Semestrální práce z KIV/PC

JEDNODUCHÝ STEMMER

Mukanova Zhanel
A16B0087P
mukanova@students.zcu.cz

15. 12. 2018

Contents

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.

Stemmer se bude spouštět příkazem sistem.exe corpus-file | ["]word-sequence["]i [-msl=celé číslo] [-msf=celé číslo]. Symbol corpus-file zastupuje jméno vstupního textového souboru s korpusem, tj. velkým množstvím textu, který se použije k natrénování stemmeru. Přípona souboru nemusí být uvedena; pokud uvedena není, předpokládejte, že má soubor příponu .txt. Symbol word-sequence zastupuje slovo nebo sekvenci slov, k nimž má stemmer určit kořeny. Režim činnosti programu je dán předaným parametrem: Je-li parametrem jméno (a případní 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. Program může být spuštěn se dvěma nepovinnými parametry:

-msl Nepovinný parametr -msl=celé číslo specifikuje minimální délku kořene slova (msl = Minimum Stem Length), který bude uložen do databáze kořenů. Není-li tento parametr předán, použije se implicitní minimální délka kořene 3 znaky. Tento parametr je tedy zřejmě použitelný jen v kombinaci s cestou ke korpusu, tedy v režimu učení stemmeru.

-msf Nepovinný parametr -msf=celé číslo určuje minimální počet výskytů příslušného kořene (msf = Minimum Stem Frequency). Pokud se tento kořen v korpusu nevyskytl alespoň tolikrát, kolik je určeno tímto parametrem, nepoužije se při zpracování slov, tj. stemmer nemůže u zádného zpracovávaného slova oznámit, že tento kořen je kořenem předmětného slova. Není-li tento parametr předán, použije se implicitní minimální počet výskytů kořene 10×. Tento parametr je tedy zřejmě použitelný jen v kombinaci se slovem nebo sekvencí slov, tedy v režimu zpracování slov.

Celé zadání je na: *https://www.kiv.zcu.cz/studies/predmety/pc/doc/work/sw2018-03.pdf*

2 Analýza úlohy

Podle zadání jsme museli vytvořit program, který by byl schopen najít kořeny k zadaným slovům. Prvním problémem se kterým jsem se setkala byl správný výběr **datové struktury**. Prvním požadavkém bylo ukládaní seřazených slov podle ASCII hodnot. Druhým nejdůležitějším požadavkem byla rychlost vkládaní a získávání slov ze slovníku. Měla jsem několik variant datových struktur.

Hlavní datové struktury

- **Hash-table**
  Hashovací tabulka (hash table) je datová struktura, která slouží k ukládání dvojic klíč-hodnota. Tato tabulka kombinuje výhody vyhledávání pomocí indexu a procházení seznamu. Má nízké nároky na paměť a složitost *O(1)*.
  Pokud bych chtěla pouze uložit slova a pak zkontrolovat, zda je mezi nimi hledané slovo nebo ne, pak by standardní hash tabulka byla rozumnou volbou. Pokud by počet položek seznamu byl znám předem, použila bych ideální hash pro dosažení nejvyššího výkonu a optimální velikosti uložených dat.
  Hashovací tabulka je velmi efektivní vyhledávací metoda, avšak musíme vědět, kdy a jak ji použít. Musíme ještě zvážit alternativy jako třeba binární vyhledávací strom. Ten je v porovnání s hashovací tabulkou o něco pomalejší, pokud uvažujeme ideální případ, na druhou stranu pro případ nejhorší strom vítězí. Hashovací tabulka nám navíc neumožňuje vyhledávat klíče v určitém intervalu nebo jenom částečně specifikované klíče - např. všechny řetězce s určitou předponou.

- **Prifix tree**
  Prefix tree nebo Trie je strom, který umožňuje uchovávat páry klíč - hodnota a získávat hodnoty podle klíče. Klíčem však musí být řetězec. Vyhledání dat v trie je rychlejší v nejhorším případě, tj. O(m), oproti hašovací tabulce s kolizemi O(N), typicky však O(1).
  Prefix tree v každém uzlu obsahuje všechny podřetězce, kterými může pokračovat řetězec v dosud prohledané cestě. Všechny následníci uzly mají společný prefix, který je shodný s řetězcem přiřazeným k danému uzlu.
  Prefix tree je vhodná datová struktura v případě rychlého vyhledávání předpony slov, i když to může být trochu neefektivní z hlediska ve-

likosti uložených souborů. Tato datová struktura také podporuje rychlé vkládání a odstraňování. Navíc Prefix tree nepotřebuje hašovací funkce nebo její změna při přidávání dalších klíčů a ještě může poskytnout abecední řazení záznamů podle klíče, co v hash-tabulkách chybí.

