Searching Algorithm 效能實驗

10303118a,homework

題目

以 C++ 撰寫動態隨機產生一萬、五萬、十萬、十五萬、二十萬、二十五萬筆測資，分別採迭代二元搜尋法、遞迴二元搜尋法還有雜湊表的方式來實作搜尋並評比各自方式效能與差異。

TL;DR

實作

1. 資料產生器

```
std::deque<std::string> dataGenor(int count) {
    const char alphabet_list[] = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";
    const int alphabet_list_len = (sizeof(alphabet_list) / sizeof(char)) - 1;
    std::deque<std::string> retn_que;
    srand(time(NULL));
    while (count--) {
        std::string tmpStr = "";
        for (int i = 0; i < 6; i++)
            tmpStr += alphabet_list[rand() % alphabet_list_len];
        retn_que.push_back(tmpStr);
    }
    return retn_que;
}
```

這邊沒有太多的技術性可言。主要是隨機產生每一筆資料都是六個字母英文變數名稱，必須產生一萬、五萬、十萬、十五萬、二十萬、二十五萬筆資料當作每一次測資，因此如上方所示程式碼將產生亂數的部分以 `std::deque` 記憶，最後回傳。

2. 二元搜尋法

二元搜尋法概念在於先將資料排序，好比原始陣列資料為：`{"aa", "a", "ab", "aca", "abcde"}`，排序根據 ASCII 碼小的在前、大的在後，並且字串較短的置前反之置後；排序後獲得陣列為：`{"a", "aa", "ab", "abcde", "aca"}`。因此可以得知排序方式依據字串每一字元比對 ASCII 碼 大小來排序在前或後方，若透過

ASCII 碼大小無法區分時，再以兩者字串長短排序。

由以上邏輯配合 `std::sort` 函數即可傳入要排序的陣列起始元素、最終元素。而此函數的第三個參數為可選參數，可傳入自訂的比對函數，`std::sort` 將會以自訂的比對函數來做排序動作。以此概念而完成實作了以下排序函數用於字串陣列依大小排序：

```
/**
 * @function sortStrData
 *
 * @brief sorting strings in the deque via ASCII value and length
 * (this implement can be replaced by std::sort)
 **/
void sortStrData(std::deque<std::string> &arr) {
    struct {
        bool operator() (std::string s1, std::string s2) {
            for (int i = 0; i < s1.length() && i < s2.length(); i++)
                if (s1[i] != s2[i])
                    return s1[i] < s2[i];
            return s1.length() == s2.length() ? false : s1.length() < s2.length();
        }
    } compareFunc;
    std::sort(arr.begin(), arr.end(), compareFunc);
}
```

以上便是基於 `std::deque<std::string>` 所儲存的文字陣列實現排序的函數。由第十行至第十二行針對兩傳入字串由第 0 個字元開始依次比對下去，若兩字串同一位置上的字元不同則比較大小給予排序差異；若依次比對下去在兩字串上每個位置的字元皆一樣，那麼比較字串的長短給予排序差異。

不過事實上後面測試時發現 **直接調用 `std::sort(arr.begin(), arr.end())` 時預設使用strcmp因此與我們自行實作的排序結果一致**。因此這部分二元搜尋排序部分無需自己特別再次撰寫，逕使用 `std::sort` 函數預設方式排序即可。

迭代二元搜尋法

完成資料的排序後，即可對資料進行二元搜尋。迭代的二元搜尋方式即是不斷的將當前陣列對半切為左陣列、右陣列，比較陣列正中間的字串與目標字串大小，**若目標字串大小大於正中間的字串，代表目標字串被插入陣列時應該在右陣列，反之則在左陣列，並選定左右其中一方陣列**。接著將已選定陣列作為當前陣列，重複上述步驟，直至陣列正中間的字串等於目標字串即完成搜尋；或者已無法切割為左右陣列，代表找尋不到目標字串。

完成概念的程式碼如下：

```
int iterBinarySearch(std::deque<std::string> array, std::string str) {
    int left = 0, right = array.size() - 1, middle;

    while (left <= right) {
        middle = (right + left) / 2;

        if (!array[middle].compare(str))
            return middle;
        for (int i = 0; i < array[middle].length() && i < str.length(); i++) {
            if (array[middle][i] == str[i])
                continue;
            if (array[middle][i] > str[i])
                right = middle - 1;
```

```
            else
                left = middle + 1;
            break;
        }

    }
    return -1;
}
```

