Doplňování Diakritiky

Denis Leskovar

Revize

Vytvořeno: 9. 4. 2020
Verze dokumentu: 05
Změněno: 17. 4. 2020

Poznámky

17. 4. 2020 - Doplněny obrázky a odkazy na zdrojové soubory
13. 4. 2020 - Popsání vnitřní stuktury programu
10. 4. 2020 - Doplnění způsobu řešení projektu
9. 4. 2020 - Vytvoření dokumentu

Stručné zadání

Cíl projektu

Hlavním úkolem projektu je vytvořit program, který dostatečně spolehlivě doplní do vstupního textu diakritiku.
Program rozpozná chybné výskyty za pomoci slovníku, vůči němuž ověřuje jednotlivá slova, a díky pravděpodobnostním metodám, viz níže.

Popis funkcionality

Program slouží jako nástroj, který přečte vstupní soubor, doplní do textu diakritiku a výstup uloží jako nový soubor.

Přesné zadání

Projekt si klade za cíl vytvoření programu, který ze vstupu zpracovává text, v němž do libovolné míry chybí diakritika. Výstupem programu je zpracovaný text, do nějž byla diakritika doplněna.
Program může využít dvou externích zdrojů: slovníku a korpusu.
K dosažení zadaného cíle program nasadí pravděpodobnostní a statistické metody - chybné výskyty nahradí za ty nejvíce pravděpodobné, pravděpodobnost přitom plyne z trigramového modelu (pravděpodobnost výskytu dané trojice slov v daném pořadí). Tyto pravděpodobnosti lze získat zpracováním zvoleného korpusu.

Popis funkcionality

Programu je poskytnut vstup (ve formě souboru), který program přečte slovo po slově a za pomocí připravených dat ze slovníku a pravděpodobnostní statistiky slova nahradí jejich nejpravděpodobnější variantou. Výstupní text uloží do nového souboru či na standardní výstup, původní soubor ponechá nepozměněný.

Uživatelské rozhraní

Uživatel interaktuje s programem na příkazové řádce pomocí přepínačů. Ty při spuštění specifikují chování programu a případně nabízejí další dialog.

- -s --silent (Program uživatele neupozorní na cizojazyčný text.)
- -c --conflict (Program nabídne uživateli možnost výběru správného tvaru slova u chybných tvarů, u nichž se doplnění nezdá jednoznačné.)
- -i --install (Program připraví potřebné externí soubory pro jeho správné fungovaní.)

Funkcionální požadavky

- Rozpoznat slova s chybnou diakritikou.
- Chybné slovo s co největší přesností nahradit.
- Poskytnout uživateli přepínač pro revizi konfliktních slov.
- Upozornit uživatele, pokud je vstupní text psán v cizím jazyce.
- V revizním režimu zobrazit konfliktní slovo včetně jeho kontextu a nabídnout uživateli možnost výběru správného tvaru slova.

Zvolený způsob řešení

Na základě jediného externího zdroje (značkovaný korpus Syn2015) se program snaží připravit všechna nutná data pro jeho fungování.
Aktuální řešení z korpusu vybere jeho přečtením množinu všech unikátních slov, kterou uloží do souboru na disk jako slovník. Slovník se pak při každém běhu načítá do paměti do obousměrné mapy, která slova mapuje na celá čísla a naopak. Poté opětovně přečte korpus a zachytí všechny trojice po sobě jdoucích slov v textu. Pokud se daná trojice vyskytuje v textu vícekrát, zvýší se její počet výskytů. Všechny trojice včetně jejich frekvence výskytu je opět uložena na disk do souboru 'model'. Pro snížení velikosti tohoto souboru jsou trojice "zahashovány" do 16B posloupností. Trojice jsou v modelu uloženy souvisle, tedy všechny trojice se stejným slovem uprostřed leží za sebou.
Pro rychlé hledání vytvoří program také 'komprimovaný model' obsahující dvojice <'slovo', 'číslo první řádky v modelu začínající tímto slovem'>. Ten při každém spuštění přečte a uloží dvojice do mapy pro rychlé vyhledávání za běhu - program předloží slovo, mapa odpoví číslem řádku, na němž posloupnost všech trojic s daným slovem uprostřed začíná.
Po zpracování externích dat lze program používat tak, jak bylo původně zamýšleno. Ze vstupu přečte text, zpracuje ho po trojicích slov, do níž pomocí souboru z disku doplní diakritiku. Přestože se zpracovává po trojicích, okénko trojice se posouvá vždy o jedno slovo **(obr. 1)**, tedy všechna slova kromě prvních a posledních dvou jsou brána v potaz pro výpočet 3x, diakritika se do nich ovšem doplní jen jednou. Pro každé slovo se vygeneruje kontejner všech jeho možných variant - do písmen, kterým chybí diakritika, se diakritika doplní. Slova jsou "zahashována" do universa čísel, které zároveň slouží jako offsety v souboru 'model'. Pro každou trojici variant, která není nesmyslná (slova existují ve slovníku), se v souboru 'model' vyhledává četnost celé trojice. Pokud by se trojice v modelu nevyskytovala, hledají se obě dvojice a nakonec i individuální slovo.
Trojice se zpracovávají nezávisle, každá na vlastním vlákně. Výsledek výpočtu se ukládá do společného kontejneru, odkud se výsledný text sekvenčně vytiskne na výstup.

