FAKULTA
APLIKOVANÝCH VĚD
ZÁPADOČESKÉ
UNIVERZITY
V PLZNI

Západočeská univerzita

Semestrální práce - KIV/PC

Jednoduchý stemmer

Petr Kraus

15. ledna 2019

Obsah

1 Zadání

Naprogramujte v ANSI C přenositelnou konzolovou aplikaci, která bude pracovat jako tzv. stemmer. Stemmer je algoritmus, resp. program, který hledá kořeny slov. Stemmer pracuje ve dvou režimech: (i) v režimu učení, kdy je na vstupu velké množství textu (tzv. korpus) v jednom konkrétním etnickém jazyce (libovolném) a na výstupu pak slovník (seznam) kořenů slov; nebo (ii) v režimu zpracování slov, kdy je na vstupu slovo (nebo sekvence slov) a stemmer ke každému z nich určí jeho kořen.

Režim činnosti programu je dán předaným parametrem: Je-li parametrem jméno (a případně cesta k) souboru, pak bude stemmer pracovat v režimu učení, tedy tvorby databáze kořenů na základě analýzy dat z korpusu. Je-li parametrem slovo nebo sekvence slov (ta musí být uzavřena v uvozovkách), pak stemmer bude pracovat v režimu zpracování slov, tedy určování kořene každého slova ze sekvence.

Celé zadání viz `https://www.kiv.zcu.cz/studies/predmety/pc/doc/work/sw2018-03.pdf`

2 Analýza úlohy

Úloha se skládá ze dvou hlavních problémů: (i) vyhledávání kořenů v režimu učení; (ii) Porovnání slova (či sekvence slov) s kořeny načtených ze souboru v režimu zpracování slov. Tyto problémy se dále dělí na další podproblémy: vybrání správného režimu činnosti podle vstupních parametrů; načítání slov z korpusu, jejich následné ukládání do vybraných datových struktur; ošetření vstupů; výběr algoritmu pro vyhledávání kořenů.

2.1 Výběr režimu

Prvním úkonem je vybrat režim, ve kterém bude program pracovat. Program vyžaduje jeden nebo dva parametry. Prvním parametrem může být buď soubor, či cesta k němu; nebo jím může být slovo, či více slov v uvozovkách. Druhý parametr je doplňující a specifikuje pozdější chování programu.

Je nutné otestovat počet parametrů, při dvou parametrech i jejich kombinaci. V případě jednoho parametru zjistit, zda se jedná o soubor či nikoliv, podle toho lze určit režim. Se dvěma parametry lze užít stejný postup, tedy testovat, zda je první parametr soubor, poté otestovat parametr druhý. Druhý parametr je řetězec o pěti znacích následovaný celočíselnou hodnotou. Dále ověřit, zda je řetězec validní - tedy jedná se o řetězec ”-msl=”nebo ”-msf=”.

Pokud jsou programu předány dva parametry, otestovat jejich kombinaci. Soubor lze kombinovat pouze s ”-msl=”a řetězec slov s ”-msf=”.

2.2 Zpracování vstupů ze souborů

V obou režimech je nutné načítat data ze souboru. Je potřebné zvolit jednotný formát znaků, určit které z nich jsou validní, a které nikoliv. Dále jakým způsobem načítat soubor, zda-li načíst celý soubor najednou a až poté jej zpracovat; nebo jej načítat znak po znaku a zpracovávat rovnou.

Jako jednotný formát znaků jsem zvolil malá písmena. Soubor budu načítat po znacích, které hned po načtení zpracují. Zpracováním znaků se rozumí rozhodnutí, zda se jedná o validní znak a upravení znaku na malé písmeno.

2.3 Datová struktura frekvenčního slovníku

V režimu učení je slova z korpusu a kořeny nutné uložit do vhodné rychle přístupné struktury, jelikož se k nim bude často přistupovat. Tato struktura také musí zvládnout ukládat frekvenci výskytu jednotlivých slov a jejich pořadí.