- **Dalsí stromy**
  BK-Tree je skvělá datová struktura pro vytváření "slovníku" podobných slov, která používá vzdáleností mezi slovy. Ta vzdálenost je minimální počet úprav potřebných k přeměně jednoho řetězce na druhý. Kdybych chtěla použít slovník pro operace, jako je kontrola pravopisu, kde je potřebné najít slova, která nejsou podobná ostatním, je BK-strom skvělou volbou.

Po dlouhém rozmýšlení jsem si zvolila Trie jako vhodnou datovou strukturu pro toto zadání. Zaprvé podle zadání jsem potřebovala uschovávat slova v abecedním pořadí, co neposkytuje Hash-tabulka. Zadruhé nepotřebuji vymýšlet Hash-funkce, a můžu rovnou ukládat slova do datové struktury bez vzniku kolizí.

Další problém se kterým jsem se setkala bylo porovnání každého slova slovníku s ostatními. Kvůli tomu, že iterační funkce v datové struktuře Trie je obvyklé rekurzivní, musela jsem si zvolit nějakou **pomocnou datovou strukturu** pro procházení skrz celý slovník.

Pomocné datové struktury

- **Obecné pole retezcu**
  Pole je datová struktura, která sdružuje daný vždy konečný počet prvků stejného datového typu. K jednotlivým prvkům pole se přistupuje pomocí jejich indexu. Velikost pole zůstává při běhu programu neměnná a zvětšení pole je časově náročná operace.

- **Linked list**
  Spojový seznam (Linked list) je dynamická datová struktura, vzdáleně podobná poli (umožňuje uchovat velké množství hodnot ale jiným způsobem), obsahující jednu a více datových položek (struktur) stejného typu, které jsou navzájem lineárně provázány vzájemnými odkazy pomocí ukazatelů nebo referencí.

Hlávním úkolem této semestrální práce bylo najít kořeny slov. Existuje

mnoho různých **algoritmu** nacházení kořenů. Já jsem potřebovala jen ty, co hledaly nejdelší společné podřetězce.

Algoritmy

- **Longest Common Subsequence (LCS)**
  LCS algoritmus je jedním ze způsobů jak posuzovat podobnost mezi dvěma řetězci. Ale neurčuje kořeny, určuje jenom "společná písmena" slov.

- **Longest Common Substring (LCS)**
  Algoritmus zváží všechny podřetězce prvního slova a pro každý podřetězec zkontroluje, zda existuje podřetězec ve druhém slově. Vyhledá podřetězec maximální délky.

Z těchto dvou algoritmu jsem si zvolila **Longest Common Substring (LCS)**, protože on nachází společné podřetězce, co jsem potřebovala podle zadání. Na jeho implementaci jsem potřebovala vytvořit 2d matici, do které bych mohla ukládat informace o společných podřetězců.

3 Popis implementace

Obsah programu

1. **main.c + library.h**
   Hlavní spustitelný soubor programu, který obsahuje funkci *main()*. Soubor má v sobě všechny funkce na parsování vstupních parametru a spuštění příslušných metod pro dva možných režimy programu (Režim učení a Režim zpracování slov).
   Hlavičkový soubor **library.h** obsahuje konstanty (IMPLICITNI_MIN_DELKA_KORENE, IMPLICITNI_MIN_POCET_VYSKYTU_KORENE), určující implicitní minimální délky kořenů a počet jejich výskytů. Také konstanty (DICTIONARY, STEMS) cest k souborům *stems.dat* a *dictionary.txt*.

   Pomocné funkce.

   - **get_parameter_value()** - funkce pro získání celočíselného parametru. (např. -msl=9, získame 9)
   - **match()** - funkce pro pracování s regulární výrazy

   Trénování stemmeru.

   - **create_dictionary(char *corpus_file_path)** - funkce přijímá jako vstupní parametr jméno vstupního textového souboru s korpusem, tj. velkým množstvím textu, který se použije k natrénování stemmeru, dále čté tento soubor a kážde slovo ukládá do datové struktury *Trie*.
   - **create_words_array(Word *root, char prefix[])** - rekurzívní funkce, která prochází přes celou datovou strukturu Trie a ukládá do LinkedListu seřezený slovník slov.
   - **create_stems_file(int min_delka_korene)** - funkce vytváří nové Trie, pro kořeny slov. Pouští funkci *compare_strings()* a ukládá do souboru *stems.dat* již seřazený podle ASCII hodnot slovník kořenů slov.
   - **compare_strings(FILE *fp, Trie* trie, List *head, int max_delka_slova)** - funkce dvakrát prochází přes LinkedList a pro káždé dva slovy pouští funkci *LCS_algorithm()*, která pomocí LCS algoritmu nachází nejdelší společný kořen a ukládá ho do datové struktury Trie.

