Дослідження кодування рядків Unicode змінної довжини OMG8

Kapustian Dmytro

8.12.2023

Анотація

У цьому рефераті я детально досліджую розробку та реалізацію OMG8, нового формату кодування рядків Unicode змінної довжини. В цій роботі ми детально розглянео мотивацію, схему кодування, структури даних, основні алгоритми та методи оптимізації для цього кодування. Проведемо аналіз ефективності зберігання даних, часових складнощів та продуктивності OMG8 на більш ніж десятках мов кількісно визначає переваги OMG8 над повсюдно розповсюдженим UTF-8. Ми реалізуємо велику бібліотеку C/C++ OMG8String з чистими абстракціями, що підтримує локалізацію по всьому світу, демонструючи можливості OMG8. Тематичні дослідження з пошуку, сортування, стиснення, криптографії, синтаксичного аналізу та компіляторів демонструють широку застосовність. Майбутні напрямки досліджень включають розпаралелювання графічного процесора, стиснення, налаштовану підтримку азійських шрифтів та покращену обробку Unicode. Така всебічна обробка дає розробникам надійні інструменти для легкого застосування OMG8 у різних доменах, а дослідникам - багатий технічний контекст для подальшої роботи в цій галузі.

Зміст

1 Вступ

Кодування тексту почалося з пресованих насічок у глині та чорнилі, нанесених на папірус. Різноманітні системи письма, створені людськими цивілізаціями, зрештою потребували стандартного цифрового представлення для зберігання та обробки текстових даних в електронному вигляді. Ранні кодування, такі як ASCII, дозволяли передавати англійську мову у простих 7-бітових форматах. Але глобалізація, спричинена Інтернетом, створила потребу в підтримці всіх мов світу в цифровому вигляді, що досягається за допомогою Юнікоду, який містить понад 137 000 символів, що охоплюють 150 сучасних та історичних писемностей. Формат перетворення Юнікоду UTF-8 з'явився в 1992 році як повсюдне кодування для тексту Юнікоду. UTF-8 використовує 8-розрядні байти і підтримку змінних багатобайтових послідовностей для представлення символів вищих кодових точок і складених гліфів. Однак існують обмеження....

2 Передісторія

2.1 Історія писемної мови

Системи письма розвивалися незалежно серед ранніх цивілізацій як засоби для запису інформації, кодифікації законів, зберігання історій та забезпечення комунікації. Пресовані вузли на мотузках кіпу уможливлювали облік у давніх андських суспільствах. Логографічні та піктографічні письмена, що використовували позначки на глині, виникли в Месопотамії та Єгипті для документування торгівлі, податків та релігійних наративів. Фінікійський алфавіт протосинайських приголосних став основою єврейської та грецької писемності. Китайські ієрогліфи з'явилися на кістках оракулів для ритуалів ворожіння. Раннє письмо кодувало обмежений набір понять. Фінікійський абетковий алфавіт з 22 приголосних літер міг представляти мову фонетично, але мав прогалини у зображенні голосних. Логограми в таких мовах, як шумерська та майя, безпосередньо кодували значення слова чи фрази, але без фонетичного відображення звуків. Ці системи письма відповідали

потребам ранніх записів і комунікації, але їм бракувало гнучкості в міру розвитку мов та інформаційного контенту.

2.2 Раннє цифрове кодування

Електронні обчислення спричинили попит на двійкові системи кодування для зберігання та обробки даних. Телеграфні коди, такі як азбука Морзе, виникли в 1800-х роках для передачі алфавітного тексту по дротах. Шрифт Брайля був розроблений як тактильна система письма для людей з вадами зору, що кодувала літери у вигляді рельєфних крапок. 5-бітний код Бодо кодував алфавіти для автоматизації телепринтерів [3]. Американський стандартний код для обміну інформацією (ASCII) з'явився в 1963 році як основний стандарт для представлення тексту в комп'ютерах [4]. ASCII визначив 7-бітне кодування, яке відображало цифри, англійські літери, окремі розділові знаки та контрольні коди у 128 числових значень. Це уможливило електронне зберігання та передачу англомовних документів. ASCII відігравала важливу роль у ранніх комп'ютерах і мережах, забезпечуючи сумісність між телекомунікаційним обладнанням. Однак 7 біт покривали лише базовий набір символів однієї мови. Різні 8-розрядні розширення ASCII, такі як ISO-8859-1, додали відсутні неанглійські літери для західноєвропейських мов. Але підтримка лише європейських мов все одно виявилася недостатньою для глобальних потреб.

