algoritmu, protože zde neexistují stavy, které by měly pouze jednu odchozí hranu, i přesto je průměrný počet hran vedoucích z jednotlivých stavů velmi nízký (viz. kapitola 6).

4.4 Implementace v jazyce C++

Implementací v jazyce C++ se snažím, stejně jako při implementaci Blumerova algoritmu, snížit požadavky na paměťovou náročnost reprezentace hran. V této situaci již není vhodné použít kontejner *map*. První znak hrany by bylo nutné explicitně ukládat jako klíč v tomto kontejneru, což by byla, při implementaci s indexy do vstupního řetězce, redundantní informace. Pro tuto implementaci přicházejí z nabídky kontejnerů STL v úvahu dva kontejnery: *list* a *vector*.

List je implementovaný jako obousměrný spojový seznam, který dokáže v konstantním čase vložit či odebrat libovolný prvek. Hledání v *listu* však vyžaduje čas lineární, protože je nutné procházet celý spojový seznam.

Vector je implementovaný jako pole, které svou velikost automaticky přizpůsobuje počtu vložených prvků. Vkládání na konec tohoto kontejneru má amortizovanou složitost konstantní, ale modifikující operace insert a delete mohou mít v nejhorším případě až lineární složitost. Hledání prvků má v kontejneru *vector* lineární složitost.

Vector má mírně lepší prostorovou složitost než *list*, protože nevyžaduje paměťové místo pro ukazatele sloužící k udržení spojového seznamu. Dle mých experimentálních výsledků, které v této práci nepublikuji, protože se vymykají tématu práce, je hledání v kontejneru *vector* rychlejší, než hledání v kontejneru *list*. Crochemorův algoritmus nikdy neodebírá hrany a jejich přidávání je možné provádět vždy na konec, proto jsem se rozhodl pro použití kontejneru *vector*.

```
// třída reprezentující hranu automatu
  class THrana{
  public:
    // inicializační konstruktor
      THrana(size_t , TStav *);
    // délka hrany odpovídající počtu znaků ve vstupním textu
      size_t Delka;
    // cílový stav hrany
      TStav * S;
  };
// množina hran
  typedef std::vector<THrana> THrany;

// třída reprezentující stav automatu
  class TStav{
  public:
    // inicializační konstuktor
      TStav();
    // seznam odchozích hran
      THrany Hrany;
    // index do vstupního textu udávající pozici konce vstupních hran
      size_t i;
    // ukazatel na předchůdce
      TStav * Zpetny;
    // délka nejdelší cesty ze zdroje
      unsigned int length;
  };
```

Kapitola 5

Inenagův algoritmus

5.1 Popis algoritmu

Inenagův algoritmus prezentovaný v [3] je prvním známým on-line konstrukčním algoritmem na konstrukci CDAWGu pracující v lineárním čase vzhledem k velikosti vstupního řetězce. Tento algoritmus vychází z Ukkonenova algoritmu [6] na konstrukci sufixového stromu a svým principem se podobá Blumerovu algoritmu na konstrukci DAWGu [1]. Aby byla zachována on-line vlastnost konstrukce, algoritmus musí pracovat iterativně. V cyklu postupně vkládá do datové struktury jednotlivé znaky vstupního řetězce w zleva doprava. V i-té iteraci musí zajistit korektní CDAWG pro řetězec $w[1] \dots w[i]$.

Velmi užitečným prvkem toho konstrukčního algoritmu jsou *otevřené hrany*. Pokud není známé, kolik dalších znaků bude ve vstupním textu následovat, je délka hrany označena ∞. Toto označení znamená, že je hrana ukončena až s koncem vstupního řetězce. Při přidávání dalšího znaku na konec vstupního řetězce není nutné tuto hranu prodlužovat. Pokud nemusí být přesměrování ani rozdělena, algoritmus ji nemusí nijak modifikovat.

Změny, které je nutné provádět během přechodu od CDAWG($w[1] \dots w[i-1]$) k CDAWG($w[1] \dots w[i]$) jsou velmi podobné změnám, které musí řešit Blumerův algoritmus. Hlavní rozdíl mezi těmito algoritmy je dán tím, že ne všechny stavy CDAWGu jsou vyjádřeny explicitně, tedy nějakým stavem automatu, ale mohou být vyjádřeny implicitně (uvnitř hrany označené více znaky). Je-li nutné z těchto stavů vést další hranu, či do nich nějakou hranu směřovat, musí algoritmus vytvořit nový explicitní stav a rozdělit jím původní hranu na dvě kratší části.

Stejně jako Crochemorův algoritmus i Inenagův vyžaduje pro svou činnost informace o *předchůdci* stavu a *length* stavu. Jejich definice i význam se shodují s uvedenými v předchozí kapitole.

Na počátku konstrukce probíhá inicializace tří stavů: *zdroje*, *stoku* a $\perp$ stavu, který slouží jako pomocný stav algoritmu. $\perp$ stav je označen jako *předchůdce zdroje* a jsou z něho vedeny hrany do *zdroje* přes všechna písmena abecedy.

Při vzniku nového stavu je mu nutné nalézt *předchůdce*. Stejně jako v obou předchozích algoritmech probíhá postupný průchod automatem po *předchůdcích* z nově vytvořeného stavu směrem ke *zdroji*. Průchod končí nalezením explicitního *předchůdce*, nebo nalezením *zdroje*. Stav $\perp$ slouží jako zarážka, aby nebylo nutné kontrolovat nalezení *zdroje*. Protože z tohoto stavu vedou hrany přes všechny znaky abecedy, každý průchod se zde zastaví a jako *předchůdce* nového stavu je označen *zdroj*.

Po inicializaci je spustěn cyklus, který postupně přidává jednotlivé znaky vstupního řetězce. Přesný popis jednotlivých kroků algoritmu lze ve formě velmi dobře zpracovaného pseudokódu naleznout v [3], a proto v této práci pseudokód Inenagova algoritmu neuvádím.

5.2 Implementace

Implementac Inenagova algoritmu se velmi podobná implementaci Crochemorova algoritmu. Jediný rozdíl, kterým se tyto implementace liší, je reprezentace hran. Kvůli problémům s přepočítáváním délky otevřených hran je výhodnější ukládat informace o konci hran do reprezentace hran než do reprezentace stavů. Hrana je tudíž reprezentována dvěma indexy do původního řetezce a indexem cílového stavu. Tento rozdíl způsobuje, že je tato reprezentace náročnější na paměť. Naopak umožňuje rychlejší vyhledávání, protože pro nalezení začátku hrany není nutné přistupovat k jejímu koncovému stavu.

5.3 Implementace v jazyce C

Implementaci v jazyce C jsem stejně jako v obou předchozích implementacích v C snažím optimalizovat na rychlou konstrukci a rychlé hledání. I tato implementace je určena hlavně pro malé abecedy. Z důvodů stejných s uvedenými v kapitole 3 jsem se rozhodl pro použití pole tvořeného datovým typem *int*.

Jednotlivé stavy jsou reprezentovány polem, v němž každému stavu p náleží $(2 + |\Sigma| * 3)$ prvků. První prvek udává index *předchůdce*, druhé udává hodnotu *length(p)* a následující prvky představují seznam hran. Hrana je tvořena indexy začátku a konce hrany do vstupního řetězce a indexem cílového stavu hrany. Ve srovnání s implementací použitou na konstrukci Crochemorova algoritmu vyžaduje tato implementace o $|\Sigma| - 1$ více 32bitových čísel na každý stav, což se výrazně projeví pro větší abecedy. Na druhou stranu tato reprezentace přináší určité urychlení při vyhledávání (viz. kapitola 6).