如上方所示程式碼為迭代二元搜尋法的核心程式碼。從一開始第二行處可見先訂出當前陣列的起點、終點與中間元素，我們將完整搜尋整個陣列，因此起點為 `0`、終點為 `陣列大小 - 1`。

接著第四行處開始進入迴圈比較，起點不可大於終點元素（當無法切割為左右陣列時不成立），迴圈內部先計算出正中間的元素，接著比較若正中間字串即是目標字串代表已找到目標完成搜尋；

若否，則依次比對正中間字串與目標字串的每個位置上字元之 ASCII 碼大小，若目標字串較小（`12: array[middle][i] > str[i]`）代表目標字串預期會在右陣列，故將終點元素設置為當前正中間元素來選取右陣列；反之若目標字串較大在代表預期在左陣列，故將起始元素設置為當前正中間元素來選取左陣列。

最後，若整個迴圈跑完，但都無法完成正中間字串等於目標字串的比對（`7: array[middle].compare(str) == 0`）代表陣列中不含有目標字串，回傳搜尋失敗的結果（`20: return -1;`）

呼叫時僅需以 `iterBinarySearch(SORTED_DATA, TARGET_STR);` 即可進行二元搜尋。

遞迴二元搜尋法

遞迴的概念並不會與迭代方式差異甚遠，概念相仿，將當前陣列切割為左右陣列，若正中間字串與目標字串一致則立即回傳搜尋完成；若否，則選取出左陣列或者右陣列傳入給自身函數再次作為當前陣列進行下一輪比對，直至找到或者搜尋失敗為止。實作完畢的程式碼如下：

```
int recuBinarySearch(std::deque<std::string> array, std::string str, int left, int right) {
    int middle = (right + left) / 2;

    if (left > right)
        return -1;

    if (!array[middle].compare(str))
        return middle;
    for (int i = 0; i < array[middle].length() && i < str.length(); i++) {
        if (array[middle][i] == str[i])
            continue;
        if (array[middle][i] > str[i])
            right = middle - 1;
        else
            left = middle + 1;
        break;
    }
    return recuBinarySearch(array, str, left, right);
}
```

呼叫時僅需以 `recuBinarySearch(SORTED_DATA, TARGET_STR, 0, SORTED_DATA.size() - 1);` 即可進行二元搜尋。

3. 雜湊表搜尋

建立雜湊表

雜湊表概念在於先將要記錄的資料的 KEY（通常可能為該筆資料的 ID、名字、或者任何其他足以代表此筆資料獨特性的資訊）進行雜湊取得*雜湊值*，並將雜湊表中的雜湊值對應處紀錄下這筆資料。

根據雜湊表分群的大小一般極有可能遇到雜湊值碰撞的問題，即兩筆不同資料的雜湊值是一致的，導致兩筆測資需要記錄在**雜湊表的同一處上**。這部分依據上課所教授內容，雜湊表的雜湊處會以一個鏈串節點作起點，每次有新的測資進來便將新資料插入在此鏈串節點起點與舊資料中間。

```
1 int hashStr(std::string s) {
2     int cHash = 0, p = 0;
3     while (p < s.length()) cHash += s[p++];
4     return cHash;
5 }
```

雜湊方式可參照上面所示為一簡單雜湊實作，透過將字串中字元的總和作為雜湊結果，不過通常實務上會以較不容易發生雜湊值的實作，如 MD5、SHA1、SHA256 等雜湊方式。

雜湊表簡易實作過程如下：

```
1 /* hashing table search */
2 const int groupCount = 26;
3 std::deque<std::string> hashTable[groupCount];
4 
5 for (int i = 0, p = 0; i < DATA.size(); i++) {
6     p = hashStr(DATA[i]) % groupCount;
7     hashTable[p].push_front(DATA[i]);
8 }
```

雜湊表建立過程如上面所示。在這邊我將雜湊表設計為有二十六個字串鏈串節點，因此第三行可見我以 `std::deque<std::string>` 作為鏈串結點，並以此宣告了二十六個物件組出的陣列作為雜湊表。

將一個個字串從測資中新增進雜湊表過程由第五行處開始可見，第六行處 `p = hashStr(DATA[i]) % groupCount;` 我將目標字串透過雜湊函數計算出雜湊值並對二十六取餘數、決定當前此筆目標字串被分類在哪一群中，並將此目標字串插入進舊的字串之前。