Jako první se nabízí dynamické pole. Má rychle přístupná data, lze jej v ansi C poměrně snadným způsobem abecedně seřadit. Problémem je ukládání frekvence slov. To lze řešit načtením všech slov, včetně těch, která se opakují, do dynamického pole řetězců, které se poté abecedně seřadí. Následně vytvořit nové dynamické pole, pro které se vytvoří nová struktur, která kromě řetězců, drží i celočíselnou hodnotu. V původním poli spočíst stejné za sebou jdoucí řetězce reprezentující slovo, tuto hodnotu vložit do nově vytvořeného dynamického pole společně s daným řetězcem. Dalším problémem je nutnost držet počet uložených elementů, aby nedošlo k pokusu o přístup k paměti, která nenáleží poli. S počtem elementů v poli souvisí i nutnost zvětšení pole při jeho naplnění, to znamená realokaci paměti.

Další možností je trie. Trie umožňuje uložení řetězce, zároveň počet jeho výskytů. Přístup ke slovům je komplikovanější a více náročný na procesorový čas než u dynamického pole, protože se řetězce musí při každém přístupu skládat z jednotlivých písmen. Při vhodné implementaci trie a algoritmu pro získání řetězců je výhodou abecední pořadí vybíraných řetězců. Největší nevýhodou toho řešení je vyšší algoritmická a implementační náročnost.

Po zvážení jsem dospěl k rozhodnutí, využít pro frekvenční slovník trii. Ačkoli je trie náročnější na implementaci než dynamické pole, lze tento faktor zanedbat, protože jsme ji implementovali na cvičení. Dalším důvodem výběru trie, bylo ukládání počtu výskytů slov, které vyplývá z jejího fungování a tak jej není třeba dodatečně implementovat. Posledním důvodem je abecední pořadí výběru řetězců z trie.

2.4 Datová struktura slovníku

V režimu zpracování slov se je nutné ze souboru načíst jednotlivé kořeny a počet jejich výskytů v korpusu. Pokud je již při načítání porovnán počet výskytů kořenů a MSF (Minimum Stem Frequency - minimální počet výskytů kořene), není nutné jejich počet ukládat, což ušetří paměť a procesorový čas. Jediným

úkolem této struktury je tedy uchovávat řetězce tak, aby k nim byl co nejrychlejší přístup.

Opět se nabízí dynamické pole řetězců. V tomto případě by se řešil pouze počet elementů v poli a realokace paměti. Dynamické pole se jeví jako výhodná datová struktura pro slovník.

Trii lze také užít a díky jejímu využití jako frekvenční slovník, odpadá element implementace a vymýšlení algoritmů. Avšak vyšší náročnost získávání dat z trie, je pro tuto jednoduchou pod-úlohu nežádoucí.

Další možností je spojový seznam, který stejně jako dynamické pole umožňuje rychlý přístup k datům. Pro úsporu paměti je vhodné, a pro tuto úlohu dostačující, implementovat jednosměrný spojový seznam a uložit ukazatel na první prvek v seznamu.

Pro slovník jsem zvolil spojový seznam, protože mi přišel snadný na implementaci a následnou manipulaci. Při předávání slovníku jako parametru funkce, má dynamické pole oproti spojovému seznamu nevýhodu v podobě dvou parametrů (pole, počet prvků). Spojový seznam se předá pouze jako jeden parametr, a to ukazatel na první prvek spojového seznamu.

2.5 Vyhledávání kořenů

Pro vyhledávání kořenů jsem vymyslel vlastní algoritmus. Z frekvenčního slovníku vyberu první a druhé slovo, porovnám jejich podřetězce. V případě, že obě slova obsahují stejný podřetězec, označím jej za kořen. Jelikož hledáme nejdelší společný podřetězec (nejdelší společný kořen), porovnávám nejdříve nejdelší podřetězce, až po neúspěšném hledání společného podřetězce hledám kratší nejdelší podřetězce. Díky tomu, není nutné ukládat jeho délku, ale stačí kořen vrátit. V průběhu zkracování porovnávaných podřetězců kontroluji, zda nejsou kratší než MSL (minimální délka kořene). Grafické znázornění algoritmu viz obrázek 2.