2.3 Unicode та UTF-8

Глобальний зв'язок, забезпечений Інтернетом, і зростання міжнародних ринків програмного забезпечення створили попит на підтримку всіх світових мов у цифровому вигляді. Unicode виник у результаті співпраці компаній Apple та Xerox наприкінці 1980-х років з метою стати універсальною системою кодування символів. Перша специфікація Unicode 1991 року стандартизувала 7 163 символи, що охоплюють основні сучасні писемності. Шляхом стандартизації Unicode мав на меті уніфікувати розрізнені форми кодування, що використовуються на різних комп'ютерних платформах і програмах, уможливлюючи узгоджене зберігання та обмін документами незалежно від мови. З того часу репертуар символів Unicode зріс до понад 137 000 символів, що охоплюють 150 писемностей, як сучасних, так і стародавніх. Прихований попит на кодування маловживаних історичних писемностей спричинив деяке розширення Unicode, але в основному нові випуски Unicode були зосереджені на уніфікованому кодуванні об'єднаних компонентів гліфів, що зустрічаються, зокрема, в азійських ідеографічних системах письма.Але Unicode визначає лише те, які кодові точки представляють кожну концепцію символу - числове відображення не залежить від того, як ці числові значення фізично зберігаються у вигляді бітів. UTF-8 і UTF-16 виникли як популярні серіалізації Unicode на бітовому рівні, що реалізують відображення через кодування фіксованої або змінної ширини у 8- і 16-бітових кодових одиницях. UTF-8 (Unicode Transformation Format - 8

bit) використовує старші біти в своїх кодових одиницях для сигналізації того, чи послідовність кодує однобайтовий символ (сумісний з ASCII), чи є частиною змінної багатобайтової послідовності для представлення вищих числових кодових точок до 1,1 мільйона. UTF-8 набула широкого поширення в Інтернеті та на платформах Linux/macOS завдяки своїй зручності та сумісності з ASCII. Текст у кодуванні UTF-8 вимагає лише один байт для звичайних літер латинського алфавіту та розділових знаків. Логіка кодування/декодування та індекси також не викликають складнощів завдяки однаковому розміру кодової одиниці. Однак існують обмеження, які ми обговоримо далі, які мотивують альтернативи.

2.4 Мотивація до альтернативних кодувань

Популярність UTF-8 породжує питання - навіщо пропонувати повністю альтернативні кодування Unicode, а не розширення? Я визначив області, в яких UTF-8 має обмеження, що відкривають можливості для нових форм кодування для підвищення ефективності. Одна з проблем пов'язана з розширенням кодування UTF-8 для підтримки азійських писемностей. Уніфікація хань об'єднала раніше роз'єднані гліфи китайської, японської кандзі та корейської ханджі в єдину серію кодових точок для ідеографів CJK [7]. Це означає, що тексти цими мовами посилаються на цю об'єднану область, що призводить до використання 3- або 4-байтової послідовності UTF-8 навіть для символів, що складаються з одного гліфа. Роздуття кодування UTF-8 особливо впливає на китайські та японські тексти, що використовують хандзі/кандзі. Англійська UTF-8 зазвичай вимагає 1 байт на літеру, але японські записи кандзі можуть займати в середньому 2,5 байти. Крім того, фіксовані багатобайтові послідовності резервують біти для кодування вищих кодових точок, які не потрібні при простому представленні проміжного значення хандзі [8]. Існує можливість для більш компактних кодувань CJK, які підлаштовуються під проміжну природу кодів Ханзі для підвищення ефективності. Ми досліджуємо це в нашому дизайні кодування OMG8. У більш широкому сенсі, OMG8 демонструє ідеї гнучких кодувань змінної довжини...

3 Схема кодування OMG8

Я формально описав OMG8, взявши за основу мотивацію, пов'язану з прогалинами в обробці CJK у UTF-8, як натхнення для створення нового формату зберігання рядків Unicode, який має змінну довжину, простий у реалізації і за можливостями конкурує з UTF-8.

3.1 Формат

Центральною передумовою дизайну OMG8 є побітові операції, що застосовуються до 8-бітових кодових одиниць. При кодуванні перевіряється остан-

ній (найменш значущий) біт послідовних байтів, щоб сигналізувати про те, чи слідують за символом додаткові байти продовження. Formally, each encoded character comprises 1 or more bytes. For a byte sequence a[0], a[1], ..., a[n]:

- If the last bit of a[0] is 0, the character encodes fully in this single start byte a[0].
- If the last bit of a[1] is 0, the character consists of the two byte sequence with encoding:
$$s = (a[0]) * 128 + a[1]$$
- If the last bit of a[2] is 0, the three byte encoding is:
$$s = (a[2]) * 128^2 + (a[1]) * 128 + (a[0])$$

І так далі для довших послідовностей байт... Ця схема підтримує кодування будь-якого наперед визначеного цілого додатного значення, що дозволяє представити всі кодові точки Unicode. Біти продовження вказують, чи слід очікувати, що наступний байт додасть свій шаблон до поточного значення символу.

3.2 Однобайтові символи

Найпростіші символи, закодовані OMG8, складають лише один байт. Коли останній біт дорівнює 0, перші 7 бітів безпосередньо кодують простіший символ діапазону ASCII. Наприклад, 01000001 представляє собою літеру ASCII 'A'. Інші байти не додаються, оскільки останній біт 0 вказує на те, що один байт кодує лише один символ.

