

Data Structure Final Project – Report

學號：111000104

姓名：李佳樺

Implementation

在這次的 final project 當中，我主要使用了 Trie 來作為處理查詢的工具，會選擇 Trie 的原因是因為他可以先花 $O(|S|\log N)$ 的時間把所有文本插入 Trie，會多一個 \log 是因為我在 Trie 的節點上記錄該字元來自哪些文本編號， N 是文本總數量， $|S|$ 是全部文本的總長度，預先處理完後每次查詢 (prefix, exact, suffix) 就可以只花 $O(|q|)$ 的時間確認，其中 $|q|$ 為查詢的字串長度。而 wildcard 的部分比較特殊，後面會提到。

Build Trie

為了一次可以查詢所有文本，所以我們的 Trie 結點多儲存了兩個 `std::set`，一個是 `path` 代表當前結點是來自哪些文本編號的集合，而另外一個 `tail` 代表當前結點是文本編號的最後一個字的集合。而 `mx_depth` 是後面優化會用到的，代表以當前結點為根的子樹的最大深度。`go[]` 代表當前結點的子結點陣列。而在 `insert` 時就只要在插入字串的每個字元對應到的節點的 `path` 都插入文本編號，到最後一個字元時在 `tail` 插入文本編號。

```
1 | inline void insert(Node *p, const string& s, const int &id) {  
18 |     int depth = 0;  
2 |     p->mx_depth = max(p->mx_depth, (int)s.size());  
3 |     for(char c : s) {  
4 |         p = next(p, c);  
5 |         depth++;  
6 |         p->path.insert(id);  
7 |         p->mx_depth = max(p->mx_depth, (int)s.size() - depth);  
8 |     }  
9 |     p->tail.insert(id);  
10 | }
```

圖 1：Insert function

```
7 | struct Node {  
8 |     Node *go[26] = {};  
9 |     int mx_depth = -inf;  
10 |     set<int> path, tail;  
11 | };
```

圖 2：Trie Node Struct

Prefix Search & Exact Search

而查詢 Prefix 就只要從根結點開始遍歷 Trie，如果能走完整個字串的話就回傳走到的節點的 `path`，exact 也是一樣的方式，只是變成回傳 `tail` 而已。

Suffix Search

Suffix 其實就是倒過來的 Prefix，所以我開了另外一棵 Suffix Trie 來儲存反過來的文本，要查詢時就把 Query 反過來後查詢 Suffix Trie 的 Prefix 就可以了。

Wildcard Search

因為 `*` 可以塞任意數量的字元，所以我們使用 dfs 的方式進行搜尋，跟前面查詢一樣，從根結點出發，如果當前查詢的字元是 `*` 的話，那對於下一個節點，你可以選擇要不要放到 `*`，如果不要的話就直接移動到 `*` 的下一個字元所在的節點，如果要的話就放進去然後往子結點 dfs，如果當前查詢的字元不是 `*` 的話就檢查有沒有對應字元的子結點就可以了。

Optimization

剪枝優化

在前面有講到每個節點有紀錄當前子樹的最大深度，在搜尋時，如果搜尋的字串長度 > 最大深度的話，我們就可以直接回傳空集合，因為就算搜到底也不會有 match，可以不用浪費時間往下搜。

平行化（使用 OpenMP）

在前面有說一棵 Trie 可以儲存所有文本，但我們其實可以平行化他，開 (NUM_THREADS) 數量的 Trie，平行建樹，在處理查詢時也可以平行查詢。

```
inline set<int> query(const string &s) {
    string tmp = query_parse(s);
    set<int> res(NUM_THREADS);
    if(s[0] == '*') {
        #pragma omp parallel for num_threads(NUM_THREADS)
        for(int i = 0; i < NUM_THREADS; ++i) {
            res[i] = query_exact(prefix_trie[i], tmp);
        }
    } else if(s[0] == '+') {
        reverse(all(tmp));
        #pragma omp parallel for num_threads(NUM_THREADS)
        for(int i = 0; i < NUM_THREADS; ++i) {
            res[i] = query_prefix(suffix_trie[i], tmp);
        }
    } else if(s[0] == '<') {
        #pragma omp parallel for num_threads(NUM_THREADS)
        for(int i = 0; i < NUM_THREADS; ++i) {
            res[i] = query_wild(prefix_trie[i], tmp);
        }
    } else {
        #pragma omp parallel for num_threads(NUM_THREADS)
        for(int i = 0; i < NUM_THREADS; ++i) {
            res[i] = query_prefix(prefix_trie[i], tmp);
        }
    }
    for(int i = 1; i < NUM_THREADS; ++i) for(auto &j : res[i]) res[0].insert(j);
    return res[0];
}
```