Porovnávám první slovo s každým následujícím slovem ve frekvenčním slovníku. Poté se přesunu na druhé slovo, pro které provedu to samé. Stejným způsobem pro všechna následující slova, jak naznačují vývojové diagramy (3).

Po otestování tohoto algoritmu jsem zjistil, že ačkoli pracuje správně, jeho časová náročnost je příliš vysoká. Proto jsem se rozhodl vyhledat alternativní efektivnější algoritmus.

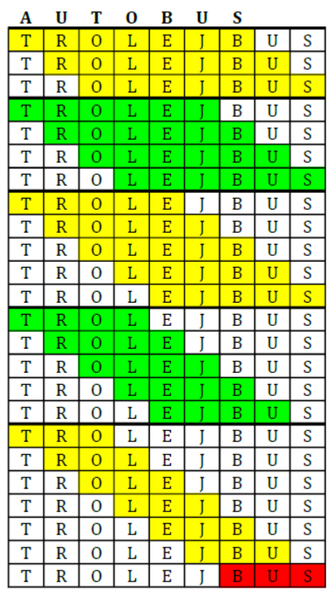
Algoritmus Longest Common Substring (dále pouze LCS) je algoritmus dynamického programování, který pomocí dvourozměrné matice vyhledá, jak z názvu vyplývá, nejdelší společný podřetězec. Základem je matice velikosti $m + 1/n + 1$, kde m je délka jednoho slova a n délka druhého.

Matice je plněna následujícím způsobem - nejdříve jsou na její první sloupec a řádek (sloupec a řádek s indexem 0) vloženy *nuly*. Zbytek matice (od indexu 1; 1) je plněn na základě hodnot na indexu $i - 1; j - 1$ a písmen ve slovech na daném indexu. Pokud se písmena na testovaném indexu $i; j$ shodují, je hodnota z indexu $i - 1; j - 1$ vložena na testovaný index a zvýšena o jedničku. Pokud se písmena neshodují je vložena nula. Ze vzniklé matice (viz obrázek 1) se vybere nejvyšší číslo, uloží se písmeno z daného indexu $i; j$, posune se na $i - 1; j - 1$, opět uloží