   Režim zpracování slov.

- **create_stems_dictionary(const char *filename)** - funkce přijíimá jako vtupní parametr jméno souboru stems.dat a ukádá káždý kořen a jeho počet výkytů do datove truktury Trie.
- **find_stems(char *word_sequence, int msf_value)** - funkce pro zádáné jako argument káždé slovo pouští funkci *find_longest_word(char *str)* a výpisuje do konzoli společný nejdelší kořen s dostatečným počtem výskytů.

2. **trie.c + trie.h**
   Soubory, které obsahují datovou strukturu Trie a všechné potřebné funkce jako vkládaní, procházení, inicializace, uvolnění a td. Také zahrnuje funkce jako *str_clean_eng()* a *str_clean_cz()*, které odstraňují z řetězce příslušného jazyka zbytečné znaky jako číslicí a td.[1]

   - **Trie* trie_initialize()** - funkce pro inicializaci root node a allokování pamětí na datovou strukturu Trie.
   - **Word* get_new_trie_node()** - funkce vrátí Trie node s naallokovaným místem v pamětí.
   - **void insert(Trie* trie, char* str)** - vkládání slova do Trie
   - **void display_trie(FILE *fp, Word *root, char prefix[])** - zápis celého Trie do souboru.
   - **void free_t (Trie* trie)** - obalující funkce k funkci *trie_free()*.
   - **void trie_free (Word *root)** - rekurzivní funkce pro uvolnění pamětí každého úzlu Trie.
   - **int is_leaf(Word* node)** - konroluje zda zadaný úzel je poslední (list).
   - **void str_clean_cz (char* src)** - odstaňuje z řetězce všechné znaky z ASCII tabulky od 0 do 65. (funguje jen pro český text)
   - **void str_clean_eng (char* src)** - pomocí standartní funkce *isalpha()* nechává v řetězci jenom písmena. (funguje jen pro anglický text text)
   - **void find_stem(Word* root, char *word, char prefix[], int msf_value, char *MAX_STR, size_t stemSize)** - rekurzivní funkce, která prochází přes celý slovník kořenů a pomocí standartní funkci *strstr()* nachází nejdelší společný kořen s dostatečným počtem výskytů.

[1] Je nutně měnit podle vstupního jazyka pro lepší výsledky

- **char *find_longest_word (char *str)** - funkce najde v zadaném řetězci nejdelší slovo.

3. **lcs.c + lcs.h**
   Soubory zahrnující samotný algoritmus Longest Common Substring, který vyhledává nejdelší společný kořen, funkci inicializace a uvolnění matici pro výpočet nejdelšího společného kořenu.

   - **int LCS_algorithm(char *s1, char *s2, char **longest_common_substring)** - samotný algoritmus LCS (Longest Common Substring).
   - **void init_lcs_matrix(int s1_length, int s2_length)** - inicializace matici pro LCS algoritmus.
   - **void destroy_lcs_matrix ()** - uvolnění pamětí matici.

4. **linkedList.c + linkedList.h**
   Soubory obsahující implementace spojového seznámu, funkci pro vyhledávaní, vkládaní a procházení přes seznam.

   - **List * make_list()** - funkce výtváří datovou strukturu LinkedList, allokuje paměť a nastavuje root node.
   - **Node * create_node(char *data)** - funkce allokuje pamět pro úzel a vkládá data do úzlu.
   - **void insert_to_list(char *data, List *list)** - funkce vkládá do LinkedListu nový úzel.
   - **void destroy(List * list)** - uvolňuje celý LinkedList.

Postup

Mým prvním krokem bylo ošetřit vstupní parametry. Nejdůležitější bylo zjistit podle parametrů co musí program dělat. Parametry *-msl* a *-msf* nejsou podle zadání povinné, a proto jsem nemohla zjistit podle těchto parametrů, zda musí program fungovat v režimu **učení** nebo **zpracování**. Na vyřešení tohoto problému jsem použila regulární výrazy, které by mohly zjistit zda je vstupní parametr cesta k souboru.