5.4 Implementace v jazyce C++

Při implementaci v jazyce C++ jsou hrany, stejně jako při implementaci Crochemorova algoritmu, ukládány do kontejneru *vector*. Hrana je reprezentována strukturou obsahující dva indexy do původního řetězce a ukazatel na cílový stav. Indexy udávají počátek a konec reprezentované hrany. Každý stav obsahuje, kromě vectoru hran, hodnotu *length(p)* a ukazatel na *předchůdce(p)*.

```
// třída reprezentující hranu automatu
  class THrana{
  public:
    // inicializační konstruktor
      THrana(int, int, TStav *);
    // index do vstupního textu udávající počátek hrany
      int odkud;
    // index do vstupního textu udávající konec hran
      int kam;
    // cílový stav hrany
      TStav * stav;
  };

// množina hran
  typedef std::vector<THrana> THrany;

// třída reprezentující stav automatu
  class TStav{
  public:
    // inicializační konstuktor
      TStav();
    // seznam odchozích hran
      THrany Hrany;
    // délka nejdelší cesty ze zdroje
      unsigned int length;
    // ukazatel na předchůdce
      TStav * Zpetny;
  };
```

Kapitola 6

Experimentální výsledky

Při měření experimentálních výsledků jsem se snažil co nejvíce simulovat reálnou situaci, ve které by algoritmy mohly být použity. Při měření konstrukčních vlastností jsem měřil skutečnou velikost paměti spotřebovanou procesem konstruujícím danou datovou strukturu a celý čas běhu procesu. Při měření vyhledávacích vlastností jsem měřil čistý čas úspěšného vyhledání bez doby konstrukce. Měření jsem prováděl pro vzorky délek 100, 1000 a 10000 znaků. Na testování jsem použil data z korpusů [4] a náhodně generovaných vzorků.

Všechna uvedená měření jsem prováděl na osobním 32bitovém počítači s procesorem Intel(R) Pentium(R) 4 CPU 2.80 GHz s 1 GB operační paměti a operačním systémem Linux Fedora Core 5. Zdrojové kódy jsem překládal pomocí překladače gcc (g++ pro jazyk C++) verze 4.1.1 s optimalizací O3. Měření jsem prováděl za pomocí skriptů, které jsou v příloze na CD. Měření času, paměti i vyhledávání jsem prováděl jednotlivě, aby se vzájemně neovlivňovaly. Každé měření paměti jsem prováděl v deseti nezávislých pokusech a paměť jsem měřil vždy třikrát. Jako výsledek měření uvádím průměrnou hodnotu všech pokusů.

6.1 Vlastnosti vstupních dat

Testování algoritmů jsem prováděl na dvou skupinách dat. První skupinou jsou vstupní data používaná v praxi a druhou jsou náhodně či speciálním způsobem generovaná data.

Mezi reálná vstupní data jsem zařadil vzorky z korpusů [4] a přidal k nim data použitá pro tuto bakalářskou práci. Soubory *bakalarka.tar.gz*, *bakalarka.tar.bz2* a *bakalarka.zip* obsahují celý můj Subversion repository. Jeho obsah je v příloze na CD.

Mezi generovanými vstupními daty se nacházejí soubory s pseudonáhodně generovanými posloupnostmi znaků a několik souborů se speciálně strkturovanými po-

sloupnostmi.Všechny náhodně generované soubory jsou označeny prefixem *random*, který je následován číslem udávajícím velikost abecedy, nad níž jsou data generována. Generování jsem prováděl standardním pseudonáhodným generátorem jazyka C. Soubory *alphabet.txt* a *alphabet2.txt* jsou soubory tvořené stále se opakující posloupnosti 26 znaků abecedy. Soubor *a.txt* obsahuje jeden jediný znak. Soubor *aaa.txt* je tvořen posloupností jediného znaku. Soubor *aaaaaab.txt* obsahuje posloupnost jednoho stále se opakujícího znaku a až posledním znakem posloupnosti je znak odlišný. Pro tento soubor jsou datové struktury DAWG i CDAWG totožné. Soubor *abbbbb.txt* je tvořen posloupností shodných znaků, s výjimkou prvního znaku, který se liší od následující posloupnosti. Datová struktura CDAWG dokáže tuto posloupnost uložit do pouhých dvou stavů a dvou hran, ale struktura DAWG vyžaduje na každý znak dva stavy a dvě hrany.

Vlastnosti datových struktur DAWG a CDAWG jsou uvedeny v příloze C. Sloupce *stavy* a *hrany* udávají počet stavů a hran datových struktur DAWG a CDAWG zkonstruovaných pro jednotlivé vstupní soubory. Sloupec *průměr* udává průměrný počet hran vycházejících z jednoho stavu.

6.2 Konstrukční vlastnosti algoritmů

Při měření konstrukčního času algoritmů jsem měřil celkovou dobu běhu procesu vytvářejícího danou datovou strukturu, tedy včetně načtení vstupních dat, alokace paměti a uvolnění paměti. Vzhledem k tomu, že načítání dat z disků může způsobit výrazné časové výkyvy a zkreslit tak měření, načetl jsem nejprve testovaný soubor do paměti jiným procesem. Tento soubor zůstal v diskové cache a měření nebylo ovlivněno přesuny hlaviček disků. Poté jsem pomocí standardního unixového programu *time* měřil celkovou dobu běhu procesů provádějících jednotlivé konstrukční algoritmy. Program *time* vypisuje tři statistické údaje: reálnou dobu mezi spuštěním a ukončením měřeného procesu, dobu běhu procesu v uživatelském režimu a dobu běhu procesu v režimu jádra. Do výsledků měření jsem zahrnul pouze celkovou dobu běhu procesu, protože zbývající dva údaje nepovažuji za důležité z hlediska konstrukčních vlastností algoritmů.

Každé měření času jsem prováděl v deseti nezávislých pokusech a do výsledků uvádím jejich průměrnou hodnotu. Přesnost měření je dána programem *time*, který pracuje v milisekundách. Odchylka jednotlivých měření se pohybovala v okolí $\pm 5\%$ od průměrné hodnoty.

Při měření paměti jsem opět pracoval s vlastnostmi celého procesu provádějícího konstrukci. Konstruující proces jsem v okamžiku dokončení konstrukce přepnul do režimu spánku a pomocí příkazů *ps −o size* určil velikost jím spotřebovávané paměti. Parametr *size* standardního unixového příkazu *ps* vypisuje velikost spotřebované virtuální paměti procesu (součet velikosti běžícího kódu, velikosti alokovaných dat

na haldě a velikosti zásobníku) v kilobytech. Tato metoda jako výslednou hodnotu neurčí maximální paměť spotřebovávanou procesem v průběhu konstrukce, protože před přepnutí do režimu spánku provádím uvolnění pomocných proměnných, ale určí paměť ve které setrvává proces, který je schopný s vytvořenou strukturou dále pracovat, například v ní vyhledávat nebo ji ještě dále prodlužovat (*on-line* algoritmy).

Měření paměti jsem prováděl vždy ve třech nezávislých pokusech. Jako výslednou hodnotu opět uvádím průměr těchto tři pokusů. Ve většině případů se všechny tři naměřené hodnoty shodovaly, takže rozptyl měření je velmi malý. Opakování měření jsem prováděl spíše pro kontrolu, než pro nalezení střední hodnoty.

Výsledky provedených měření jsou uvedeny v příloze D. Všechny uvedené časy jsou udávány v milisekundách na 1MB vstupních dat a velikosti paměti jsou uváděné v bytech spotřebované paměti na 1 znak vstupu.

Při obvyklých přístupech měření velikosti paměti uváděných například v [7] není měřena velikost paměti včetně dat běžícího procesu, ale je měřena pouze velikost samotné datové struktury. Při těchto měřeních jsou navíc odstraňovány pomocné informace konstrukčních algoritmů, což znemožňuje využít *on-line* vlastnosti algoritmů a dále prodlužovat vstupní data. I přesto se však ukazuje, že implementace v jazyce C je z hlediska paměťové složitosti vhodná pro abecedy nejvýše 4, protože mezi reálnými daty je *E.coli* jediným souborem, jehož výsledky jsou srovnatelné s výsledky uváděnými v [7]. Paměťové nároky implementace v jazyce C++ jsou sice menší než nároky implementace v jazyce C, ale i přesto je tato implementace paměťově náročnější, než implementace prezentované v [7].