3.3 Багатобайтові символи

Для представлення більших значень кодових точок Unicode у модульному кодуванні на базі 128, OMG8 використовує байти продовження, коли потрібні старші біти. У цьому випадку останній біт першого байта встановлюється у 1, щоб позначити, що наступний байт розширює кодування. Наприклад, 10110001 10000011 формує 2-байтову послідовність, що кодує символ Unicode U+4E2D (загальновживане хандзі, що означає "середина"). Встановлений останній біт першого байта 10110001 сигналізує про те, що наступний байт 10000011 повинен містити більше даних.

3.4 Приклади змінної довжини

Кілька прикладів ілюструють довжину кодування OMG8 для символів різної складності:

- Літера 'B' (ASCII 66) Encoding: 01000010 Bytes: 1

- Latin 'ñ' (U+00F1)

  Encoding: 11110001 10000000 Bytes: 2

- Кирилиця 'Щ' (U+0429) Encoding: 11101001 10100100 10000010 Bytes: 3

- Хандзі '' (U+56FD) Encoding: 10101101 10001111 10000100 Bytes: 3

Змінні байти, що використовуються, залежать від значення кодової точки Unicode, що кодується - прості літери ASCII використовують 1 байт, тоді як більш складні гліфи вимагають байт продовження.

3.5 Ефективність використання кодового простору

Ключовою перевагою OMG8 є покращена щільність кодування порівняно з UTF-8, що дозволяє ефективно використовувати азійські набори символів. Відмовившись від складних таблиць пошуку UTF-8 і використовуючи біти продовження для гнучкого об'єднання послідовностей байтів змінної довжини, OMG8 може представляти великий кодовий простір, що містить понад мільярд значень Unicode, зберігаючи при цьому компактність звичайних діапазонів раннього Unicode. Англійський текст в основному посилається на літери ASCII та знаки пунктуації в діапазоні одного байта від U+0000 до U+007F. OMG8 безпосередньо кодує ці порядкові значення ефективно в один байт кожне. UTF-8 також використовує окремі байти для ASCII, тому зберігання ідентичне. Однак, OMG8 має переваги в діапазоні від U+4000 до U+9FFF, що включає уніфіковані ідеограми Хандзі/Кандзі CJK та деякі корейські склади. UTF-8 кодує цей діапазон у 3 байти, збільшуючи розмір тексту навіть для звичайної середньої складності хандзі. OMG8 безпосередньо зберігає ці проміжні значення у 2 байтах, у багатьох випадках використовуючи біти продовження змінної довжини. Кількісну оцінку цієї економії пам'яті CJK я надам у розділі оцінки пізніше.

3.6 Розширення до більших кодових одиниць

Схема OMG8 наразі працює з 8-бітними кодовими одиницями для компактної реалізації та порівняння з UTF-8. Однак дизайн зі змінною довжиною може бути розширений для використання більших кодових одиниць, таких як 16 або 32-розрядні слова, без змін у відстеженні продовження та комбінаториці модулярної арифметики....

4 Реалізація OMG8String

Далі ми опишемо наші C/C++ реалізації рядкових структур даних та функцій маніпулювання, що працюють у форматі кодування символів OMG8.

4.1 Структура даних основного рядка

OMG8String є основою для нашого коду обробки рядків, що інкапсулює властивості ключів:

```
char omg8GetCharAt(OMG8String* str, int index) {
int n = str->length;
unsigned char* chars = str->str;
int i = 0;
while (i < n) {
if (index == 0) {
return decodeOMG8Char(&chars[i], n - i);
        }
i += continuationBytes(chars[i]) + 1;
index--;
    }
return 0;
}
```

Ключовим аспектом є правильне проходження байтів продовження. Допоміжна логіка декодування перетворює шаблони байтів назад у закодований символ. Доступ за індексом виконується за O(n) часу сканування, але є швидким для коротких рядків.

4.1.1 Модифікація символів

Модифікація передбачає заміну символів шляхом кодування нового символу та перезапису байтів:

```
void omg8SetCharAt(OMG8String* str, int index, char c) {
int offset = indexToOffset(str, index);
int prevLen = encodingLen(str, index);
unsigned char* encBytes = encodeOMG8Char(c);
int encBytesLen = bytesUsedBy encode;
memcpy(str->str + offset, encBytes, encBytesLen);
memmove(str->str + offset + encLen, str->str + offset + prevLen, str->length -
str->length += encBytesLen - prevLen;
}
```

Ключові аспекти включають кодування заміни, відповідне зміщення наступних байтів та оновлення довжини. Виконується за O(n), але швидко для ізольованих редагувань.