上面整體過程便完成了雜湊表的建立。

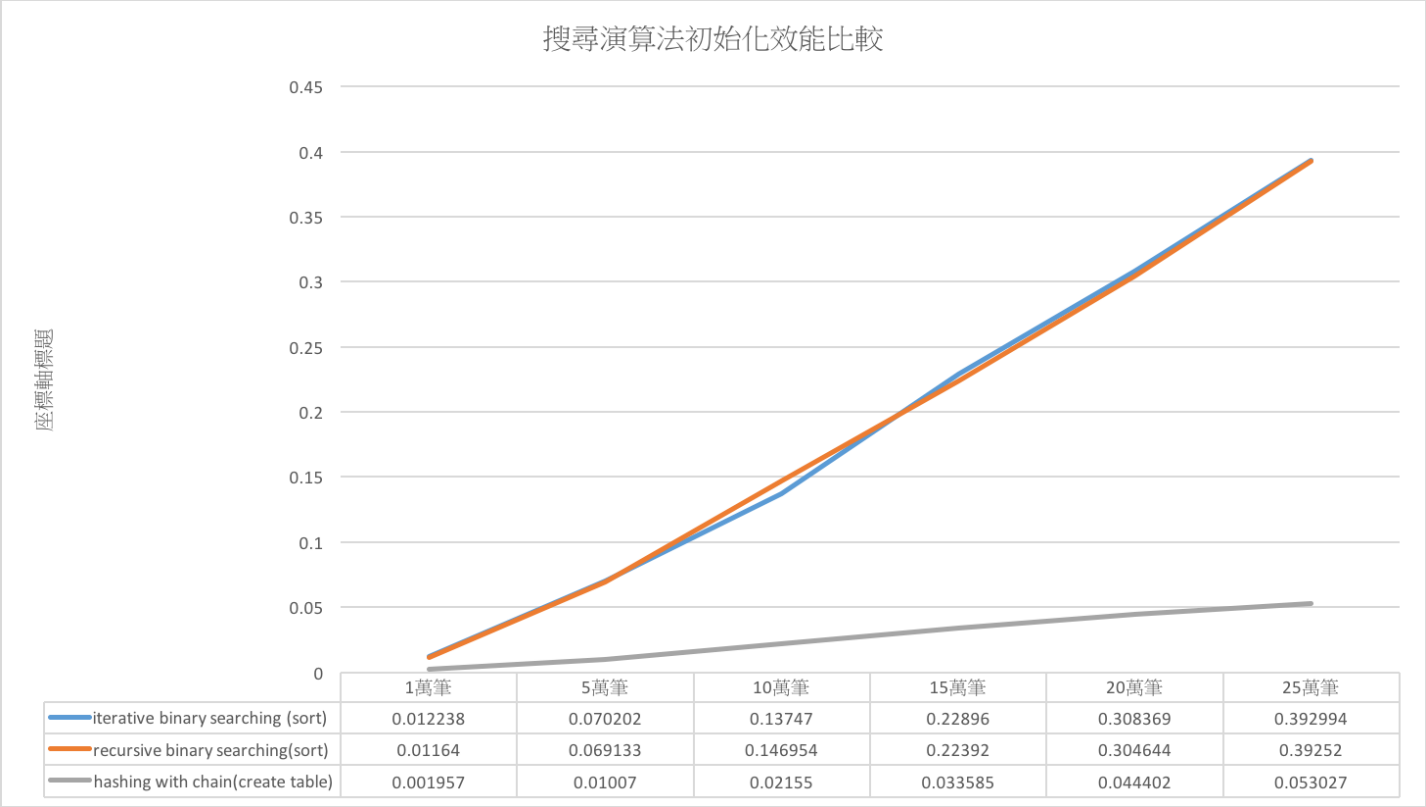
在雜湊表中搜尋

在前面建立雜湊表的過程，已經讓搜尋過程顯而易見可知，僅需將待查詢之目標字串先做雜湊取得雜湊值、並分群決定此目標字串預期在二十六群中的哪一群。接著便可遍歷該群鏈串中的每個字串，確認是否存在目標字串，即可快速地查詢完畢找到結果。實作程式碼如下：

```
1 int pos = hashStr(TARGET_STR) % groupCount;
2 
3 for (int i = 0; i < hashTable[pos].size(); i++)
4     if (!hashTable[pos][i].compare(TARGET_STR))
5         puts("found");
6 
7 puts("not found");
```