Výhodou v této práci uvedených implementací je vysoká rychlost konstrukce a vyhledávání vzorků, která je lepší než výsledky uvedené v [7]. Na rychlost vyhledávání má však velký vliv použitý hardware, takže lze naměřené hodnoty jen velmi těžko porovnávat.

6.3 Zhodnocení výsledků konstrukčních vlastností

Při srovnání implementace **Blumerova** algoritmu v jazyce C s implementací v jazyce C++ z hlediska **paměťové** složitosti docházíme k očekávatelnému závěru, že implementace v jazyce C je výrazně úspornější pro menší velikost abecedy. Implementace v C++ je naopak vhodnější pro situace s větší abecedou. Velikost abecedy, pro níž jsou algoritmy vyrovnané, se nachází v okolí šestnácti. Zajímavé je zjištění, že i délka vstupního řetězce má výrazný vliv na vzájemný vztah obou implementací. Například srovnáním výsledků měření pro soubory *random16* a *random16velky* zjistíme, že při přibližně pětinásobném prodloužení délky vstupní posloupnosti zůstane paměťová náročnost (vzhledem k 1B vstupu) implementace v jazyce C přibližně stejná, ale náročnost implementace v C++ klesne téměř na polovinu.

Budeme-li srovnávat zmíněné implementace **Blumerova** algoritmu z hlediska **časové** složitosti zjistíme, že pro většinu použitých vzorků vychází výhodněji implementace v jazyce C. Mezi výjimky patří například soubor *obj2* nebo *kennedy.xls*. Společnou vlastností těchto souborů je velikost abecedy pohybující se v okolí sta a více. Číslo sto by tedy mohlo udávat přibližnou hranici, od které je z hlediska časové složitosti lepší implementace v C++. Proti této úvaze však stojí soubor *random128*, jehož abeceda je větší než sto a přesto je časová složitost implementace v C téměř třikrát nižší.

Při srovnání implementace **Crochemorova** algoritmu v jazyce C s implementací v jazyce C++ z hlediska **paměťové** složitosti se stejně jako u Blumerova algoritmu projeví skutečnost, že implementace v jazyce C je vhodná pouze pro malé abecedy. Hranice šířky abecedy, na níž jsou obě implementace vyrovnané, je opět přibližně šestnáct. Tentokrát však pro oba soubory *random16* i *random16velky* vychází výhodnější implementace v jazyce C.

Z hlediska **časové** složitosti implementací **Crochemorova** algoritmu vychází jednoznačně výhodnější implementace v jazyce C. Tato implementace je dle naměřených výsledků rychlejší pro většinu sledovaných vzorků bez ohledu na velikost abecedy či délku použitého souboru dat. Vyjímku tvoří sobory *ppt5*, *obj1* a *sum*. U souborů *sum* a *obj1* může být tato anomálie způsobena kombinací široké abecedy a krátkého vzorku. Široká abeceda způsobuje, že má implementace v jazyce C velkou spotřebu paměti a tím i velký čas vynaložený na její alokaci, ale pro krátkou vstupní posloupnost se tato investice nevyplatí. Tuto úvahu potvrzuje soubor *obj2*, který je svým charakterem souboru *obj1* velmi podobný, jen je výrazně delší a konstrukce CDAWGu při implementaci v jazyce C je pro něho rychlejší než implementace v jazyce C++.

Ze vzájemného porovnání obou implementací **Inenagova** algoritmu z **paměťového** hlediska vyplývají výsledky velmi podobné výsledkům získaným při porovnávání implementací obou předchozích algoritmů. Implementace jsou si přibližně rovnocenné pro abecedu velikosti šestnáct, pro užší abecedy je implementace v jazyce C úspornější a u větších abeced je tomu naopak.

Zajímavé je srovnání implementací **Inenagova** algoritmu z hlediska **časové** složitosti. Zatímco u Crochemorova algoritmu byla pro většinu vzorků výhodnější implementace v jazyce C, u Inenagova algoritmu jsou výsledky značně různorodější. Měření na reálných datech ukazují, že jsou obě implementace poměrně vyrovnané a hodně záleží na druhu vstupních dat, která jsou jim předkládána. Například pro soubory *E.coli* nebo *pi.txt* je značně rychlejší implementace v jazyce C, ale například pro soubory *ptt5*, *obj2* nebo *world192.txt* se ukazuje být rychlejší implementace v jazyce C++. Celkově vyznívají výsledky měření na reálných datech lépe pro implementaci v jazyce C++, ale naopak je implementace v C rychlejší téměř pro všechny generované vzorky.

Zajímavé je také pozorování, že implementace v jazyce C je rychlejší téměř pro všechny generované vzorky, na reálných datech však výsledky vyznívají lépe pro implementaci v jazyce C++.

Při vzájemném porovnání Blumerova algoritmu na konstrukci DAWGu a algoritmů na konstrukci CDAWGu se ukazuje, že Blumerův algoritmus je vhodný pro soubory s velmi malou abecedou. Jedná se například o soubory *E.coli*, *aaaaaab.txt* a *random2*. Tato skutečnost však platí pouze u implementace v jazyce C a rozdíl není příliš výrazný.

Při srovnání **Crochemorova** a **Inenagova** algoritmu se Crochemorův algoritmus zdá být **paměťově** úspornějším. Ukazuje se tak, že možnost ukládat informaci o koncích hran do stavů uspoří část paměti. Nejvýznamnější rozdíl je pozorovatelný na souboru *aaaaaab.txt*.

Z hlediska **časové** složitosti je při implementaci v jazyce C výrazně lepší **Crochemorův** algoritmus než algoritmus **Inenagův**. Při implementaci v jazyce C++ vycházejí výsledky měření výhodněji pro Inenagův algorimus, což může být mírně ovlivněno použitou úspornější implementací hran. Zajímavý je výsledek měření pro soubory *abbbbb.txt* a *alphabet2.txt*. Crochemorovu algoritmu se v obou implementacích podařilo postavit CDAWG pro tyto vstupy více než sedmkrát rychleji než Inenagovu algoritmu. Společnou vlastností obou uvedených souborů je, že navzdory jejich velké délce obsahuje nad nimi zkonstruovaný CDAWG pouze dva stavy a několik málo hran. Crochemorův algoritmus je schopen využít této skutečnosti výrazně lépe než algoritmus Inenagův.

6.4 Vyhledávací vlastnosti implementací

Měření vyhledávacích vlastností jednotlivých implementací jsem prováděl pro tři různé délky vzorků. Vzorky délky 100 jsem vyhledával 100000krát, vzorky délky 1000 jsem vyhledával 10000krát a vzorky délky 10000 jsem vyhledával 1000krát. Hledané vzorky jsem náhodně volil jako podřetězce vstupního řetězce. Celková délka hledaných dat se při všech měřeních shoduje.

Před začátkem měření jsem si vždy určil čas pomocí standardní unixové funkce $ftime$, která vrací aktuální čas s přesností na milisekundy. Poté jsem spustil algoritmus vyhledávání a po jeho skončení jsem opět, pomocí funkce $ftime$, určil aktuální čas. Rozdíl těchto časů uvádím jako výsledný čas hledání.

Čas neúspěšného vyhledání je dán časem úspěšného nalezení nejdelšího prefixu hledného řetězce ve vstupním textu. Tento čas tedy velmi záleží na charakteru hledaných vzorků a shora je odhadnutelný úspěšným hledáním celého vzorku. V této práci žádné výsledky měření neúspěšného hledání neuvádím.

Výsledky provedených měření jsou uvedeny v příloze E. Všechny časy jsou uváděny v milisekundách.