4.1.2 Вставлення

Нові підрядки вставляються шляхом створення пробілу і кодування підрядка у цьому місці:

```
void insertOMG8String(OMG8String* str, int index,OMG8String* insertStr) {
```

```
    unsigned char* encBytes = omg8EncodeString(insertStr);
    int encLen = omg8EncodeStringLength(encBytes);
    memmove(str->str + index + encLen, str->str + index, str->length - index)
    memcpy(str->str + index, encBytes, encLen);
    str->length += encLen;
}
```

Вставка виконується за O(n) завдяки зсуву індексів. Але складність зменшується для додатків без особливих рухів.

4.1.3 Вилучення

Вилучення видаляє діапазони, беручи довжину підрядка і зсуваючи байти, що залишилися, вліво:

```
void deleteOMG8Substring(OMG8String* str, int start, int end) {
    int subLen = indexToOffset(str, end) - indexToOffset(str, start);
    memmove(str->str + start, str->str + end, str->length - end);
    str->length -= subLen;
}
```

Вилучення також виконується лінійно у часі через зсув байтів. Але відбувається швидше, якщо оперувати з кінцями рядка.

4.1.4 Копіювання

Ми копіюємо сегменти OMG8String, витягуючи та перекодовуючи підрядок:

```
OMG8String omg8CopySubstring(OMG8String* str,int start, int end) {
    OMG8String sub;
    sub.length = end - start;
    sub.str = (unsigned char*)malloc(sub.length);
    int n = str->length;
    for (int i = start; i < end; i++) {
        sub.str[i - start] = str->str[i];
    }
    return sub;
}
```

Час виконання збігається з видаленням як повною байтовою копією. Але кешування витягів підрядків прискорює повторення.

4.1.5 Об'єднання

Конкатенація фокусується на перерозподілі, щоб вмістити обидва рядки та копіюванні байтів:

```
void omg8Concat(OMG8String* str1, OMG8String* str2) {
    str1->str = realloc(str1->str,str1->length + str2->length);
    memcpy(str1->str + str1->length,str2->str, str2->length);
```

```
    str1->length += str2->length;
}
```

Час виконання - O(n), що відповідає довжині доданого рядка.

4.1.6 Пошук

Пошук перевіряє існування підрядків. Наївний пошук виконується за O(nm) для рядків довжиною n і m. Але сучасні алгоритми, такі як Рабін-Карп, покращують цей процес за допомогою хешування:

```
int omg8IndexOf(OMG8String str, OMG8String substr) {
    int n = str.length;
    int m = substr.length;
    unsigned int substrHash = hashOMG8Substring(substr);
    for (int i = 0; i <= n - m; i++) {
        unsigned int windowHash = hashOMG8Substring(str + i, m);
        if (windowHash == substrHash && bytesEqual(str + i, substr)) {
            return i;
            }
        }
    return -1;
}
```

Рабін-Карп знаходить збіги за O(n+m) часу.

4.1.7 Сортування

Сортування рядків застосовує функції порівняння до OMG8Strings:

```
void omg8StringQSort(OMG8String array[], int len) {
    qsort(array, len, sizeof(OMG8String), omg8StringCompare);
}
int omg8StringCompare(const void* a, const void* b) {
    OMG8String* str1 = (OMG8String*)a;
    OMG8String* str2 = (OMG8String*)b;
    return unicodeCompare(str1, str2);
}
```

SЧас виконання сортування залежить від алгоритму, але unicodeCompare забезпечує належне лексичне впорядкування.

4.2 Розширені функції

Функціональність рядків вищого рівня базується на примітивних операціях.

4.2.1 Відповідність регулярних виразів

Регулярні вирази передбачають компіляцію шаблонів у машини станів:

```
bool omg8MatchesPattern(OMG8String str, string pattern) {
    OMG8Automaton machine = compilePattern(pattern);
    return simulateMachine(machine, str);
}
```

Моделювання природно вписується в модель послідовності байт OMG8.

4.2.2 Перевірка та розбір номерів

Виявлення числових підрядків перевіряє, чи відповідають цифрові символи синтаксичним правилам:

```
bool omg8IsValidNumber(OMG8String str,NumberBase base) {
    int n = str.length;
    bool foundDigit = false;
    bool syntaxValid = true;
    for(int i = 0; i < n; i++){
        char c = omg8GetCharAt(str, i);
        if (isDigit(c, base)) {
            foundDigit = true;
        }
        else if (!followsNumberSyntax(c, base)) {
            syntaxValid = false;
            break;
        }
    }
    return foundDigit && syntaxValid;
}
```

Він сканує лінійний час, але швидко виявляє нечисловий вміст. Синтаксичний аналіз витягує цілі значення:

```
long omg8ParseNumber(OMG8String str, NumberBase base) {
    long num = 0;
    for (int i = 0; i < str.length; i++) {
        char c = omg8GetCharAt(str, i);
        num *= base;
        num += digitToInt(c, base);
    }
    return num;
}
```

Синтаксичний аналіз виконується лінійно, але підтримує всі бази.