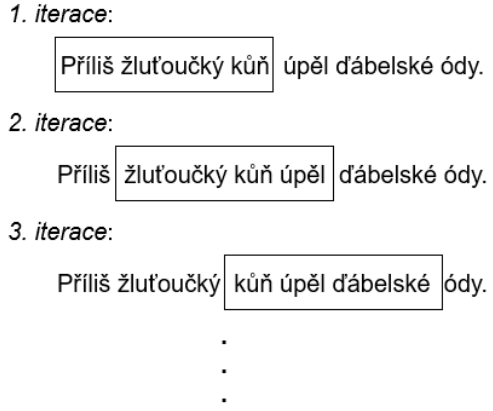

Figure 1: Postup při zpracování vstupu

Diskuze zvoleného řešení

Trigramový statistický model v lingvistice využívá předzpracování dlouhých textů, ze kterých si ke každé možné trojici slov pamatujeme pravděpodobnost jejího výskytu. Pro všechny trojice, které se v dlouhých textech nevyskytují, nastavíme pravděpodobnost na hodnotu velmi blízkou nule, aby ve výpočtech využívajících tento model nevycházely nulové pravděpodobnosti.
Nevýhodou tohoto řešení je především velikost všech dat, která si musíme pamatovat. Pro n slov v jazyce existuje n^3 možných trojic. To je problém, který je nutný ve všech přístupech vyřešit.
Aktuální řešení spoléhá na dostupnost obou externích souborů - slovníku a zpracovaného korpusu. Je také nutno zmínit, že program bude fungovat s jakoukoliv kombinací slovníku a korpusu, pokud jsou dodány ve správných formátech a program využije přípravné funkce a procedury pro vygenerování pomocných souborů. Toho lze využít tehdy, pokud by vznikly potíže s licencemi korpusu či slovníku.
Slovník, jež je k programu dodán v lidsky čitelném textovém souboru, obsahuje téměř všechny tvary běžných českých slov bez jakéhokoliv morfologického značkování. Obsah slovníku je složen z jednotlivých slov použitého korpusu. Jeho velikost přesahuje 1,8 milionů slov (včetně speciálních znaků a číslovek). Při každém běhu se načítá do paměti do obousměrné hashovací tabulky, kde číslo řádku slova odpovídá celočíselnému klíči daného slova.
Korpus byl zvolen tak, aby obsahoval co největší počet tvarů slov a umožnil tak algoritmu pracovat bez nutnosti slova skloňovat či časovat. Aby jeho velikost nebyla zbytečně velká, zvolil jsem korpus reprezentační češtiny Syn2015, který celkem obsahuje zhruba 100 milionů slov.
Slovník slouží především k ověření, zda vstupní slovo je opravdu českým slovem. Pokud by nebylo, můžeme se pokusit ve slovníku najít jeho variantu doplněnou o diakritiku.
Jelikož má korpus v základním formátu přes 10 GB, je nutno ho zpracovat. Nejen kvůli tomu, aby byla jeho velikost menší, ale také proto, aby na jeho základě šla vypozorovat četnost výskytů jednotlivých slov a jejich dvojic či trojic. Formát korpusu **(obr. 2)** lze snadno převést do běžného souvislého textu. Takový soubor v kódování UTF-8 zabere jen 720 MB.
Jelikož chceme pro každou trojici v korpusu uložit její četnost, měli bychom v nejhorším případě soubor o velikosti přes $3 * 720$ MB pro zapsání trojic a $100000000 * 2$ nebo 4 B pro uložení četností. Dodávat takto velký soubor k jednoduchému nástroji je velká zátěž.
Využil jsem slovníku a již přítomné obousměrné mapy a každé slovo namapoval na celé číslo. Jelikož je slov pouze 1,8 milionů, lze takový formát s přehledem vyjádřit pomocí 4B. Ve finále můžeme libovolnou trojici slov zapsat pomocí 4 * 4B, což ušetří v průměru 9B pro každou trojici. Trojice jsou pak uloženy ve formátu 'prostřední slovo' 'levé slovo' 'pravé slovo' 'četnost'.
V takto formátovaném modelu s pevnými offsety pro každou hodnotu lze snadno vyhledávat pomocí std::istream::seekg(2). Toho program využívá při zpracovávání jednotlivých trojic. Slova se převedou na celá čísla, pomocí nichž se spočítá offset, na kterém začínají trojice. Jelikož doplňujeme diakritiku do prostředního slova trojice, začíná každá řádka trojice v modelu prostředním slovem. Jakmile se v souboru přesuneme na daný offset, čteme