6.5 Zhodnocení výsledků vyhledávacích vlastností implementací

Základní poznatek, který z experimentálních výsledků vyhledávání jasně vyplývá, je výrazný rozdíl mezi časem hledání v automatu DAWG a CDAWG. Čas hledání v datové struktuře DAWG je mnohokrát horší než čas hledání v datové struktuře CDAWG. Jedinou výjimkou je soubor *aaaaaab.txt* v němž je, pro implementaci v jazyce C, hledání v datové struktuře DAWG rychlejší než hledání v datové struktuře CDAWG. Důvodem je pravděpodobně skutečnost, že DAWG i CDAWG mají nad uvedeným souborem stejný počet stavů i hran a všechny hrany CDAWG mají délku jedna. V takovéto situaci jsou nároky spojené s indexací hran pomocí řetězců značnou nevýhodou.

Vyhledávání v implementaci použité pro Inenagův algoritmus je rychlejší než vyhledávání v implementaci použité pro Crochemorův algoritmus. Ukazuje se tedy, že ukládání informace o konci hran do stavů přináší znatelné zvýšení času hledání.

Při pozorování vlivu délky hledaného řetězce na celkový čas hledání se ukazuje, že čím kratší vzorky jsou hledány, tím menší je rozdíl mezi hledáním v DAWG a hledáním v CDAWG. Naopak rozdíl mezi hledáním v reprezentaci použité při implemenaci Inenagova algoritmu a hledáním v reprezentaci použité při implementaci Crochemorova algoritmu je výraznější při hledání kratších vzorků. Dále je znatelné, že čím kratší vzorky jsou hledány, tím větší čas vyžaduje hledání v datové struktuře CDAWG. Pro datovou strukturu DAWG tato závislost není tak výrazná a pro některé soubory (například *alphabet.txt*, *alphabet2.txt*) je tomu dokonce naopak.

Soubory *random4maly* a *random4* přinášejí zajímavý poznatek vlivu délky vstupního řetězce na čas hledání vzorků různých délek. Zatímco hledání dlouhých vzorků trvá přibližně stejně dlouho v obou zmíněných souborech, hledání krátkých vzorků je výrazně rychlejší pro krátký vstupní text (*random4maly*) než pro delší vstupní text (*random4*).

Kapitola 7

Závěr

V této práci jsem popsal tři základní konstrukční algoritmy pro datové struktury DAWG a CDAWG. Implementoval jsem je ve dvou programovacích jazycích s použitím dvou různých reprezentací. Testoval jsem je na různých datech běžně používaných v praxi a na několika náhodně generovaných posloupnostech.

Ukázalo se, že Blumerův algoritmus dává nejlepší konstrukční výsledky pro soubory, jejichž velikost abecedy je nejvýše 4 a to při své implementaci v C. Čas jeho konstrukce je pro tyto soubory nejlepší ze všech zkoumaných implementací a paměťová nároky jsou jen mírně horší, než nároky implementace Crochemorova algoritmu.

Při srovnání Crochemorova a Inenagova algoritmu se zdá být paměťově úspornějším algoritmus Crochemorův, který umožňuje ukládat informaci o koncích hran do stavů. Největší rozdíl je pozorovatelný na datech, pro něž je počet hran datové struktury CDAWG výrazně vyšší než počet stavů. Z hlediska časové složitosti je při implementaci v jazyce C výrazně lepší Crochemorův algoritmus než algoritmus Inenagův. Při implementaci v jazyce C++ vycházejí výsledky měření výhodněji pro Inenagův algoritmus, což může být mírně ovlivněno použitou úspornější implementací hran, protože dle výsledků hledání je ukládání informací o hranách do stavů nevýhodou při vyhledávání.

V budoucnu by bylo zajímavé pokusit se o implementaci, která by zajistila přístup k hranám v konstantním čase, ale nebyla tak redundantní jako zde uvedená implementace v jazyce C. Dalo by se využít například hašování nebo podobných technik. Dále by bylo zajímavé zaměřit se na implementaci DAWGu a CDAWGu určenou pro větší abecedy. Speciální pozornost by si také jistě zasloužilo užití persistentních pamětí, což by mohlo být užitečné pro ukládání větších datových struktur.

Literatura

[1] A.Blumer, J.Blumer, D.Haussler, A.Ehrenfeucht, M.T.Chen, J.Seiferas. The smallest automaton recognizing the subwords of a text, Theoretical Computer Science 40 (1985) 31-55.

[2] M. Crochemore, R. Vérin, On compact directed acyclic word graphs, Structures in Logic and Computer Science, Lecture Notes in Computer Science, vol. 1261, Springer, Berlin, 1997, 192-211.

[3] S. Inenaga, H. Hoshino, A. Shinohara, M. Takeda, S. Arikawa, G. Mauri, G. Pavesi: On-Line Construction of Compact Directed Acyclic Word Graphs, Discrete Applied Mathematics 146(2005), 156-179.

[4] The Canterbury Corpus, http://corpus.canterbury.ac.nz

[5] E.McCreight. A space-economical suffix tree construction algorithm. Jurnal of the ACM, 23(2):262-272, Apr. 1976.

[6] E. Ukkonen. On-line construction of suffix trees. Algorithmica 14(3):249-260, 1995

[7] M.Balík. DAWG versus Suffix Array. Department of Computer Science and Engineering, Czech Technical University. Springer-Verlag Berlin Heidelberg, 2003, 289-294.

Příloha A

Obsah přiloženého CD

- Zdrojové kódy implementace všech algoritmů v jazyce C včetně Makefile.
- Dokumentace ke zdrojovým kódům implementace v jazyce C.
- Zdrojové kódy implementace všech algoritmů v jazyce C++ včetně Makefile.
- Dokumentace ke zdrojovým kódům implementace v jazyce C++.
- Zdrojové kódy tohoto textu pro sázecí systém LaTeX.
- Pracovní adresář obsahující pomocné skripty pro příkazový interpret *shell* používané při měření vlastností obou datových struktur a všech tří algoritmů.
- Použitá testovací data.
- Výsledky provedených měření v podobě označkovaných textových souborů, které jsou použity programem LaTeXpři vytváření tabulek.
- Soubor README obsahující podrobnější instrukce, jak zkompilovat a spustit dodané zdrojové kódy.

Příloha B

Crochemorův algoritmus

Procedura *postavCDAWG* má jediný argument. Je jím řetězec, pro který má být vytvořena datová strukturu CDAWG. Globální proměnná *last* udává poslední vytvořený stav, který ještě nemá známého předchůdce. p je aktivní stav, ve kterém aktuálně probíhá činnost algoritmu.

Funkce *slowFind* provádí hledání řetězce v doposud sestaveném automatu. Začíná ze stavu, který přijímá jako první parametr, prochází znak po znaku po hranách a kontroluje jejich shodu s aktuálně vkládanou příponou. Návratovou hodnotou je poslední navštívený stav a z něho vedoucí hrana, existuje-li hrana, po níž se dá pokračovat.

Funkce $fastFind$ je rychlejší variantou *slowFind*. $FastFind$ je spouštěn v situaci, kdy je jisté, že se jím hledaný řetězec v aktuálně sestaveném automatu nachází a jeho úkolem je pouze najít místo, kde hledaný řetězec končí. Funkce vždy pouze nalezne vhodnou hranu podle jejího prvního písmene a poté pokračuje ve stavu, do něhož nalezená hrana směřuje. Toto pokračuje tak dlouho, dokud není hledaný řetězec kratší, než nalezená hrana. Návratovou hodnotou je poslední navštívený stav.

```
    postavCDAWG(string retezec):
vytvoř a inicializuj zdroj
vytvoř a inicializuj stok
i = 0
p = zdroj
while (i + length(p) < délka(retezec))) do
  (stav, hrana) = slowFind(p, i+length(p))
  if (h == null) then
      vytvoř novou hranu ze stav do stok
      if (stav == zdroj) then
          p = zdroj
      else
```

```
            p = predchudce(p)
        endif
    else
        vytvoř nový stav novy
        rozdel hrana a vlož novy mezi stav a koncový stav hrany hrana
        vytvoř novou hranu z novy do stok
        last = novy
        stav r = fastFind(stav, zbytek hrany h vedoucí ze stavu novy)
        predchudce(novy) = r
        p = predchudce(last)
    endif
    zvys i o počet přípon přidaných ve fastFind
    i = i + 1
enddo
```

```
    (Stav, Hrana) slowFind(Stav stav, int index):
{ hledá nejdelší cestu ze stavu s, jejíž hrany tvoří předponu přípony retezec začínající
na pozici index }
if neexistuje hrana ze stav počínající znakem retezec[index] then
    return (stav, null)
else
    h = hrana ze stav počínající znakem retezec[index]
    if (not isSolid(h)) then { make h solid }
        vytvoř duplikát dup cílového stavu hrany h
        nastav dup jako cílový stav hrany h
        přesměruj hrany vedoucí do původního cílového stavu hrany h do
            nově vytvořeného duplikátu dup pomocí fastFind(stav, h)
    endif
    nalezni j pocet znaků, které jsou shodné od začátku h a od retezec[index]
    if (j == délka hrany h) then
        { rekurzivní volání }
        return slowFind(koncový stav hrany h, index + délka hrany h)
    else
        return (stav, h)
    endif
endif
```