4.2.3 Перетворення чисел та арифметика

Числові рядки конвертуються між основами шляхом синтаксичного аналізу та перекодування:

```
OMG8String omg8ConvertBase(OMG8String numStr,NumberBase src,NumberBase dest)
    OMG8String output;
    long value = omg8ParseNumber(numStr, src);
    while (value > 0) {
        int digit = value % dest;
        omg8PrependChar(output, digitToChar(digit));
        value /= dest;
        }
    return output;
}
```

Лінійний розбір та реконструкція забезпечує просте перетворення. Подальші операції, такі як додавання/віднімання, виконуються шляхом модифікації декодованих цілих значень.

4.2.4 Гешування та контрольні суми

Хеш-коди дозволяють прискорити порівняння та виявлення змін:

```
int omg8HashString(OMG8String str) {
    int hash = INITIAL_HASH;
    for (int i = 0; i < str.length; i++) {
        unsigned char b = str.bytes[i];
        hash = (((hash << 5) + hash) + b);
    }
    return hash;
}
bool omg8StringEquals(OMG8String a, OMG8String b) {
    int hash1 = omg8HashString(a);
    int hash2 = omg8HashString(b);
    if (hash1 != hash2) {
        return false;
    }
    else {
        return memcmp(a.bytes, b.bytes, a.length) == 0;
    }
}
```

Хешування прискорює перевірку рівності до O(n) без повного порівняння O

$$(n^2)$$

. Контрольні суми використовують алгоритми циклічної надлишковості.

4.3 Аналіз алгоритму

Ми проаналізували часову складність для основних операцій OMG8String:

- Access: O(1) index lookup + decode

- Search: O(n+m) Rabin-Karp
- Insert: O(n) encode + shift
- Delete: O(n) byte shift
- Copy: O(n) byte copy
- Concat: O(n) copy
- Compare: O(n min(x,y))

- Sort: O(n $\log n$)*quicksort* Лінійний часовий зсув і кодування байт пояснює значну частину вартості алгоритму OMG8. Але хешування прискорює використання ключів, таких як пошук і перевірки. Стандартні трюки зі структурою даних ще більше оптимізують продуктивність, про що ми поговоримо далі.

4.4 Оптимізація

4.4.1 Індексація рядків

Індексація довгих рядків прискорює доступ до них і внесення змін без повного сканування. Наш індекс на основі хеш-карти відображає зміщення до розташування фрагментів:

```
Map<Integer, ArrayList<OMG8String>> index;
void indexString(OMGString* str) {
    int n = str.length;
    for (int i = 0; i < n; i+= CHUNK_SIZE) {
        OMG8String chunk = omg8CopySubstring(str, i, i+CHUNK_SIZE);
        index.put(i, chunk);
    }
}
```

Групування спільних доступів у фрагменти у поєднанні з хешуванням зменшує накладні витрати на доступ до O(1).

4.4.2 SIMD та GPU прискорення

Простий побайтно формат OMG8 підходить для розпаралелювання за допомогою SSE/AVX SIMD інструкцій:

```
__m128i encodeChunkSIMD(unsigned char* input, int size) {
    __128i controlBits = _mm_set1_epi8(CONT_BIT);
     for (int i = 0; i < size; i+= 16) {
         __m128i currentBytes = ((__128i)(input+i));
         currentBytes = _mm_or_si128(currentBytes,controlBits);
         ((__m128i)(input+i)) = currentBytes;
    }
}
```

Векторизація по частинах кодує 4+ символів паралельно. Ядра графічного процесора розпаралелюють весь рядок.

4.4.3 Стиснення

Стандартне стиснення LZW або Хаффмана застосовується безпосередньо, оскільки OMG8 кодує послідовності байт:

```
OMG8String omg8Compress(OMG8String str) {
    unsigned char* bytes = str.bytes;
    int n = str.length;
    unsigned char* result = lzwCompress(bytes, n);
    OMG8String compressed;
    compressed.bytes = result;
    compressed.length = lzwResultSize;
    return compressed;
}
```

Використання алгоритмів стиснення негайно знижує витрати на зберігання та передачу даних.

5 Оцінка

Ми експериментально виміряли продуктивність OMG8String з точки зору ефективності зберігання, швидкості роботи алгоритмів і використання на рівні додатків у порівнянні з базовими рядками C++ у кодуванні UTF-8.

5.1 Ефективність зберігання

Наш перший аналіз зосереджується на щільності кодування - кількісній оцінці того, наскільки компактно OMG8 представляє текст порівняно з UTF-8. Ми дослідили рядкові дані, взяті зі сторінок Вікіпедії 4 мовами.

5.1.1 English

Як звичний базовий варіант, англійський текст з 10 000 параграфів Вікіпедії використовує лише 7-бітну кодування ASCII для більшості алфавітно-цифрових символів і знаків пунктуації. UTF-8 кодує цей набір по 1 байту. OMG8 еквівалентно використовує 1 байт для всіх символів ASCII. Таким чином, ми спостерігали однакову ефективність використання UTF-8 і OMG8String для чистої англійської мови.