jednotlivé řádky modelu a pro každou možnou diakritickou variantu trojice ověřujeme, zda aktuální řádka obsahuje hledanou variantu. Pokud ano, přidáme ji do kontejneru výsledných variant. Pokud ne, pokračujeme ve čtení modelu, dokud neprojdeme všechny trojice se zpracovávaným slovem uprostřed. Pokud by kontejner výsledných variant byl prázdný, hledáme nejpravděpodobnější dvojici variant z dané trojice (buďto <'levé slovo', 'prostřední slovo'>, nebo <'prostřední slovo', 'pravé slovo'>). Pokud stále nenalézáme žádnou variantu, která by se v modelu vyskytovala, vybereme nejpravděpodobnější variantu samotného prostředního slova stejným způsobem, jakým tomu bylo u dvojic a trojic.

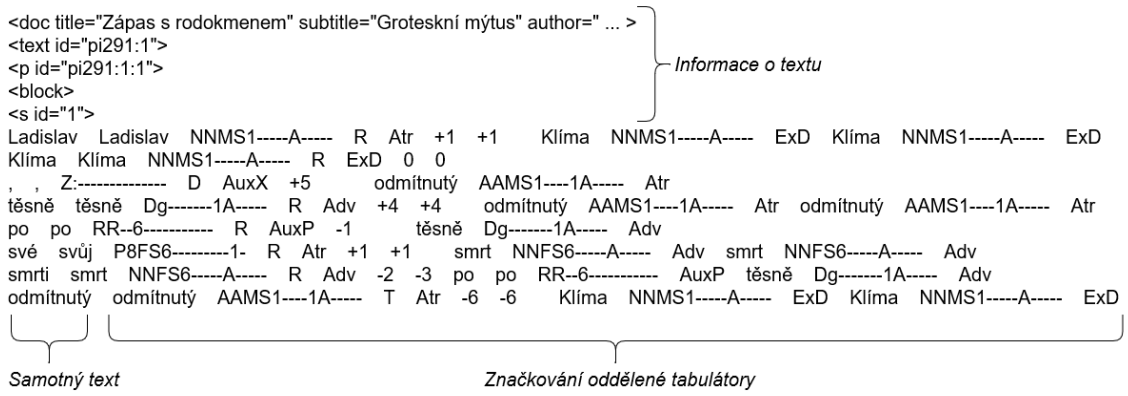

Figure 2: Formát zvoleného korpusu

Jelikož zpracování jedné trojice obnáší vyzkoušení mnoha variant, z nichž se každá hledá na disku, je tento proces velmi zdlouhavý. Na některých systémech ho lze ovšem urychlit vícevláknovým počítáním. Předpokládá se nejen dostupnost fyzických vláken, ale také dostatečně rychlé úložné médium (NVME/m.2 SSD). To především proto, že přístup několika vláken na jeden disk může přinést značnou režii při každém pokusu o čtení.
Implementace více vláken přináší další nutný kompromis - ačkoliv se dvě sousední dvojice prolínají, každou z nich je nuté zpracovat zvlášť. Doplněním diakritiky do prvního slova nijak neovlivní to, jak bude doplněna diakritika do slova druhého. Takové chování může být benefitem, pokud by chybné doplnění do prvního slova znamenalo chybné doplnění do druhého - zabránili bychom kaskádě chyb. Naopak, pokud by správné doplnění prvního slova předešlo chybě ve slově druhém, pak této chybě šlo v sekvenčním zpracování předejít.
Přestože se způsob řešení jeví naivní, slouží jako platforma ke složitějšímu vyhodnocování. Pro jednu trojici můžeme sestavit formuli, která určí nejpravděpodobnější variantu prostředního slova na základě celé trojice, obou dvojic i individuálního slova. Pokud se chceme dívat na sousední trojice, lze přijít se vzorcem, který bere v potaz průnik sousedních trojic a na základě jeho upravuje pravděpodobnost slov v tomto průniku. Chytrým přístupem by také mohl vzniknout způsob, jak získat optimální pravděpodobnosti pro celý text pomocí dynamického programování - ten by se lišil od toho aktuálního tím, že by doplňovaný text stavěl postupně a bral ho v potaz jako celek, nikoliv jako posloupnost jednotlivých slov, do nichž se doplňuje hladovým způsobem.