Stav fastFind(Stav `stav`, string `hledane`):
`h` = hrana ze `stav` počínající znakem `retezec[index]` { *musí vždy existovat* }
if délka `h` == délka `hledane` **then**
  return cílový stav `h`
else if (délka `h` je menší než délka `hledane`) **then**
  zkrať `hledane` o delku `h`
  return fastFind(koncovy stav hrany `h`, `hledane`); { *rekurzivní volání* }
else if (`h` je solid)) **then**
  vytvoř nový stav `novy`
  rozdel `h` a vlož `novy` mezi `stav` a koncový stav hrany `h`
  vytvoř hranu z `novy` do `stok`
  nalezni `predchudce(novy)` pomocí FastFind(`predchudce(stav)`, `hledane`)
  `last` = `novy`
  return `novy`
else
  přesměruj `h` do `last`
  return fastFind(`predchudce(stav)`, `hledane`); { *posun na předchůdce* }
endif

Příloha C

Vlastnosti vstupních dat

Následující tabulky udávají vlastnosti vstupních dat.

- Sloupec *stavy* udává počet stavů automatu DAWG (CDAWG) na 1KB dat vstupního souboru.
- Sloupec *hrany* udává počet hran automatu DAWG (CDAWG) na 1KB dat vstupního souboru.
- Sloupec *průměr* udává průměrný počet hran vycházejících z jednoho stavu.

Zdroj	$\|\Sigma\|$	$\|w\|$ (B)	počet stavů (1KB vstupu)	počet hran (1KB vstupu)	průměr
alice29.txt	74	152089	1577	2227	1.41
asyoulik.txt	68	125179	1537	2234	1.45
bakalarka.tar.bz2	256	68420	1264	2283	1.80
bakalarka.tar.gz	256	201109	1334	2319	1.74
bakalarka.zip	256	371632	1777	2007	1.13
bib	81	111261	1557	2027	1.30
bible.txt	63	2097152	1602	2074	1.29
book1	82	768771	1546	2273	1.47
book2	96	610856	1578	2164	1.37
cp.html	86	24603	1569	2079	1.32
E.coli	4	4638690	1680	2590	1.54
fields.c	90	11150	1650	2092	1.27
geo	256	102400	1328	2085	1.57
grammar.lsp	76	3721	1633	2124	1.30
lcet10.txt	84	426754	1575	2162	1.37
news	98	377109	1560	2146	1.38
obj1	256	21504	1382	2094	1.51
obj2	256	246814	1494	1930	1.29
paper1	95	53161	1589	2183	1.37
paper2	91	82199	1560	2194	1.41
paper3	84	46526	1551	2223	1.43
paper4	80	13286	1561	2254	1.44
paper5	91	11954	1551	2221	1.43
paper6	93	38105	1602	2201	1.37
pi.txt	10	1000000	1437	2438	1.70
plrabn12.txt	81	481861	1535	2237	1.46
progc	92	39611	1584	2150	1.36
progl	87	71646	1679	2072	1.23
progp	89	49379	1699	2083	1.23
ptt5	159	513216	1567	1931	1.23
sum	255	38240	1477	2022	1.37
trans	99	93695	1751	2057	1.18
world192.txt	90	1572864	1562	1935	1.24
xargs.1	74	4227	1571	2201	1.40

Tabulka C.1: Vlastnosti datové struktury DAWG na reálných datech

Zdroj	$\|\Sigma\|$	$\|w\|$ (B)	počet stavů (1KB vstupu)	počet hran (1KB vstupu)	průměr
alice29.txt	74	152089	278	928	3.34
asyoulik.txt	68	125179	288	985	3.41
bakalarka.tar.bz2	256	68420	233	1253	5.36
bakalarka.tar.gz	256	201109	272	1258	4.62
bakalarka.zip	256	371632	70	300	4.25
bib	81	111261	189	659	3.48
bible.txt	63	2097152	209	681	3.25
book1	82	768771	308	1036	3.36
book2	96	610856	249	835	3.35
cp.html	86	24603	190	700	3.67
E.coli	4	4638690	549	1459	2.65
fields.c	90	11150	179	621	3.46
geo	256	102400	180	937	5.20
grammar.lsp	76	3721	206	697	3.37
lcet10.txt	84	426754	248	835	3.37
news	98	377109	223	809	3.63
obj1	256	21504	207	918	4.44
obj2	256	246814	151	587	3.88
paper1	95	53161	245	839	3.42
paper2	91	82199	266	900	3.38
paper3	84	46526	278	950	3.42
paper4	80	13286	283	976	3.45
paper5	91	11954	264	934	3.53
paper6	93	38105	246	845	3.43
pi.txt	10	1000000	391	1392	3.56
plrabn12.txt	81	481861	294	996	3.38
progc	92	39611	223	789	3.53
progl	87	71646	174	568	3.25
progp	89	49379	161	545	3.39
ptt5	159	513216	139	502	3.61
sum	255	38240	165	709	4.29
trans	99	93695	132	438	3.32
world192.txt	90	1572864	152	526	3.45
xargs.1	74	4227	243	873	3.58

Tabulka C.2: Vlastnosti datové strukury CDAWG na reálných datech

Zdroj	$\|\Sigma\|$	$\|w\|$ (B)	počet stavů (1KB vstupu)	počet hran (1KB vstupu)	průměr
aaa.txt	1	100000	1024	1024	1.00
aaaaaab.txt	2	1000000	1024	2047	1.99
abbbbb.txt	2	1000000	2047	2047	1.00
alphabet2.txt	26	1000000	1024	1024	1.00
alphabet.txt	26	100000	1024	1024	1.00
a.txt	1	1	1	1	1.00
random128	128	1048576	1225	2248	1.83
random16	16	1048576	1381	2392	1.73
random16velky	16	5000000	1368	2383	1.74
random2	2	1048576	2047	2821	1.38
random32	32	1048576	1337	2355	1.76
random4	4	1048576	1661	2604	1.57
random4maly	4	100000	1659	2602	1.57
random4velky	4	5000000	1660	2604	1.57
random64	64	1048576	1290	2313	1.79
random8	8	1048576	1483	2480	1.67

Tabulka C.3: Vlastnosti datové struktury DAWG na generovaných datech

Zdroj	$\|\Sigma\|$	$\|w\|$ (B)	počet stavů (1KB vstupu)	počet hran (1KB vstupu)	průměr
aaa.txt	1	100000	2	1	0.50
aaaaaab.txt	2	1000000	1024	2047	1.99
abbbbb.txt	2	1000000	2	2	1.00
alphabet2.txt	26	1000000	2	26	13.00
alphabet.txt	26	100000	2	26	13.00
a.txt	1	1	2	1	0.50
random128	128	1048576	200	1223	6.11
random16	16	1048576	344	1355	3.93
random16velky	16	5000000	336	1351	4.02
random2	2	1048576	773	1546	1.9
random32	32	1048576	307	1324	4.31
random4	4	1048576	560	1503	2.68
random4maly	4	100000	558	1501	2.69
random4velky	4	5000000	560	1503	2.68
random64	64	1048576	265	1288	4.85
random8	8	1048576	433	1430	3.30

Tabulka C.4: Vlastnosti datové struktury CDAWG na generovaných datech

Příloha D

Konstrukční vlastnosti algoritmů

V následujících tabulkách jsou uvedeny výsledky provedených měření konstrukčních vlastností algoritmů. Všechny uvedené časy jsou udávány v milisekundách na 1MB vstupních dat a velikosti paměti jsou uváděné v bytech spotřebované paměti na 1 znak vstupu.