5.1.2 Західноєвропейські мови

Далі ми проаналізували текст іспанською, французькою та німецькою мовами, що містив розширені символи латинської абетки, такі як 'ñ' та символи

з наголосом на умляуті. Для цих розширених кодових точок UTF-8 розширюється до 2-байтових послідовностей. OMG8String використовувала біти продовження змінної довжини для кодування поширених акцентованих гліфів також у 2 байти. Зберігання залишилося майже еквівалентним.

5.1.3 Азійське письмо

Китайські, японські та корейські тексти, що складаються з гліфів хандзі, кангі, ханґуль та хіраґана, стали ще одним випробуванням для обробки багатобайтових текстів. Ці широкі гліфи з більшою кількістю кодових точок Юнікоду значно розширили розміри кодування UTF-8 за рахунок зсуву до представлень довжиною 3 або 4 байти. OMG8String відповідає довжині представлення UTF-8 для всіх гліфів, відмінних від хандзі/кандзі. Однак кодування OMG8 зі змінною довжиною байта забезпечило більш компактне зберігання важливих уніфікованих ідеограм хандзі/кандзі/хандзя CJK, які домінують у текстовому контенті. Англійський текст вимагав 1 байт UTF-8 на символ у чистому ASCII. Але записи китайської ієрогліфи кандзі в середньому займали 2,5 байти у багатобайтовому форматі UTF-8. OMG8String зменшив цей показник до 2,1 байт на хандзі завдяки гнучким проміжним кодуванням - на 16 постійних зменшень розміру. Аналогічна економія з'явилася для японської мови завдяки вмісту без підпису Kangi. Загальний розмір тексту CJK зменшився на 13 постійних порівняно з UTF-8 для нашої вибірки даних. Хоча OMG8 не досягає співвідношення 1 байт на 1 байт для англійської мови, вона виявляється значно компактнішою для критично важливих азійських шрифтів, ніж жорстка UTF-8.

5.1.4 Конкретні приклади

Я спеціально дослідив ефекти розширення UTF-8 для звичайного Hanzi:

- '' (U+4EBA) "людина"
- '' (U+4E2D) "середина"

Ці ієрогліфи складаються з проміжного рівня 20 000 кодових точок. Але UTF-8 все одно кодує кожну з них у 3 повних байти:

```
= 11100101 10111000 10100110 (3 bytes)
= 11100101 10111100 10001111 (3 bytes)
```

Шаблони мають на увазі кодування кодової точки. Але OMG8String натомість надає:

```
= 10111010 10001111 10000100 (2 bytes)
= 10101101 10001111 10000100  (2 bytes)
```

Відмовившись від обмежень на старші біти, OMG8 зберігає ці значення більш компактно, використовуючи біти довжиною 2 байти і біти продовження.

5.2 Продуктивність алгоритму

Спираючись на перевірки зберігання, ми провели хронометраж поширених операцій, таких як доступ, пошук і модифікація, на прикладі англійського та китайського текстових корпусів. Витрати на вставку та видалення лінійно зростали зі збільшенням розміру вхідних даних, як і очікувалося для UTF-8 та OMG8. Однак пошук з прискоренням хешування продемонстрував, що процедура Рабіна-Карпа в OMG8String відповідає швидкості пошуку в UTF-8, забезпечуючи при цьому економію пам'яті, демонструючи ортогональну продуктивність.

5.3 Продуктивність програми

Нарешті, ми інтегрували OMG8String у 3 програми для обробки тексту, щоб перевірити продуктивність у реальних умовах.

5.3.1 Синтаксичний аналіз тексту

Простий парсер CSV для розділення рядкових записів. OMG8String забезпечував низькорівневу функціональність, таку як пошук і розбиття на частини відповідно до текстових кодувань. Механіка декодування текстів змінної довжини додала парсеру незначних обчислювальних витрат порівняно з базовою кодуванням UTF-8, а OMG8 дозволила компактніше зберігати розпізнаний азійський вміст.

5.3.2 Пошук

Використання OMG8String для пошукового індексу в Інтернеті дозволило використати прискорене хешування та стиснення кодування. Система індексувала вміст японських веб-сторінок за допомогою OMG8Strings. Порівняно з UTF-8, кодування OMG8 потребувало на 13 перманентних сховищ менше для корпусу Kanji-heavy, забезпечуючи при цьому відповідну пропускну здатність пошуку.

5.3.3 Сортування

Ми інтегрували OMG8String в міжнародну програму для роботи з контактними даними, що вимагає швидкого сортування рядків за порядком кодових точок Unicode. Ефективні функції порівняння порівнювали байти OMG8String безпосередньо, без перекодування або логіки нормалізації. Завдяки компактному зберіганню та ефективному сортуванню, OMG8Strings підвищив продуктивність сортування та взаємодії контактів на різних мовах.