Program

Strukturu programu lze snadno popsat pomocí jeho hlavičkových souborů.

Utility pro zpracovávání wchar_t

Ukládání diakritiky se sebou nese dilema, zda použít osmibitové kódování typu Latin a obdobných, či se ponořit do kódování typu Unicode.
Druhá zmíněná možnost je přenositelnější a v jiných jazycích dobře podporovaná, nicméně C++ jen krkolomně umožňuje transparentní práci s UTF.
Program cílí na UTF-8, tedy předpokládá vstupní soubor v tomto kódování a výstup programu je zakódován stejným způsobem. Využívá k tomu širokých řetězců std::wstring a vícebytových znaků wchar_t. Ačkoliv práce s řetězci není příliš rychlá, standardní knihovna poskytuje příjemné prostředí pro jejich zpracovávání. I přesto bylo potřeba doplnit některé funkce a procedury.

Práce s diakritikou

Funkce standardní knihovny na převod mezi velkými a malými písmeny bohužel nefungovaly, proto bylo potřeba vytvořit sadu funkcí, která by již existující knihovnu doplnila.
Vznikly tak funkce rozpoznávající písmena s diakritikou nebo ta, která diakritiku mít mohou, či funkce na převody na velká a malá písmena. Mimo to lze pro dané písmeno získat jeho diakritizované varianty.
Kód lze nalézt v souborech WideCharUtilities.h a WideCharUtilities.cpp

Příprava slov

Jelikož algoritmus předpokládá vstupní slova o daném formátu (pouze malá písmena a žádné formátovací znaky či čárky, tečky nebo uvozovky), je potřeba je dostat do tohoto formátu a pak opět zpět. O to se starají funkce prepare_word, separate_punctuation a apply_previous_formatting.
Kód lze nalézt v souborech WideCharUtilities.h, WideCharUtilities.cpp a Diacritics.cpp

Generování variant

Algoritmus mimo jiné potřebuje proceduru pro získání všech možných (nejlépe smysluplných) diakritických variant daného slova bez diakritiky. Funkce get_variants rekurzivně volá proceduru get_word_variants, která naplní kontejner všemi možnými variantami slova.
Jelikož se varianty generují rekurzivně a nesmyslná slova nelze rovnou zahodit, jelikož bychom tak mohli zavrhnout větev rekurze, která by vedla ke správnému tvaru slova, dostaneme v nejhorším případě 3^n variant pro n písmenné slovo. To může značně ovlivnit čas běhu pro dlouhá slova či vstup neoddělený bílými znaky.
V tomto případě je na místě použít nějaký způsob heuristiky, který funkci včas zastaví, nicméně pro tento projekt jsem předpokládal typicky formátovaný český text, který tímto

problémem netrpí.
Kód lze nalézt v souborech WideCharUtilities.h a WideCharUtilities.cpp

Parser korpusu

Jelikož jsem zamýšlel použití libovolného textu k obohacování statistiky, bylo potřeba převést značkovaný korpus na běžný text a ten poté zpracovat.
Procedury parse_corpus a create_trigram_model převedou korpus na souvislý text a ze souvislého textu vytvoří statistiku, která se bude dále programem využívat.
Jelikož mají typické korpusy velikosti v řádech gigabytů, uvažoval jsem nad tím, jak parsování zrychlit. Jediná z myšlenek bylo rozdělení korpusu na rovnoměrné úseky a tyto úseky pak zpracovávat na sobě na vícero vláknech funkcí parse_corpus_in_range. Vícevláknový přístup do jednoho souboru ovšem nijak významně výkon nevylepšil.
Kód lze nalézt v souborech CorpusParser.h a CorpusParser.cpp