Zdroj	$\|\Sigma\|$	$\|w\|$ (B)	čas C (ms/MB)	čas C++ (ms/MB)	paměť C (B/znak)	paměť C++ (B/znak)
alice29.txt	74	152089	1344,4	1737,4	451,9	154,6
asyoulik.txt	68	125179	1222,9	1750,7	406,7	152,9
bakalarka.tar.bz2	256	68420	2620,6	2865,8	1242,8	140,9
bakalarka.tar.gz	256	201109	2659,1	5730,1	1308,5	144,1
bakalarka.zip	256	371632	4116,6	6435,9	1743,0	157,6
bib	81	111261	1451,3	1517,3	489,4	148,3
bible.txt	63	2097152	1155,0	1639,0	391,2	149,9
book1	82	768771	1501,7	1987,2	489,6	153,2
book2	96	610856	1649,6	1750,8	584,1	151,6
cp.html	86	24603	1491,6	1491,6	530,8	155,7
E.coli	4	4638690	439,2	1997,3	32,0	108,8
fields.c	90	11150	1692,7	2257,0	594,8	166,1
geo	256	102400	2969,6	2979,8	1304,8	137,9
grammar.lsp	76	3721	1690,7	2254,3	545,2	213,9
kennedy.xls	250	524288	1710,0	970,0	1010,2	104,2
lcet10.txt	84	426754	1545,5	1774,0	511,1	151,6
news	98	377109	1685,0	1901,9	589,5	150,4
obj1	256	21504	3120,7	1609,1	1366,6	147,6
obj2	256	246814	3207,5	1720,6	1466,0	140,6
paper1	95	53161	1696,3	1617,4	586,5	156,5
paper2	91	82199	1658,3	1683,8	550,4	154,2
paper3	84	46526	1510,0	1690,3	508,1	156,2
paper4	80	13286	1420,6	1657,3	500,8	169,2
paper5	91	11954	1578,9	1666,6	564,9	165,9
paper6	93	38105	1678,6	1623,5	581,0	159,5
pi.txt	10	1000000	495,9	2188,3	60,4	152,3
plrabn12.txt	81	481861	1481,9	1901,9	480,6	151,6
progc	92	39611	1614,7	1588,3	568,4	156,7
progl	87	71646	1653,8	1419,6	567,2	156,7
progp	89	49379	1720,0	1443,9	588,2	160,4
ptt5	159	513216	2237,2	1227,9	957,0	143,9
sum	255	38240	3400,1	1672,6	1449,6	148,5
trans	99	93695	1980,8	1443,6	670,6	159,2
world192.txt	90	1572864	1508,6	1533,3	542,3	143,6
xargs.1	74	4227	1488,3	1984,5	507,6	188,7

Tabulka D.1: Konstrukční vlastnosti Blumerova algoritmu na vybraných reálných datech

Zdroj	$\|\Sigma\|$	$\|w\|$ (B)	čas C (ms/MB)	čas C++ (ms/MB)	paměť C (B/znak)	paměť C++ (B/znak)
aaa.txt	1	100000	52,4	461,3	10,2	88,4
aaaaaab.txt	2	1000000	62,9	567,2	11,8	130,9
abbbbb.txt	2	1000000	87,0	980,4	23,5	173,1
alphabet2.txt	26	1000000	179,3	399,5	105,5	87,1
alphabet.txt	26	100000	220,2	450,8	107,8	88,4
a.txt	1	1	1048576,0	3145728,0	242000,0	272000,0
random128	128	1048576	1397,0	3817,0	603,1	135,1
random16	16	1048576	577,0	2411,0	89,7	148,0
random16velky	16	5000000	763,1	2632,5	88,7	78,0
random2	2	1048576	275,0	1847,0	23,5	196,6
random32	32	1048576	744,0	2914,0	168,4	144,4
random4	4	1048576	383,0	1932,0	31,8	169,4
random4maly	4	100000	304,0	1761,6	34,1	171,6
random4velky	4	5000000	479,4	2035,7	31,7	91,2
random64	64	1048576	1006,0	3263,0	320,1	140,7
random8	8	1048576	466,0	2124,0	51,0	156,0

Tabulka D.2: Konstrukční vlastnosti Blumerova algoritmu na generovaných datech

Zdroj	$\|\Sigma\|$	$\|w\|$ (B)	čas C (ms/MB)	čas C++ (ms/MB)	paměť C (B/znak)	paměť C++ (B/znak)
aaa.txt	1	100000	83,8	41,9	2,4	2,6
aaaaaab.txt	2	1000000	214,9	869,2	60,9	182,6
abbbbb.txt	2	1000000	50,3	44,0	1,2	1,2
alphabet2.txt	26	1000000	48,2	83,8	1,2	1,2
alphabet.txt	26	100000	62,9	314,5	2,4	2,6
a.txt	1	1	1048576,0	3145728,0	242000,0	270000,0
random128	128	1048576	810,0	2223,0	296,0	43,0
random16	16	1048576	545,0	1441,0	66,9	51,5
random16velky	16	5000000	616,1	1868,9	65,1	62,3
random2	2	1048576	404,0	934,0	24,8	60,7
random32	32	1048576	605,0	1645,0	116,0	50,5
random4	4	1048576	483,0	1198,0	31,1	57,8
random4maly	4	100000	367,0	933,2	32,2	59,4
random4velky	4	5000000	549,6	1402,1	30,9	52,7
random64	64	1048576	736,0	1903,0	197,9	46,8
random8	8	1048576	513,0	1319,0	44,1	54,1

Tabulka D.3: Konstrukční vlastnosti Inenagova algoritmu na generovaných datech

Zdroj	$\|\Sigma\|$	$\|w\|$ (B)	čas C (ms/MB)	čas C++ (ms/MB)	paměť C (B/znak)	paměť C++ (B/znak)
alice29.txt	74	152089	765,2	654,9	240,1	35,7
asyoulik.txt	68	125179	762,2	678,5	228,9	39,0
bakalarka.tar.bz2	256	68420	1440,6	1072,7	689,9	52,0
bakalarka.tar.gz	256	201109	1543,3	1830,1	802,8	51,5
bakalarka.zip	256	371632	547,3	725,1	209,1	13,4
bib	81	111261	593,7	471,2	179,0	27,3
bible.txt	63	2097152	765,5	823,5	153,8	25,6
book1	82	768771	917,9	942,4	292,8	38,9
book2	96	610856	799,9	708,9	277,2	31,4
cp.html	86	24603	596,6	468,8	199,0	32,4
E.coli	4	4638690	539,1	1339,7	30,3	56,0
fields.c	90	11150	564,2	564,2	208,6	47,8
geo	256	102400	1136,6	778,2	532,2	36,1
grammar.lsp	76	3721	845,3	1127,1	246,4	73,0
kennedy.xls	250	524288	360,0	650,0	40,5	16,3
lcet10.txt	84	426754	746,9	687,9	241,9	31,6
news	98	377109	728,5	678,4	253,5	30,7
obj1	256	21504	1365,3	633,9	619,6	43,2
obj2	256	246814	926,1	471,5	446,9	22,9
paper1	95	53161	788,9	532,5	273,2	34,8
paper2	91	82199	816,4	599,5	282,5	37,0
paper3	84	46526	811,3	585,9	274,6	39,8
paper4	80	13286	789,2	631,3	279,6	50,0
paper5	91	11954	789,4	701,7	297,4	55,6
paper6	93	38105	770,5	550,3	270,7	38,2
pi.txt	10	1000000	488,6	1350,5	48,9	53,2
plrabn12.txt	81	481861	857,3	1027,1	276,6	37,9
progc	92	39611	688,2	502,9	243,2	33,5
progl	87	71646	541,5	380,5	178,8	24,0
progp	89	49379	509,6	360,9	170,1	24,1
ptt5	159	513216	570,0	345,2	256,3	18,9
sum	255	38240	1069,4	520,9	490,5	31,1
trans	99	93695	470,0	324,5	153,4	19,7
world192.txt	90	1572864	702,0	548,0	159,6	20,2
xargs.1	74	4227	744,1	992,2	297,3	95,1