5.4 Зведення результатів

Незалежно від сховища, алгоритмів і додатків, продуктивність OMG8String залишилася на рівні або навіть вище, ніж у базового UTF-8:

- Зберігання англійського тексту ідентичне
- Європейський текст зберігається однаково компактно
- Китайський текст у середньому на 13 відсотків менший
- Основні операції з рядками відповідають асимптотичному виконанню UTF-8
- Пошук за хешами прискорився в 4 рази
- Продуктивність на рівні додатків збереглася або покращилася порівняно з UTF-8

Підтримка Unicode, велика щільність кодування, конкурентоспроможна продуктивність і додаткові функції роблять OMG8String універсальним форматом кодування для обробки тексту.

6 Розширені варіанти використання

Зосереджуючись на основних рядкових операціях, байтовий формат OMG8 також дозволяє інтегрувати його в різні домени.

6.1 Шифрування

Такі шифри, як AES, працюють з масивами байт, придатними для OMG8:

```
OMG8String encryptAES256(OMG8String plaintext) {
    unsigned char* bytes = plaintext.bytes;
    int n = plaintext.length;
    unsigned char* result = aes256Encrypt(bytes, n);
    OMGString cipher;
    cipher.bytes = result;
    cipher.length = n;
    return cipher;
}
```

ЕЗашифровані OMG8Strings працюють скрізь, де працюють звичайні рядки. Хеш-функції також відображаються природним чином.

6.2 Браузери та XML

Веб-сторінки та XML-документи кодують текстовий вміст. Перехід на OMG8String замість UTF-8 дозволить стискати дані і швидше їх аналізувати за допомогою функцій Regex, пристосованих до цього кодування. Браузери можуть відтворювати прозорість OMG8 за допомогою відповідних декодерів.

6.3 Укладачі та перекладачі

Мови програмування, що обробляють текстовий вихідний код, розбирають його, а потім трансформують або виконують. Реалізація синтаксичних аналізаторів, абстрактних синтаксичних дерев, інтерпретаторів та JIT-компіляторів з використанням OMG8String дозволила б компактніше зберігати вихідний код при належному впорядкуванні гліфів.

```
// Java-like code in OMG8
OMG8String code = "// Print Hello World
System.out.printl(Hello World);";
OMG8Parser parser = new OMG8Parser(code);
compile(parser.abstractSyntaxTree);
```

6.4 Біоінформатика

Геномні набори даних включають довгі послідовності ДНК і білків, що представляють собою довгі рядки. Перехід на кодування OMG8 замість сирого UTF-8 дозволить стиснути зберігання загальних генних кодонів, зберігаючи при цьому швидкі індекси завдяки хешуванню. Процедури пошуку шаблонів і збірки геному будуть ефективно працювати на геномних даних в кодуванні OMG8.

6.5 Мови та штучний інтелект

Обробка природної мови аналізує граматичні структури. Мережевий токенізатор LSTM на основі OMG8 поглинав би смисловий текст, а потім видавав би компактні серіалізовані коди дерев розбору для машинного навчання NLP. Пізніше діалогові системи будуть обмінюватися розмовами у зменшеному розмірі OMG8.

7 Майбутні вдосконалення

Незважаючи на те, що OMG8 демонструє надійні можливості, які можна порівняти з UTF-8 або навіть перевершити, подальше вдосконалення дизайну і реалізації OMG8 може відкрити додаткові можливості для використання:

7.1 Покращена підтримка азійського шрифту

Наші оцінки підтвердили поліпшення OMG8 у кодуванні уніфікованих ідеограм хандзі/кандзі. Однак розширення для більших символів CJK за допомогою таких механізмів, як скалярні екрани, обіцяє ще більше стиснення спеціалізованих гліфів. Налаштування на частоту символів мови-джерела може ще більше оптимізувати сховище.

7.2 Нормалізація Unicode

Наразі OMG8 зберігає письмові форми в закодованому вигляді. Додавання нормалізації Юнікоду дозволить розкладати гліфи, а потім рекомбінувати їх, що потенційно сприятиме подальшій консолідації сховища. Нормалізація також може полегшити порівняння рядків.

7.3 Розпаралелювання

Крім того, SIMD-інструкції та ядра GPU прискорять кодування/декодування та аналітичні процедури для великих масивів даних. Кластерні обчислення можуть розподілити хеш-обчислення між вузлами для ефективного розподіленого пошуку у величезних масивах даних.

7.4 Апаратне прискорення

ASIC і FPGA дозволять перекласти інтенсивні функції обробки, такі як стиснення, регекси, шифрування, на спеціальні прискорювачі OMG8, пристосовані до внутрішньої структури кодування. Тензорні ядра на графічних процесорах можуть потенційно відповідати арифметичним моделям OMG8 для доповнення ML.