Předpřipravení dat

Kromě přípavy modelu je potřeba vytvořit i zbývající dva pomocné soubory. Smazání duplicit a slití slovíků řeší merge_dictionaries a generování souboru s offsety na základě předloženého modelu funkce generate_compressed_model.
Pro převod z jednoduchého modelu v textové a číselné podobě do binárního formátu je přítomná i funkce compress_model_to_4_b.
Kód lze nalézt v souborech DataPreparation.h a DataPreparation.cpp

Struktury pro uložení slov a jejich vyhledávání

Jelikož jsou slova ve většině algoritmu reprezentována celými čísly, bylo pro přehlednost nutno vytvořit struktury word, word_tuple a word_triplet, které v sobě drží jeden, dva či tři inty.
Všechny mají tři argumenty v konstrukturu, ačkoliv word a word_tuple tyto argumenty nevyužívají. Je tomu tak kvůli generickým funkcím v algoritmu, které vytvářejí instance jedné z těchto struktur a pro jednotnost potřebovaly stejnou signaturu konstruktorů.
Pro urychlení vyhledávání v externích souborech slouží obousměrná hashovací tabulka word_mapping a hashovací tabulka model. Jelikož implementaci generické unordered bimapy **(obr. 3)** komplikovala implementace unordered_map, která nedovolila si jako hodnotu uložit ukazatel na klíč v opačné unordered_map, bylo nutné tuto strukturu vytvořit jen pro int a std::wstring. Implementace se snaží ušetřit paměť tím, že std::wstring uloží pouze jednou a pak se na něj odkazuje ukazatelem **(obr. 4)**, nicméně tento způsob nedovoluje vytvořit immutable verzi celé struktury - pokud bychom chtěli po naplnění celou obousměrnou mapu překopírovat do immutable varianty, budou všichni ukazatelé odkazovat na prázdné místo v paměti.
Jsou tedy dvě jednoduchá řešení tohoto problému - ponechání mutable verze nebo ukládání prvků dvakrát. Pro správnost je více vyhovující druhá varianta.
Třída model slouží k nalezení offsetu na řádek, na kterém se nachází dané slovo v modelu.

```cpp
template<typename T1, typename T2> class unordered_bimap
{
    std::unordered_map<T1, T2*> forward_map_;
    std::unordered_map<T2, T1*> inverse_map_;

public:
    void insert(T1& type1, T2& type2)
    {
        auto t1_it = forward_map_.insert(std::pair<T1, T2*>(type1, nullptr)).first;
        auto t2_it = inverse_map_.insert(std::pair<T2, T1*>(type2, nullptr)).first;

        t1_it->second = &(t2_it->first);
        t2_it->second = &(t1_it->first);
    }

    .
    .
    .

}
```

Figure 3: Generická bimapa

```cpp
class word_mapping
{
    std::unordered_map<std::wstring, int> word_to_int_map_;
    std::unordered_map<int, const std::wstring*> int_to_word_map_;

public:
    word_mapping() = default;

   .
   .
   .

}
```

Figure 4: Aktuální implementace bimapy

To lze opět řešit správným formátováním souboru s offsety, v němž by se pak hledalo pomocí std::istream::seekg(2). V aktuálním řešení byla zvolena varianta uložení do hashovací tabulky, kde si pro každé slovo pamatujeme offset, na kterém se nachazí. Tabulka není příliš velká a jejím načtením neztratíme mnoho paměti.
Kód lze nalézt v souborech LookupStructures.h a LookupStructures.cpp

Kód algoritmu

Funkce a procedury plnící práci algoritmu se nachází z velké části v třídě text_processor. Její konstruktor naplní vyhledávací struktury zmíněné výše a nastaví flagy pro další chování dle uživatelského vstupu. Pak lze volat proceduru process_text, která ze vstupního streamu určí jméno výstupního souboru a do něj vypisuje výstup algoritmu. Před výpisem probíhá zpracování prvního načteného slova ze vstupního streamu, poté se načte i slovo druhé a třetí slova se čtou bez bílých znaků až do konce vstupního streamu, přičemž první a druhá se taktéž postupně posouvají (idea z **obr. 1**). Po každém načtení třetího slova se od něj oddělí případná interpunkce, která se poté přenese jako další samostatné třetí slovo, a na všech třech načtených slovech proběhne iterace. Iterace pouze asynchronně spustí pomocnou funkci fill_result_word a zamění první slovo s druhým a druhé s třetím, čímž posune slova vpřed.
Jednotlivá asynchronní volání poté zbaví slova speciálních či formátovacích znaků včetně převedení na malá písmena a volá funkci most_common_triplet, která vrací výslednou nejvíce pravděpodobnou variantu prostředního slova. Do té se poté doplní veškeré formátování, kterého bylo slovo na začátku zbaveno, a vloží ho do společného kontejneru výsledných slov.