圖 3：parallel query part 1

```
inline void solve() {
    #pragma omp parallel for num_threads(NUM_THREADS)
    for(int i = 0; i < qry.size(); ++i) {
        vector<string> tmp_string = split(qry[i], " ");
        ans[i] = query(tmp_string[0]);
        for(int j = 1; j < tmp_string.size(); j += 2) {
            if(tmp_string[j][0] == '+') {
                And(ans[i], query(tmp_string[j + 1]));
            } else if(tmp_string[j][0] == '<') {
                Sub(ans[i], query(tmp_string[j + 1]));
            } else {
                Or(ans[i], query(tmp_string[j + 1]));
            }
        }
    }
}
```

圖 4：parallel query part 2

```
inline void build_trie() {
    #pragma omp parallel for num_threads(NUM_THREADS)
    for(int i = 0; i < NUM_THREADS; ++i) {
        prefix_trie[i] = new Node();
        suffix_trie[i] = new Node();
    }
    #pragma omp parallel for num_threads(NUM_THREADS)
    for(int i = 0; i < data_nums; ++i) {
        int thread_id = omp_get_thread_num();
        for(const auto &s : data_set[i]) {
            insert(prefix_trie[thread_id], s, i);
            rev_insert(suffix_trie[thread_id], s, i);
        }
    }
}
```

圖 5：parallel build trie

而經過多次測試後，發現把 NUM_THREADS 設為 6 效果最好，因為自己電腦的 CPU (i7 12700) 跟測試用 CPU (i7 12700k) 的核心數、線程數都相同，所以 NUM_THREADS 設一樣表現應該不會差太多。

多平台測試結果如下（一萬筆 dataset，queryfile 使用 query_more.txt）：

```
> make search
g++ -fopenmp -std=c++17 -O2 -o essay-search ./main.cpp
> time make query_my
./essay-search mydata query_more.txt output.txt
diff output.txt myoutput.txt

CPU    414%
user    3.643
system  0.471
total   0.992
```

圖 6：windows wsl (i7 12700 4.43Ghz, 32G ram)

```
readfile : 333754
build_trie : 3064334
read query : 126
solve : 206483
output answer : 66599
diff output.txt myoutput.txt

CPU    377%
user    3.289
system  0.450
total   0.991
```

圖 7：macbook air (m1, 8G ram)

```
Days           : 0
Hours          : 0
Minutes        : 0
Seconds        : 1
Milliseconds   : 673
Ticks          : 16737977
TotalDays      : 1.93726585648148E-05
TotalHours     : 0.000464943805555556
TotalMinutes   : 0.0278966283333333
TotalSeconds   : 1.6737977
TotalMilliseconds : 1673.7977
```

圖 8：windows powershell (規格同 wsl)

Challenges & Conclusion

優化到最後發現瓶頸在於 Trie 的建立上，我猜是 set 的常數太大，而且多了一個 log，所以後來有嘗試換成各種資料結構，像是 VEBTree、B-Tree 等，甚至是自己重新寫一個 Red Black Tree，但效果都不是很好，可能是因為自己在寫資料結構時都蠻常會出現常數太大的問題，所以表現不會比 std::set 好，甚至比較差。

在這次的 Final Project 當中，學到了許多優化技巧，也嘗試了各種資料結構，如果日後有專案需要用到這種字串工具就會更加得心應手，程式也會更有效率。

References

<https://zh.wikipedia.org/zh-tw/OpenMP>