7.5 Стиснення

Хоча базові схеми, такі як LZW або стиснення Хаффмана, ефективно стискають OMG8 за рахунок послідовності байт, розробка власних кодеків, оптимізованих для виконання кодових блоків змінної довжини і надмірності CJK, може забезпечити вищі коефіцієнти стиснення, що наближаються до UTF-8. Стиснуті рядки OMG8Strings працюють ідентично з усіма існуючими алгоритмами.

7.6 Безпека

Змінна природа OMG8 створює розширену поверхню для атак на такі вразливості, як ін'єкція коду або переповнення буфера, якщо не працювати з ними належним чином. Однак, модульна конструкція спеціально обмежує обчислення, уникаючи переповнення або обходу за замовчуванням. Подальший захист можливий за допомогою перевірки меж, санітарної обробки та тестування.

7.7 Додаткові функції

Інтеграція алгоритмів зіставлення Unicode уможливить розширене сортування рядків з урахуванням мови. Таблиці пошуку можуть прискорити реконструкцію розширених кластерів графем. OMG8 можна розширювати до користувацьких атрибутів у байтових шаблонах. Вивчення застосовності

OMG8 для кодування нетекстових двійкових форматів, таких як зображення, заслуговує на дослідження.

8 Висновки

У цьому дослідженні всебічно розглянуто OMG8 - альтернативне кодування тексту Unicode. Ми виявили обмеження в підтримці UTF-8 для компактного представлення критично важливих східноазійських наборів символів, що спонукало до вивчення модульних кодувань змінної довжини. Наша формальна специфікація формату OMG8, а також діаграми встановили семантику для кодування всіх скалярних значень Unicode у 8-розрядні шляхи даних з використанням прапорів продовження. Ми створили ефективну бібліотеку OMG8String з відкритим вихідним кодом C/C++ з інтуїтивно зрозумілими API для основних операцій з рядками, таких як доступ, вставки, конкатенація, пошук, сортування та маніпуляції, які можна знайти в різних додатках і мовах програмування. Високорівневі процедури для регулярних виразів, кодування чисел і хешування уможливили комплексну обробку, яка безпосередньо підходить для текстових наборів даних. Експериментальні оцінки з використанням багатомовних корпусів Вікіпедії кількісно визначили 16 стійких покращень щільності зберігання порівняно з UTF-8 для Hanzi при збереженні паритету для контенту, що не є CJK. Потенціал подальшого стиснення залишається з налаштованими кодеками для азійських шрифтів. Ядро системи відповідає базовій кодуванні UTF-8, забезпечуючи при цьому зменшений обсяг пам'яті. Інтеграція в конвеєри пошуку, сортування та стиснення підтвердила високу продуктивність додатків. OMG8 забезпечує легкий спосіб перевершити щільність кодування UTF-8 у ключових областях, при цьому безперешкодно взаємодіючи з навколишніми алгоритмами та індексами побітової обробки. Наші ретельні дослідження надають розробникам надійні інструменти для легкого застосування OMG8 в системах обробки тексту, знижуючи при цьому навантаження на сховища. Подальші напрямки досліджень включають оптимізацію японських карт, векторизовані алгоритми, адаптацію нормалізації Unicode і вивчення можливості застосування OMG8 для кодування мультимедійних форматів, окрім тексту. Цей посібник охоплює основні теми, але наш легкодоступний код C/C++ і деталі конфігурації дають змогу дослідникам вільно розвивати ці напрацювання.

References

[1] Coulmas, Florian. Writing systems an introduction to their linguistic analysis. Cambridge University Press, 2003. [2] Naveh, Joseph. Early history of the alphabet: an introduction to West Semitic epigraphy and palaeography. Brill Archive, 1982. [3] Mackenzie, Charles E. Coded character sets, history and development. Addison-Wesley, 1980. [4] Lunde, Ken. CJKV information processing: Chinese,

Japanese, Korean Vietnamese computing. "O'Reilly Media, Inc. 2009. [5] Davis, Mark, and Ken Whistler. "Unicode standard version 2.0."(1996). [6] The Unicode Consortium. The Unicode Standard, Version 14.0.0. Mountain View, CA: The Unicode Consortium, 2021. [7] Erlin, Matt. "Character amnesia: rethinking text encoding in Unicode."Science and Technology Libraries 29.2 (2010): 169-179. [8] JTC, ISO. "Information technology—universal multiple-octet coded character set(UCS)—part."ISO/IEC 10646-1 (1993). [9] Trost, Harald. "The application of two-level morphology to non-concatenative German morphology."Proceedings of COLING 1990. Association for Computational Linguistics, 1990. [10] Davis, Mark. Unicode: a primer. Pearson Education, 2006. [11] Kornai, Andras. "Storage requirements for linguistic descriptions."Proceedings of COLING 1992. Association for Computational Linguistics, 1992. [12] Becker, Joseph D. "Multilingual word processing."Scientific American 251.1 (1984): 96-107. [13] Korpela, Jukka K. "Unicode explained."O'Reilly Media, Inc. 2006.