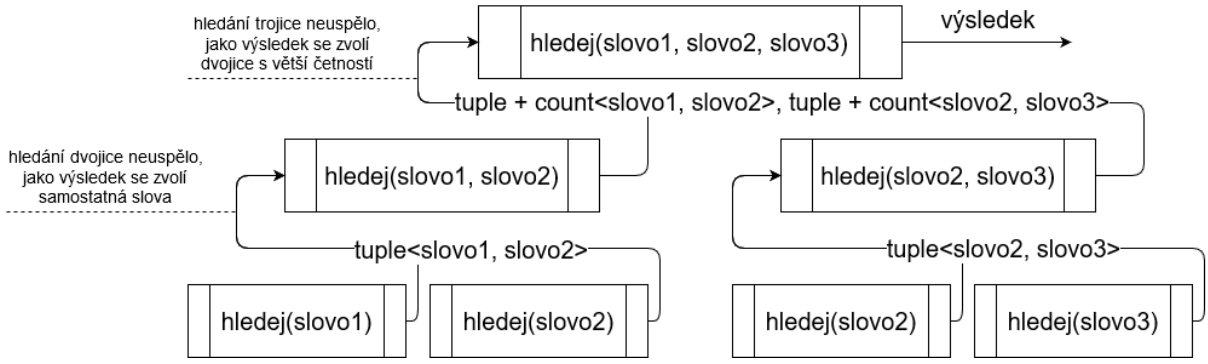

Figure 5: Předávání výsledku při hledání

Hledání nejpravděpodobnější varianty probíhá kaskádově. Nejprve jsou předána tři slova. U každého z nich se vygenerují všechny možné varianty a pro každou kombinaci těchto variant (pořadí slov zůstává stejné) se čtením modelu od prvního řádku, na němž začínají trojice s druhým slovem uprostřed, hledá řádek, na kterém všechna tři slova odpovídají dané variantě, tedy zda tři 4B hodnoty odpovídají celočíselné reprezentaci hledaných slov. Pokud se takový řádek najde, přečte se z něj četnost trojice a ta se uloží do mapy vyhovujících variant. Pokud se žádná trojice z variant nenajde, předají se k hledání dvě slova, <první a druhé> a <druhé

a třetí>. Hledá se naprosto stejným způsobem, jen se iteruje přes varianty dvou slov na místo tří a při čtení řádku z modelu se hodnota chybějícího slova ignoruje. Pokud se v modelu vrací dvojice, opět se vloží společně s četností do mapy variant. Do vyšší vrstvy kaskády se předá nejpravděpodobnější dvojice s jejím počtem, aby bylo možné porovnat počty mezi oběma dvojicemi v trojici **(obr. 5)** a vrátit variantu slova z té více pravděpodobné. Pokud nebyla nalezena ani dvojice, hledá se stejným způsobem samostatné slovo. To se opět vrací i s četností do vyšší vrstvy (vrstva dvojic), která výsledek předá výše.
Po tom, co hlavní vlákno rozdá práci všem volným, tedy poprvé přečte vstupní stream, čte jej ještě jednou, tentokráte uloží bílé znaky, které dříve přeskočilo, do kontejneru s formáty. Při vypisování se zpracovávají dvojice slov a formátů, přičemž formáty slouží k oddělení jednotlivých slov tak, jak tomu bylo ve vstupním dokumentu.
Kód lze nalézt v souboru Diacritics.cpp

Instrumentace a profiling

Pro snadnější optimalizaci celé aplikace byl nasazen jednoduchý instrumentační podprogram, který u funkcí označených makrem PROFILE_FUNCTION() zaznamenává čas strávený jejich vykonávaním pomocí funkcí knihovny std::chrono. Výsledek poté ve formátu json ukládá na disk do souboru result.json. Soubor lze otevřít v nástroji Chromium Trace, který běh programu vizuálně znázorní.
Kód lze nalézt v souboru Intrumentation.h