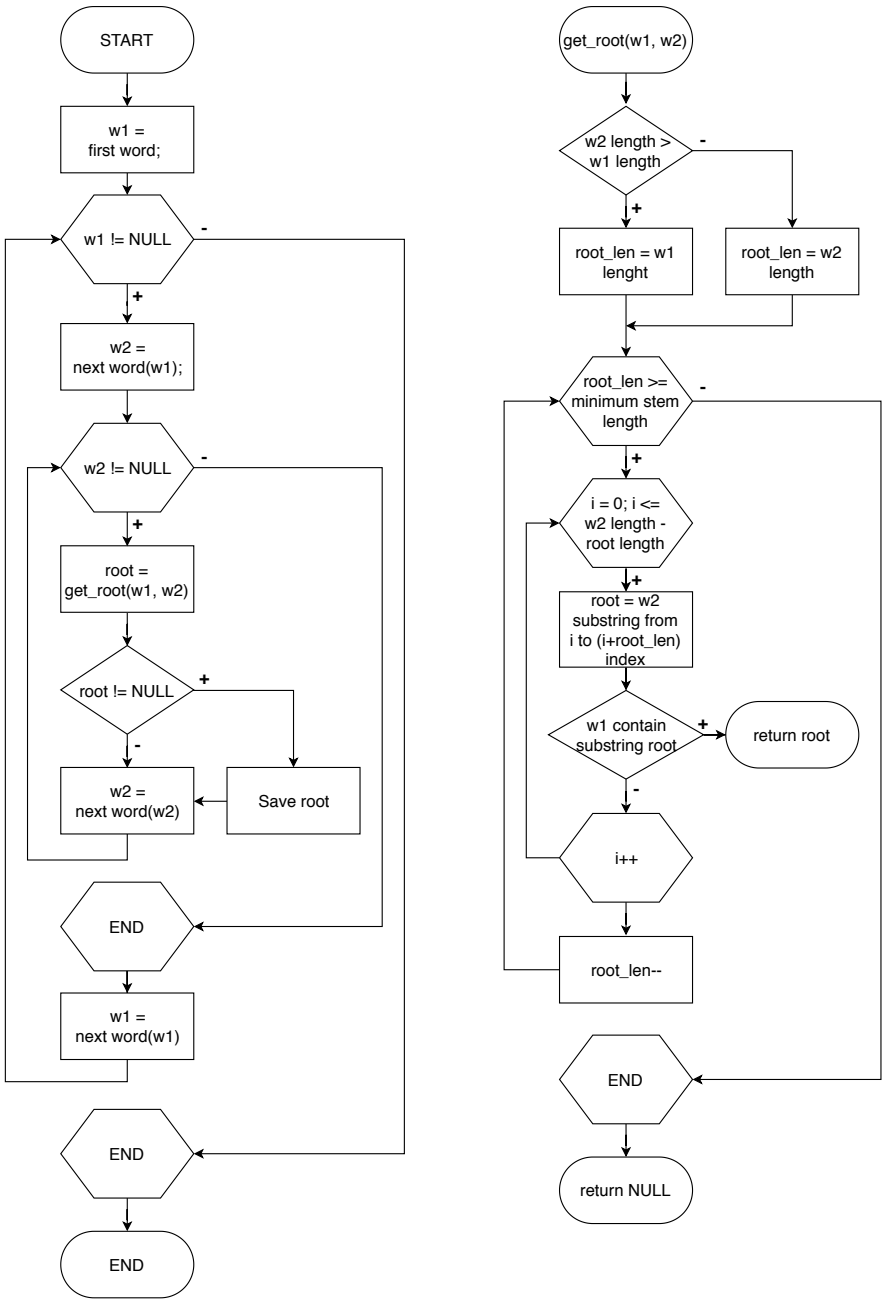
písmeno a stejným způsobem dokud je hodnota vyšší než nula. Takto vzniklý řetězec je nutné obrátit, poté již vzniká nejdelší společný podřetězec.

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

Obrázek 1: LCS matice po vyplnění.



Obrázek 2: Algoritmus vyhledávání kořenů.



Obrázek 3: Vývojový diagram algoritmu vyhledávání kořenů.

2.6 Nalezení kořenů ve slovech

Posledním úkolem je nalezení nejdelších kořenů ve slovech. K tomu stačí porovnat načtené slovo s kořenem, pokud slovo skutečně kořen obsahuje, uložit si jej a takto porovnat slovo s každým kořenem. Je nutné dbát na to, aby byl každý další kořen delší, než původní nalezený. K porovnávání slova a jeho případného kořene lze použít funkci ze základních knihoven ansi C, která zjišťuje zda řetězec obsahuje zadaný podřetězec.