Tabulka D.4: Konstrukční vlastnosti Inenagova algoritmu na vybraných reálných datech

Zdroj	$\|\Sigma\|$	$\|w\|$ (B)	čas C (ms/MB)	čas C++ (ms/MB)	paměť C (B/znak)	paměť C++ (B/znak)
alice29.txt	74	152089	606,7	792,8	161,8	34,0
asyoulik.txt	68	125179	594,7	820,9	156,0	35,8
bakalarka.tar.bz2	256	68420	1088,1	1793,0	463,5	44,4
bakalarka.tar.gz	256	201109	1105,3	5917,8	537,0	43,0
bakalarka.zip	256	371632	366,8	1458,7	140,0	11,6
bib	81	111261	433,5	537,1	122,2	24,9
bible.txt	63	2097152	450,0	826,0	104,1	24,1
book1	82	768771	728,3	1181,1	197,4	36,3
book2	96	610856	605,9	896,0	186,6	29,5
cp.html	86	24603	426,1	468,8	138,2	32,3
E.coli	4	4638690	473,1	1496,9	24,0	49,8
fields.c	90	11150	470,2	564,2	147,8	35,8
geo	256	102400	829,4	1259,5	357,7	33,5
grammar.lsp	76	3721	563,5	1127,1	187,0	72,5
kennedy.xls	250	524288	106,0	652,0	27,3	14,0
lcet10.txt	84	426754	567,5	864,8	163,0	29,7
news	98	377109	550,5	931,4	170,6	28,6
obj1	256	21504	926,4	682,6	419,1	43,2
obj2	256	246814	650,0	713,7	299,0	21,3
paper1	95	53161	552,2	572,0	186,1	32,3
paper2	91	82199	612,3	676,0	191,9	33,7
paper3	84	46526	585,9	653,5	187,5	36,9
paper4	80	13286	552,4	631,3	195,1	50,2
paper5	91	11954	614,0	614,0	207,5	55,7
paper6	93	38105	550,3	550,3	185,0	34,7
pi.txt	10	1000000	447,7	1734,3	35,4	49,8
plrabn12.txt	81	481861	668,0	1081,5	186,5	35,5
progc	92	39611	476,4	529,4	166,5	33,4
progl	87	71646	365,8	380,5	122,4	24,0
progp	89	49379	339,7	339,7	117,0	24,2
ptt5	159	513216	388,1	347,3	172,0	21,5
sum	255	38240	740,3	520,9	331,0	31,1
trans	99	93695	324,5	324,5	104,9	18,3
world192.txt	90	1572864	391,3	696,0	107,6	18,8
xargs.1	74	4227	744,1	992,2	199,1	94,8

Tabulka D.5: Konstrukční vlastnosti Crochemorova algoritmu na vybraných reálných datech

Zdroj	$\|\Sigma\|$	$\|w\|$ (B)	čas C (ms/MB)	čas C++ (ms/MB)	paměť C (B/znak)	paměť C++ (B/znak)
aaa.txt	1	100000	20,9	31,4	3,4	2,6
aaaaaab.txt	2	1000000	94,3	530,5	28,4	87,3
abbbbb.txt	2	1000000	7,3	8,3	1,0	1,2
alphabet2.txt	26	1000000	7,3	8,3	1,0	1,2
alphabet.txt	26	100000	20,9	31,4	3,4	2,6
a.txt	1	1	1048576,0	3145728,0	242000,0	268000,0
random128	128	1048576	618,0	6216,0	198,9	37,9
random16	16	1048576	474,0	2061,0	47,0	47,8
random16velky	16	5000000	553,2	2697,7	45,8	56,3
random2	2	1048576	351,0	926,0	21,7	60,7
random32	32	1048576	508,0	2839,0	79,5	46,4
random4	4	1048576	423,0	1323,0	24,6	55,8
random4maly	4	100000	293,6	1027,6	26,8	58,1
random4velky	4	5000000	487,3	1575,3	24,4	45,9
random64	64	1048576	611,0	4175,0	133,9	42,9
random8	8	1048576	450,0	1640,0	32,4	51,2

Tabulka D.6: Konstrukční vlastnosti Crochemorova algoritmu na generovaných datech

Příloha E

Vyhledávací vlastnosti algoritmů

Následující tabulky udávají výsledky provedených měření vyhledávacích vlastností algoritmů. Všechny časy odpovídají hledání 10 miliónů znaků a jsou uváděny v milisekundách.

Zdroj	$\|\Sigma\|$	$\|w\|$ (B)	blumer (ms)	crochemore (ms)	inenaga (ms)
aaaaaab.txt	2	1000000	85	163	184
aaa.txt	1	100000	88	22	21
abbbbb.txt	2	1000000	111	23	22
alice29.txt	74	152089	1808	24	23
alphabet.txt	26	100000	775	21	22
alphabet2.txt	26	1000000	773	23	22
asyoulik.txt	68	125179	1796	25	22
bakalarka.tar.bz2	256	68420	1824	22	22
bakalarka.tar.gz	256	201109	1843	22	22
bib	81	111261	1804	23	23
book1	82	768771	1818	25	24
book2	96	610856	1821	24	24
cp.html	86	24603	1760	22	22
E.coli	4	4638690	236	25	25
fields.c	90	11150	1813	22	22
geo	256	102400	1832	22	22
lcet10.txt	84	426754	1816	25	24
news	98	377109	1818	24	23
obj1	256	21504	1795	23	23
obj2	256	246814	1849	24	23
paper1	95	53161	1793	22	22
paper2	91	82199	1801	22	22
paper3	84	46526	1782	23	22
pi.txt	10	1000000	415	23	23
plrabn12.txt	81	481861	1815	24	24
progc	92	39611	1775	23	22
progl	87	71646	1796	23	23
progp	89	49379	1791	23	22
random16	16	1048576	648	24	24
random2	2	1048576	179	25	25
random32	32	1048576	1603	23	23
random4	4	1048576	233	24	24
random4maly	4	100000	221	23	22
random8	8	1048576	354	24	24
sum	255	38240	1786	22	22
trans	99	93695	1807	23	22

Tabulka E.1: Vyhledávací vlastnosti implementace v jazyce C 1000krát hledán vzorek délky 10000 znaků

Zdroj	$\|\Sigma\|$	$\|w\|$ (B)	blumer (ms)	crochemore (ms)	inenaga (ms)
aaaaaab.txt	2	1000000	87	107	153
aaa.txt	1	100000	76	29	22
abbbbb.txt	2	1000000	107	32	27
alice29.txt	74	152089	1784	42	32
alphabet.txt	26	100000	143	29	22
alphabet2.txt	26	1000000	147	37	26
asyoulik.txt	68	125179	1769	40	32
bakalarka.tar.bz2	256	68420	1765	34	26
bakalarka.tar.gz	256	201109	1824	35	28
bib	81	111261	1772	42	30
book1	82	768771	1842	51	42
book2	96	610856	1819	51	45
cp.html	86	24603	1592	34	26
E.coli	4	4638690	259	50	45
fields.c	90	11150	1290	29	24
geo	256	102400	1796	36	30
lcet10.txt	84	426754	1813	45	39
news	98	377109	1810	47	36
obj1	256	21504	1583	39	35
obj2	256	246814	1838	47	36
paper1	95	53161	1717	38	29
paper2	91	82199	1755	39	30
paper3	84	46526	1696	37	29
pi.txt	10	1000000	428	40	35
plrabn12.txt	81	481861	1811	48	41
progc	92	39611	1675	36	28
progl	87	71646	1747	39	30
progp	89	49379	1715	40	28
random16	16	1048576	661	40	34
random2	2	1048576	201	53	48
random32	32	1048576	1605	39	33
random4	4	1048576	250	44	38
random4maly	4	100000	229	36	29
random8	8	1048576	369	41	35
sum	255	38240	1736	36	27
trans	99	93695	1775	43	31