Alternativní přístupy v programování

Způsobů, jak data rozložit do externích souborů a do paměti je mnoho. Jelikož jsem k projektu přistupovat spíše jako k posloupnosti experimentů, jejichž výsledek v přesnosti a výkonu byl směrodatný k tomu, kterým směrem se vydat dále, zvážil jsem jen zlomek všech možných návrhů.
První alternativou je zachycení vstupu jako posloupnosti bytů, nikoliv wchar_t. To by přineslo výhody jako rozpoznání kódování a přizpůsobení se jemu, či přenositelnost napříč všemi lokalizacemi. Tento přístup by ovšem vyžadoval znovu vynalézt kolo, tedy opět implementovat již přítomné (a občas i fungující) API.
Veškeré IO operace lze namísto operátoru >> a std::getline(1) řešit pomocí memory mappingu či bufferů, což by také pomohlo s výkonem. Výstup si aktuální verze drží v paměti a po skončení algoritmu jej vypíše najednou, alternativní řešení by jej vypisovalo v malých dávkách a předešlo tak zbytečné spotřebě paměti.
Uložení slovníku do paměti lze udělat efektivněji, pokud by programátor přišel s jiným návrhem, jak reprezentovat slova, než je stávající mapování na celá čísla. Celá tabulka s offsety taktéž není nutná, pokud by se v modelu nasadilo vyhledávání pomocí binárního půlení.
Při doplňování diakritiky se stejná část modelu pro každé slovo v jedné iteraci může procházet až 7x (1x pro trojici, 1x pro obě dvojice a v každé dvojici 1x pro obě slova). Pokud bychom se chtěli ochránit před nejhorším případem, bylo by dobré buďto úsek přečíst jen jednou

a riskovat, že děláme zbytečné operace navíc (nalezneme trojici, ale uděláme přitom navíc porovnávání a přiřazování do map pro dvojice a jednotlivá slova), či úsek namapovat do paměti pro rychlejší přístup.
Zpracování formátu slov by jistě šlo udělat chytřeji, například přepínáním n-tice bitů v bitfieldu, kde každá n-tice bitů určuje formát písmene slova (velké písmeno, uvozovky, čárka, tečka, ...).

Reprezentace vstupních dat a jejich příprava

Vstupní data se programu předloží na základě přepínačů (viz. níže) jako soubor. Nejmenší předpokládaná délka vstupu jsou 3 slova. Slova jsou skupina znaků oddělená bílými znaky.

- Soubor: Uživatel při spuštění programu předá cestu k textovému souboru uloženém v kódování UTF-8 jako poslední parametr. Všechen obsah souboru bude interpretován tak, jako by byl v UTF-8 zakódován.

Od programu lze vynutit následující chování.

- Silence: přepínačem '-s' uživatel potlačí případné varování, že předložený vstup je cizojazyčný. Vstupní data může uživatel definovat buďto jako soubor předáním jeho cesty jako posledního parametru.
- Conflict: přepínačem '-c' bude při doplňování diakritiky uživateli poskytnut výběr mezi variantami těch slov, jejichž pravděpodobnost výskytu si je blízká. Na chybový výstup se kromě těchto variant zobrazí i obě sousední slova. Jednotlivé varianty se zobrazí jako číslovaný seznam. Uživatel poté zvolí libovolnou variantu zapsáním jejího čísla na standardní vstup. Tato funkcionalita je podmíněná makrem a je by default vypnutá. Po zprovoznění konzolových vstupů a výstupů bude fungovat tak, jak byla zamýšlena. Vstupní data může uživatel definovat buďto jako soubor předáním jeho cesty jako posledního parametru.
- Install: přepínačem '-i' bude spuštěna dekomprese pomocných souborů. Od uživatele se nečeká žádný vstup.
- Compress: přepínačem '-hc' uživatel spustí kompresi souboru, jehož cestu specifikuje jako poslední parametr.
- Decompress: přepínačem '-hd' uživatel spustí dekompresi souboru, jehož cestu specifikuje jako poslední parametr.
- Help: přepínač '–help' slouží k zobrazení nápovědy. Od uživatele se neočekává žádný vstup.

Velikost vstupních dat může ovlivnit chování programu. Jelikož spotřeba paměti v jisté míře závisí i na velikosti vstupu, je možné, že programu při zpracování dojde paměť. Pro příliš velké soubory (v rámci gigabytů) není na většině systémů zaručena správná funkčnost programu.

Reprezentace výstupních dat a jejich interpretace

Po úspěšném zpracování vstupu vypíše program na výstupní stream všechna zpracovaná data z paměti najednou. Ta budou uložena v kódování UTF-8 do stejnojmenného souboru s příponou .out. Navíc platí následující.