3 Popis implementace

3.1 Datové struktury

3.1.1 Trie

Trie je využívána pro uložení slov načtených z korpusu a z nich získaných kořenů. Datová struktura jednoho uzlu, která je definovaná v souboru *trie.h*, vypadá následovně:

```
typedef struct thenode {
    struct thenode *subtries[CHARSET_LEN];
    int frequency[CHARSET_LEN];
} node;
```

Proměnná *substries* je pole ukazatelů na trie a *frequency* je pole, které určuje počet slov, končící na daný znak. Obě tyto pole mají velikost určenou počtem znaků, ze kterých trie dokáže ukládat řetězce. Každý index reprezentuje číslo, které implicitně reprezentuje znak. Pro obsluhu trie lze použít funkce:

- *create_trie* - Vytvoří jeden uzel trie.
- *insert_to_trie* - Vloží do trie zadaný řetězec.
- *dump_trie* - Vypíše všechny řetězce uložené v trii.
- *get_frequency* - Vrátí počet vložení zadaného řetězce.
- *get_word* - Vrátí první slovo (v abecedním pořadí) v trii.
- *get_next_word* - Vrátí první slovo v trii, které následuje po slově zadaném.
- *free_trie* - Slouží pro uvolnění alokované paměti.

3.1.2 Spojový seznam

Spojový seznam je využit v režimu zpracování slov pro ukládání kořenů. Datová struktura spojového seznamu je definována v souboru *word.h* a vypadá následovně:

```
typedef struct thell_node {
    char *word;
    struct thell_node *next;
} ll_node;
```

Proměnná *word* je řetězec, kam se uloží načtený kořen. *next* odkazuje na další prvek ve spojovém seznamu. Pro jeho obsluhu slouží funkce:

- *add_node* - Přidá na konec spojového seznamu uzel se zadaným řetězcem.
- *free_linked_list* - Pro zadaný a každý následující uzel volá funkci *free_linked_list_node*.
- *free_linked_list_node* - Uvolní zadaný uzel.

word.h kromě obsluhy spojového seznamu obsahuje funkci na zpracování znaku - *process_char*.

3.2 Implementace programu

3.2.1 Vstup a vybrání režimu

Program začíná v souboru *main.c*, který nejdříve zjistí, zda jsou na vstupu správně zadané parametry. Zkontroluje jejich počet a zda případné doplňující parametry odpovídají správnému formátování. Poté zjistí, zda je první parametr soubor, pokusí se jej otevřít, v případě neúspěchu přidá příponu *".txt"*, pokud ani v tomto momentě nelze otevřít, je zvolen režim zpracování slov. V opačném případě režim učení. Soubor, či řetězec se slovy je odeslán do dalšího souboru.

3.2.2 Režim učení

O režim učení se stará soubor *learning.c*, který začne funkcí *learn*, případně *learn_msl*, podle toho, zda-li je zadaný parametr *MSL*, či nikoliv. Funkce *learn* volá funkci *learn_msl*, kam jako *MSL* dodává výchozí hodnotu.

Poté se volá funkce *load_words_msl* načítající slova ze souboru. Slova, která jsou kratší než je minimální požadovaná délka kořenů *(MSL)*, nejsou z důvodu optimalizace ukládána. Ostatní slova, včetně počtu jejich výskytů, se ukládají do trie.

Ve chvíli kdy jsou všechna slova načtená, se začnou procházet a hledat jejich kořeny (Algoritmus *LCS* hledání kořenů viz kapitola 2.5). K tomu slouží funkce *find_roots* zajišťující porovnání všech slov a funkce *get_root*, která porovnává dvě slova a z nich získává kořeny. Kořeny se následně ukládají do trie.

Když jsou všechny kořeny v trii, funkce *save_roots* je spolu s četnostmi uloží do souboru *"stems.dat"*. Poté se zavolají funkce na uvolnění alokované paměti, program se vrátí do funkce *main.c* a ukončí.

3.2.3 Režim zpracování slov

Režim zpracování slov je řízen souborem *word_processing.c*, do kterého je vstupováno pomocí jedné ze dvou funkcí. První funkcí je *process_words*, která volá druhou vstupní funkci *process_words_msf* s výchozí hodnotou pro *MSF*. Poté jsou pomocí funkce *load_roots* načteny kořeny do spojového seznamu. Pokud má kořen menší počet výskytů než je *MSF*, není do seznamu uložen. Po načtení

všech kořenů vrací funkce *load_roots* ukazatel na první prvek vytvořeného spojového seznamu.