Tabulka E.2: Vyhledávací vlastnosti implementace v jazyce C 10000krát hledán vzorek délky 1000 znaků

Zdroj	$\|\Sigma\|$	$\|w\|$ (B)	blumer (ms)	crochemore (ms)	inenaga (ms)
aaaaaab.txt	2	1000000	92	105	165
aaa.txt	1	100000	73	33	24
abbbbb.txt	2	1000000	92	51	42
alice29.txt	74	152089	1849	163	130
alphabet.txt	26	100000	99	33	25
alphabet2.txt	26	1000000	122	52	42
asyoulik.txt	68	125179	1822	142	113
bakalarka.tar.bz2	256	68420	1825	77	65
bakalarka.tar.gz	256	201109	1886	97	83
bib	81	111261	1829	161	115
book1	82	768771	1896	219	196
book2	96	610856	1901	230	203
cp.html	86	24603	1620	86	67
E.coli	4	4638690	447	218	220
fields.c	90	11150	1362	59	52
geo	256	102400	1854	92	73
lcet10.txt	84	426754	1883	196	182
news	98	377109	1885	189	161
obj1	256	21504	1455	88	76
obj2	256	246814	1926	202	157
paper1	95	53161	1771	122	96
paper2	91	82199	1812	135	108
paper3	84	46526	1743	112	90
pi.txt	10	1000000	559	134	129
plrabn12.txt	81	481861	1879	196	171
progc	92	39611	1726	109	84
progl	87	71646	1778	143	110
progp	89	49379	1730	125	90
random16	16	1048576	791	126	121
random2	2	1048576	373	245	254
random32	32	1048576	1667	122	113
random4	4	1048576	395	168	167
random4maly	4	100000	325	99	98
random8	8	1048576	501	141	137
sum	255	38240	1716	96	73
trans	99	93695	1819	168	115

Tabulka E.3: Vyhledávací vlastnosti implementace v jazyce C 100000krát hledán vzorek délky 100 znaků

Zdroj	$\|\Sigma\|$	$\|w\|$ (B)	blumer (ms)	crochemore (ms)	inenaga (ms)
aaaaaab.txt	2	1000000	1530	1255	1388
aaa.txt	1	100000	1509	33	29
abbbbb.txt	2	1000000	2143	31	30
alice29.txt	74	152089	2052	34	32
alphabet.txt	26	100000	1485	32	29
alphabet2.txt	26	1000000	1509	34	30
asyoulik.txt	68	125179	2041	33	32
bakalarka.tar.bz2	256	68420	1956	35	38
bakalarka.tar.gz	256	201109	2011	41	35
bib	81	111261	1971	31	32
book1	82	768771	2072	39	34
book2	96	610856	2043	39	36
cp.html	86	24603	1976	32	30
fields.c	90	11150	2012	29	29
geo	256	102400	1911	33	32
lcet10.txt	84	426754	2050	36	36
news	98	377109	2042	36	33
obj1	256	21504	1904	36	31
obj2	256	246814	1927	37	37
paper1	95	53161	2013	32	31
paper2	91	82199	2036	38	31
paper3	84	46526	2033	31	31
pi.txt	10	1000000	2084	37	38
plrabn12.txt	81	481861	2047	38	33
progc	92	39611	2015	30	32
progl	87	71646	2043	32	31
progp	89	49379	2045	33	31
random16	16	1048576	2052	36	38
random2	2	1048576	2458	40	38
random32	32	1048576	2026	38	35
random4	4	1048576	2221	37	35
random4maly	4	100000	2186	32	33
random8	8	1048576	2098	36	34
sum	255	38240	1981	30	34
trans	99	93695	2057	32	33

Tabulka E.4: Vyhledávací vlastnosti implementace v jazyce C++ 1000krát hledán vzorek délky 10000 znaků

Zdroj	$\|\Sigma\|$	$\|w\|$ (B)	blumer (ms)	crochemore (ms)	inenaga (ms)
aaaaaab.txt	2	1000000	279	205	230
aaa.txt	1	100000	276	29	29
abbbbb.txt	2	1000000	307	33	33
alice29.txt	74	152089	1234	70	49
alphabet.txt	26	100000	269	30	29
alphabet2.txt	26	1000000	284	33	33
asyoulik.txt	68	125179	1194	57	46
bakalarka.tar.bz2	256	68420	1101	83	39
bakalarka.tar.gz	256	201109	1162	98	48
bib	81	111261	1124	59	46
book1	82	768771	1273	76	65
book2	96	610856	1246	76	64
cp.html	86	24603	1091	39	36
fields.c	90	11150	1024	32	33
geo	256	102400	1054	62	43
lcet10.txt	84	426754	1232	72	60
news	98	377109	1223	69	56
obj1	256	21504	1007	47	43
obj2	256	246814	1109	82	57
paper1	95	53161	1175	49	42
paper2	91	82199	1170	53	46
paper3	84	46526	1182	48	42
pi.txt	10	1000000	1270	72	58
plrabn12.txt	81	481861	1234	71	58
progc	92	39611	1180	43	39
progl	87	71646	1194	58	43
progp	89	49379	1200	52	43
random16	16	1048576	1237	73	56
random2	2	1048576	1682	98	86
random32	32	1048576	1210	86	58
random4	4	1048576	1409	82	67
random4maly	4	100000	1344	53	46
random8	8	1048576	1299	72	59
sum	255	38240	1103	46	37
trans	99	93695	1225	53	45

Tabulka E.5: Vyhledávací vlastnosti implementace v jazyce C++ 10000krát hledán vzorek délky 1000 znaků

Zdroj	$\|\Sigma\|$	$\|w\|$ (B)	blumer (ms)	crochemore (ms)	inenaga (ms)
aaaaaab.txt	2	1000000	275	208	232
aaa.txt	1	100000	264	32	32
abbbbb.txt	2	1000000	308	49	49
alice29.txt	74	152089	1637	308	226
alphabet.txt	26	100000	280	38	35
alphabet2.txt	26	1000000	288	56	52
asyoulik.txt	68	125179	1582	275	202
bakalarka.tar.bz2	256	68420	1369	306	121
bakalarka.tar.gz	256	201109	1501	973	214
bib	81	111261	1555	259	199
book1	82	768771	1816	455	348
book2	96	610856	1791	446	344
cp.html	86	24603	1351	104	91
fields.c	90	11150	1207	65	63
geo	256	102400	1392	320	147
lcet10.txt	84	426754	1748	414	315
news	98	377109	1745	396	279
obj1	256	21504	1120	127	100
obj2	256	246814	1633	503	273
paper1	95	53161	1494	199	153
paper2	91	82199	1535	249	185
paper3	84	46526	1453	185	146
pi.txt	10	1000000	1723	386	285
plrabn12.txt	81	481861	1726	400	298
progc	92	39611	1453	168	129
progl	87	71646	1590	217	172
progp	89	49379	1566	171	135
random16	16	1048576	1698	406	275
random2	2	1048576	2214	573	552
random32	32	1048576	1675	469	274
random4	4	1048576	1885	427	365
random4maly	4	100000	1653	249	202
random8	8	1048576	1761	390	295
sum	255	38240	1396	173	112
trans	99	93695	1659	244	181

Tabulka E.6: Vyhledávací vlastnosti implementace v jazyce C++ 100000krát hledán vzorek délky 100 znaků