- Compress: po úspěšném běhu programu s přepínačem '-hc' bude komprimovaný soubor uložen do stejného adresáře jako vstupní soubor a ponese stejné jméno s přidanou koncovkou .hzip.
- Decompress: po úspěšném běhu programu s přepínačem '-hd' bude dekomprimovaný soubor uložen do stejného adresáře jako vstupní soubor a ponese stejné jméno s přidanou koncovkou .out.

Průběh prací

Práce začala snahou seznámit se se stylem, kterým C++ pracuje s non-ASCII znaky. Už v této fázi bylo potřeba udělat nenávratné rozhodnutí a tím bylo zvolení širokých řetězců a práce jen s UTF-8. V rámci této fáze vznikl podprogram pro odstraňování diakritiky, který poté pomáhal s přípravou dat při testování přesnosti na skutečných textech.
Teprve až po té jsem přemýšlel nad tím, jakým způsobem problém doplňování diakritiky řešit. Abych se více seznámil s texty, na základě kterých budu vytvářet statistiku, rozhodl jsem se vytvořit funkce pro parsování korpusu. To mi umožnilo se podívat, jak velká tato data skutečně jsou a jak těžké bude udržovat je v paměti. Na základě této zkušenosti jsem napsal funkce, které statistiku setřízeně uložily do externího souboru a protože i ten byl příliš velký, vytvořil jsem způsob, jak slova ukládat efektivněji a napsal pro ně příslušné funkce a paměťové reprezentace.
Při vymýšlení algoritmu mě mnoho dobrých myšlenek, které jsem zmínil v sekcích 'Diskuse řešení' a 'Alternativní přístupy' napadlo až příliš pozdě, proto nejsou žádné z nich nasazeny. Jelikož jsem i přesto viděl skvělé výsledky v prvotních testech (cca 98% přenost na novinových článcích), již jsem se implementací těchto vychytávek příliš nezaobíral.
Přesto že přesnost algoritmu splnila má očekávání, celková paměťová a časová náročnost programu mě nutila neustále měnit způsoby, kterými jsou data uložená v paměti. Mnoha změn, které ušetřily spotřebovanou paměť, poté musela být navrácena z důvodu implementace multithreadingu.
V poslední řadě bylo na místě zařídit, aby distribuce programu nebyla příliš náročná. Jelikož stahování 1GB externích souborů je pro tento druh projektu nestandardní, pokusil jsem se zátěž trochu snížit implementací Huffmanova komprimačního algoritmu. Protože existují

mnohem efektivnější metody komprese, například knohovna 'Zlib', rozhodl jsem se použít právě ji.

Co nebylo doděláno

Pro snížení velikost externích souborů o polovinu byla snaha držet si v modelu jen slova, která obsahují diakritiku, a pomocí nějaké heuristiky hádat, kdy je vhodnější doplnit nějakou variantu slova s diakritikou či bez ní. Tento přístup nebyl dokončen, jelikož jsem žádnou dostatečně přesnou heuristiku nevymyslel.
Dále bylo zamýšleno poskytnou funkcionalitu k parsování korpusu a přípravě vlastního modelu z daného zdroje také uživatelům. Jelikož jsou stávající funkce a procedury pro tuto práci velmi paměťově i časově neefektivní, rozhodl jsem se tento krok nepodniknout. Vylepšení těchto funkcí a jejich poskytnutí uživateli je ovšem něco, co dává rozum dodělat.
Původně měl kaskádový styl doplňování zahrnovat také na základě neuronové sítě, která by měla pro každé písmeno bez diakritiky perceptron. To by zajistilo přesnost na všech typech textu. Při použití pouze této sítě by program bylo možné používat s mnohem menšími externími soubory. Implementaci bránila moje neexistující znalost strojového učení, na vyžádání je ovšem možno program do tohoto stavu předělat (pokud by na to bylo poskytnuto rozumné časové okno).
V poslední řadě nebyly implementovány žádné pokročilejší metody pro výběr nejvhodnější varianty, jako například průniky trojic.
Dřívějšímu odevzdaní bránila především implementace standardního vstupu a výstupu, která na Windows nefunguje, jak by měla. Přestože jsou znaky vypisovány se správnými kódy, neumí je konzole zobrazit v lidsky čitelné podobě, většinou proto zobrazí dva znaky na místo jednoho. Oprava by znamenala vázanost kódu na danou platformu, proto jsem projekt nechal pracovat pouze se soubory.