Následuje hledání kořenů zadaných slov, což obstarává funkce *find_words_roots*. Ta nejdříve zpracuje jedno slovo ze zadaných a poté jej společně se seznamem kořenů odešle do funkce *get_longest_root*. Tato funkce postupně porovnává kořeny se slovem tak, že zjišťuje, zda slovo obsahuje podřetězec s kořenem. Pokud ano, je uložen. V následujících krocích musí být délka dalších nalezených kořenů ve slově delší. Po porovnání slova se všemi kořeny se vypíše slovo a jeho kořen, případně slovo a nula, která značí, že nebyl nalezen žádný kořen.

Ukázka kódu hledání nejdelšího kořene slova:

```
int find_words_roots(const char *words, ll_node *init_root) {
...
  while (i < words_len && words[i]) {
        w = (char *) malloc(sizeof(char) * words_len + 1);
        if (!w)
            return OUT_OF_MEMORY_ERR;

        if (words[i] == ASCII_QUOT_MARK)
            i++;

        // Load one word
        n = 0;
        while (i < words_len && words[i] != SEPARATOR && words[i]
    != ASCII_QUOT_MARK) {
            w[n] = words[i];
            i++;
            n++;
        }
        w[n] = '\0';

        // Find ll_node of the word and print it
        w_len = strlen(w);
        if (w_len > 0) {
            temp = get_longest_root(w, init_root);
            printf("%s -> %s\n", w, temp);
        }
        i++;
        free(w);
    }
...
}

char *get_longest_root(char *word, ll_node *roots) {
...
  // while there are roots search
    while (roots->word) {
        // if roots->word is root of the word
        if (strstr(word, roots->word)) {
            root = roots->word;
            max_root_len = strlen(root);
        }

        if (!roots->next)
            break;

```

```
45          while (roots->next) {
46              roots = roots->next;
47              if (strlen(roots->word) > max_root_len)
48                  break;
49          }
50      }
51  ...
52  }
```

Poté se zavolají funkce na uvolnění alokované paměti, program se vrátí do funkce *main.c* a ukončí.

4 Uživatelská příručka

4.1 Překlad programu

4.1.1 Windows

Překlad na systémech Windows vyžaduje instalaci překladačů s podporou *make*. Poté z příkazové řádky v adresáři se zdrojovými kódy programu zadá příkaz *make -f makefile*, který program přeloží. Pro funkčnost je nutné, aby byla správně nastavena cestu k překladači v systému.

4.1.2 Linux

V systémech Linux stačí do terminálu otevřeném v adresáři se zdrojovými kódy zadat příkaz *make*.

4.2 Spuštění programu

Spuštění programu se liší podle režimu, který je vyžadován.

4.2.1 Režim učení

Pro režim učení se použije příkaz:

sistem.exe <corpus-file>[-msl=<celé číslo>]

corpus-file je název souboru, nebo celá cesta k němu, jehož přípona nemusí být uvedena. Argument *-msl=* je nepovinný a udává minimální délku kořenu, který se uloží do výstupního souboru.

4.2.2 Režim zpracování slov

Pro režim zpracování slov se použije příkaz:

sistem.exe <["]word-sequence["]>[-msf=<celé číslo>]

Argument *word-sequence* představuje slovo, či slova, ke kterým bude program přiřazovat kořeny. Tato část musí být v uvozovkách. Nepovinný argument *-msf=* v tomto případě udává minimální počet výskytu kořene, který je možný slovu přiřadit.

5 Závěr

Program je funkční, ale k vyhledávání kořenů jsem zvolil nevhodný algoritmus. Časová náročnost hledání kořenů je velmi vysoká, z toho důvodu se ve validátoru program řádně neukončí. Datová struktura trie, také nebyla nejvhodnější - i přes její funkční implementaci, bylo poměrně složité vytvořit funkci *get_next_word*.