Režim učení

Po kontrole parametrů jsem začala implementovat datovou strukturu Trie. Vybrala jsem tuto strukturu pro její rychlé zpracování a uložení slov podle ASCII hodnot, což jsem potřebovala kvůli zadání. První problém se kterým

jsem se setkala při implementaci Trie byl problém s kódováním. První můj Trie mohl ukládat do sebe jen 127 ASCII hodnot, a proto při vkládaní českých slov se program ukončil chybovým výpisem SIGNAL 11. Zvýšila jsem hodnotu ASCII symbolů z 127 do 256, což mě povolilo ukládat nejen anglické znaky, ale i české. Ale dostala jsem jiný problém. V hodnotě 256 nejsou jen písmena abeced, ale i různé znaky a číslice. To jsem vyřešila tím, že při vkládaní nového slova do Trie jsem kontrolovala jestli slovo neobsahuje ASCII hodnoty od 0 do 65, ostatní znaky jsem dělala nezápornými a dávala jsem je do nížního registru.
Po vytvoření slovníku jsem musela porovnat každé slovo ve slovníku a vytvořit slovník kořenů. Ale procházet skrz celý Trie dvakrát by byla velmi náročná akce, protože ve svém programu jsem použila rekurzívní funkce procházení stromem. Takže jsem jedním procházením Trie vytvořila už seřazeny LinkedList pomocí kterého jsem porovnávala každá dva slova v slovníku. Na hledání kořenů jsem použila algoritmus LCS (longest common substring). Každý kořen jsem ukládala zase do nového Trie a potom už seřazeny slovník jsem ukládala do souboru *stems.dat*.

Režim zpracování slov

Po spuštění programu v režimu zpracování slov, program nejprvé musí přečist už existující soubor *stems.dat*, který obsahuje databázi kořenů. Během čtení suboru jsem ukládala každý kořen a počet jeho výskytů do datové struktury Trie. Dalším krokem bylo porovnat každé zadané slovo se slovníkem. Podle zadání jsem musela najít nejdelší společný kořen, který by měl nejvýsši ASCII hodnotu. Slova jsem porovnávala pomocí standardní funkce v C *strstr()*, která by měla vrátit nenulovou hodnotu v případě jestli slovo obsahuje kořen. Pak jsem porovnávala velikost předchozího slova se slovem aktuálním a ukládala jsem do bufferu nejdelší společný kořen.

4 Uživatelská příručka

Semestrální práce je vytvořena programovacím jazykem ANSI C a lze spustit různými způsoby. Hlavním požadavkem pro spuštění bude nainstalovaný kompilátor pro jazyk C *GCC*.

Postup spuštění programu přes Linuxový Terminal

1. Příkazem *"cd path/to/program"* přijdeme do složky s programem.

2. Pro kompilace a generování spustitelného souboru použijeme příkaz *"make"*, který vytvoří spustitelný soubor *sistem.exe*

3. Pro spuštění programu v režimu učení použijeme příkaz *"./sistem.exe path/to/corpus/file -msl=[cislo]"*. Parametr *"-msl=[cislo]"* není povinný. Určuje minimální délku kořene. Implicitní minimální délka kořene 3 znaky. Program vytvoří soubor stems.dat, který bude obsahovat kořeny slov.

4. Pro spuštění programu v režimu zpracování slov použijeme příkaz *"./sistem.exe "sekvence slov" -msf=[cislo]"*. Parametr *"-msf=[cislo]"* není povinny. Určuje minimální počet výskytů příslušného kořene. Program vypíše do Terminalu zadane slovo a jeho koren.

Postup spuštění programu v programovácím prostředí CLion

1. Pro spuštění programu v režimu učení potřebujeme nejprve nastavit vstupné parametry. Ve složce Run->Edit Configurations..-> Program arguments dosadíme příkaz *"./sistem.exe path/to/corpus/file -msl=[cislo]"*. Parametr *"-msl=[cislo]"* není povinný. Určuje minimální délku kořene. Implicitní minimální délka kořene 3 znaky. Program vytvoří soubor stems.dat, který bude obsahovat kořeny slov.

2. Pro spuštění programu v režimu zpracování slov nejprve nastavit vstupné parametry. Ve složce Run->Edit Configurations..-> Program arguments dosadíme příkaz *"./sistem.exe "sekvence slov" -msf=[cislo]"*. Parametr *"-msf=[cislo]"* není povinny. Určuje minimální počet výskytů příslušného kořene. Program vypíše do Terminálu zadané slovo a jeho kořen.

5 Závěr

Dané zadání na začátku mně přišlo jednoduché. Ale během implementování jsem se dozvěděla, že pracování s řetězci v jazyce C je docela těžká akce. Díky tomuto zadání jsem se seznámila s různými algoritmy a dočetla se mnoho o kódování. Tento úkol mi předal cenné zkušenosti v jazyce C. Kvůli obtížnosti zadání jsem musela všechno prostudovat a použít své znalosti ve svém programu. Myslím si, že zadání semestrální práce bylo splněno a tímto považuji semestrální práci za splněnou. Jediná výjimkou je, že aby program správně pracoval i s anglickým i s českým textem, musíme v programu v souboru Trie.c -> insert() změnit funkce *str clean cz()* na *str clean eng()*.