資料結構期中專案暨字串比對結報

資訊工程學系 黃右萱(0416323)

一、算法介紹

本次作業採用 KMP 字串比對演算法，做法先自底向上以線性時間建立一個 prefix array，其中記錄每個前綴序列的次長前綴後綴，字串比對時，逐一比對字元，遇到字元不匹配則參考 prefix array，向前遞取次長前綴後綴長度回溯匹配進度，單論比對效率，KMP 演算法的時間複雜度為 O(n+m)，空間複雜度為 O(n+m)，若搭配適當的記憶體控管，空間複雜度可以優化至 O(m)。註：n、m 為代測字串與 pattern 字串長度。

二、測試資料

測試資料分為兩種規模，分別為數十字元以內小型測試資料，以及數百行的中型測試資料、每行一至二千字元。

小測資有範例測資，以及設計測資如下：

1. Pattern: ab, String: aaaaaaaaab
2. Pattern: abcabc, String: abcabcabc

中型測資行數規模大約在 1e2~1e3 之間。有隨機化測資：字元種類為 69，生成代碼如下圖(一)；以及極端測資：其生成代碼如下圖(二)。

圖(一)

```cpp
#include<bits/stdc++.h>

using namespace std;

string str="abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789., ";

int main(int argc, char* argv[]){
    int T=3e2;
    ofstream pat,txt;
    pat.open(argv[1],ios::out);
    txt.open(argv[2],ios::out);
    srand(time(NULL));
    while(T--){
        int variety=rand()%69+1;
        int len=rand()%1000;
        for(int cnt=0;cnt<len;cnt++)
            pat.put(str[rand()%variety]);
        for(int cnt=0;cnt<len*2;cnt++)
            txt.put(str[rand()%variety]);
        pat<<"bb\n";
        txt<<"b\n";
    }
    pat.close();
    txt.close();
}
```

圖(二)

```cpp
#include<bits/stdc++.h>

using namespace std;

string str="abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789., ";

int main(int argc, char* argv[]){
    int T=1e2;
    ofstream pat,txt;
    pat.open(argv[1],ios::out);
    txt.open(argv[2],ios::out);
    srand(time(NULL));
    while(T--){
        int variety=1;
        int len=rand()%1000;
        for(int cnt=0;cnt<len;cnt++)
            pat.put(str[rand()%variety]);
        for(int cnt=0;cnt<len*2;cnt++)
            txt.put(str[rand()%variety]);
        pat<<"bb\n";
        txt<<"b\n";
    }
    pat.close();
    txt.close();
}
```

下圖為範例測資執行結果：

```
D:\交大\C程式\作業類型\DataStructure\Lab01\Pack>find.exe 1.pat 1.txt
1 6 "mile" "you smile, I smile and everyone smiles"
1 15 "mile" "you smile, I smile and everyone smiles"
1 34 "mile" "you smile, I smile and everyone smiles"
0ms

D:\交大\C程式\作業類型\DataStructure\Lab01\Pack>find.exe 2.pat 2.txt
1 1 "Xx" "xxxxx"
1 2 "Xx" "xxxxx"
1 3 "Xx" "xxxxx"
1 4 "Xx" "xxxxx"
2 1 "oo" "ooxx"
2 3 "Xx" "ooxx"
16ms

D:\交大\C程式\作業類型\DataStructure\Lab01\Pack>find.exe 3.pat 3.txt
1 2 "123a b" "3123a bpp 12 3ab"
0ms
```

設計測資執行結果：

```
D:\交大\C程式\作業類型\DataStructure\Lab01\Pack>find.exe D1.pat D1.txt
1 9 "ab" "aaaaaaaaab"
0ms

D:\交大\C程式\作業類型\DataStructure\Lab01\Pack>find.exe D2.pat D2.txt
1 1 "abcabc" "abcabcabc"
1 4 "abcabc" "abcabcabc"
0ms
```

中型隨機測資結果：(含對照)

```
D:\交大\C程式\作業類型\DataStructure\Lab01\Pack>find.exe random.pat random.txt
KMP:
3089ms

D:\交大\C程式\作業類型\DataStructure\Lab01\Pack>find.exe random.pat random.txt
Brute Force:
2380ms
```

中型極端測資結果：(含對照)

```
D:\交大\C程式\作業類型\DataStructure\Lab01\Pack>find.exe extreme.pat extreme.txt
Brute Force:
56256ms

D:\交大\C程式\作業類型\DataStructure\Lab01\Pack>find.exe extreme.pat extreme.txt
KMP:
443ms
```

三、比較結果

單論比對效率，平均情況下，KMP、Brute Force 算法效率相差無幾，甚至於 KMP 由於額外的 Overhead，效率會略差於 Brute Force，在少數極端資料下，例如大量重疊、規律性的字串匹配，Brute Force 算法的複雜度會比 KMP 高，前者複雜度 O(nm)，後者複雜度為 O(n+m)。

圖為中型極端測資之執行結果，KMP 演算法消耗約 443ms，Brute Force 算法消耗約 56256ms，可以顯見其差異。

四、其他說明

1. 由於本次作業禁用 STL，因此這裡仿 STL 介面設計了一個非模板字串，配有 const char* casting, cascading, slice, allocate, find, random access, allocation, swapping, I/O, resizing 等功能，方法複雜度同於 STL。註：單元素 resizing 採用倍增均攤，平均複雜度 O(1)。
2. 另外，本次作業亦仿 STL 介面設計了一個泛型鍵結串列，配有各種類 modify、access、iterator 等功能。
3. 此次專案的輸出要求，在部分情況下會導致以 KMP 實作的版本複雜度降解至 O(nm)，原因在於匹配成立時必須印出全字串，導致輸出複雜度覆蓋了比對複雜度。
4. 此次作業所要達成目標為多重字串比對，事實上對於多重字串比對，較為合適的作法為 AC Automation，AC Automation 是 KMP 的變種、

它與資料結構 Trie 結合，將部分相同功能的次長相同前綴後綴的查詢合併至同個結點而形成樹狀結構，然而本次作業規定須以 KMP 實作，故不採用 AC Automation。