4. 結論與心得

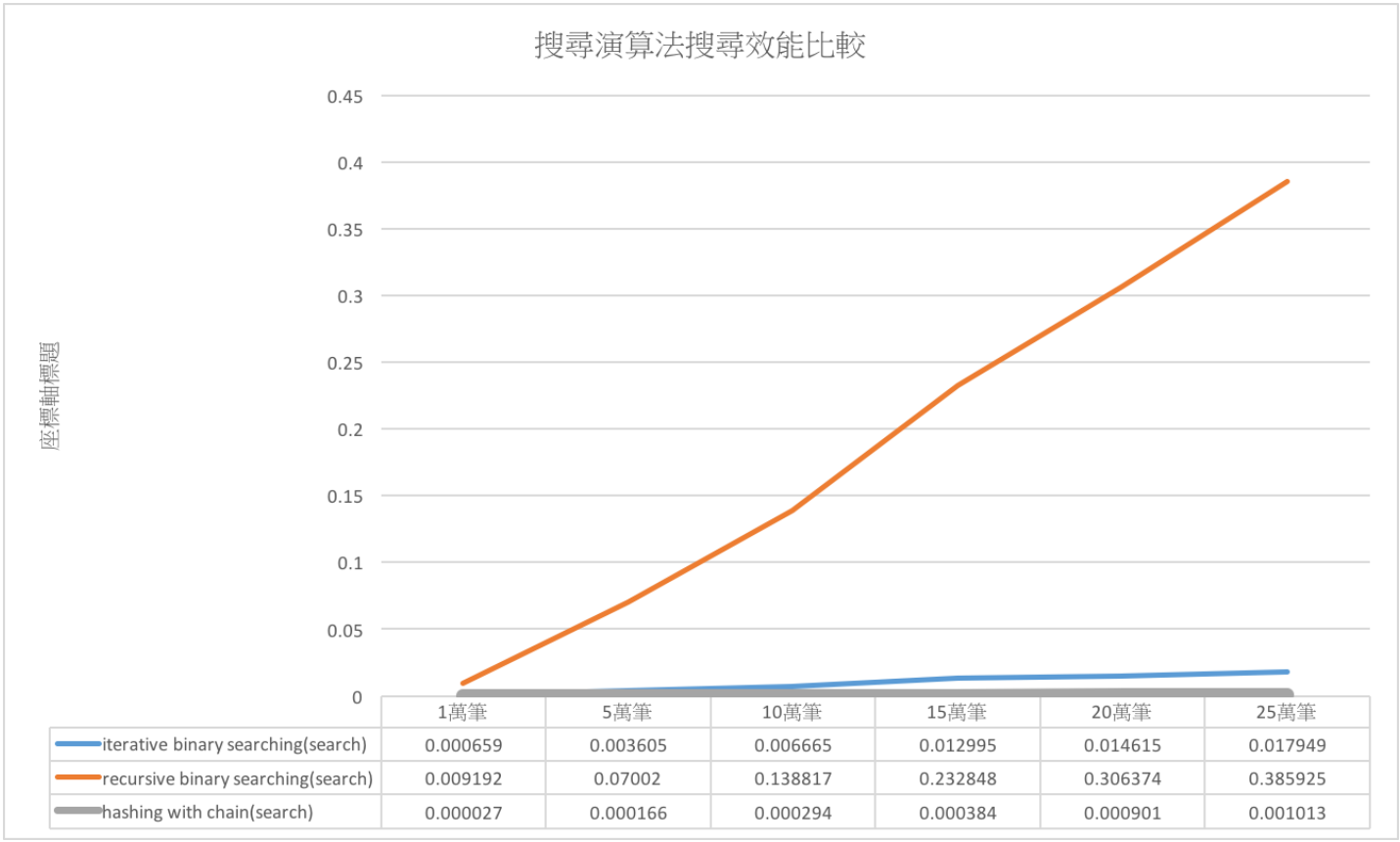
附註: 完整測試資料用程式碼可於此檢閱隨本文件附帶的 search_algorithm.cpp



	1萬筆	5萬筆	10萬筆	15萬筆	20萬筆	25萬筆
iterative binary searching (sort)	0.012238	0.070202	0.13747	0.22896	0.308369	0.392994
recursive binary searching(sort)	0.01164	0.069133	0.146954	0.22392	0.304644	0.39252
hashing with chain(create table)	0.001957	0.01007	0.02155	0.033585	0.044402	0.053027

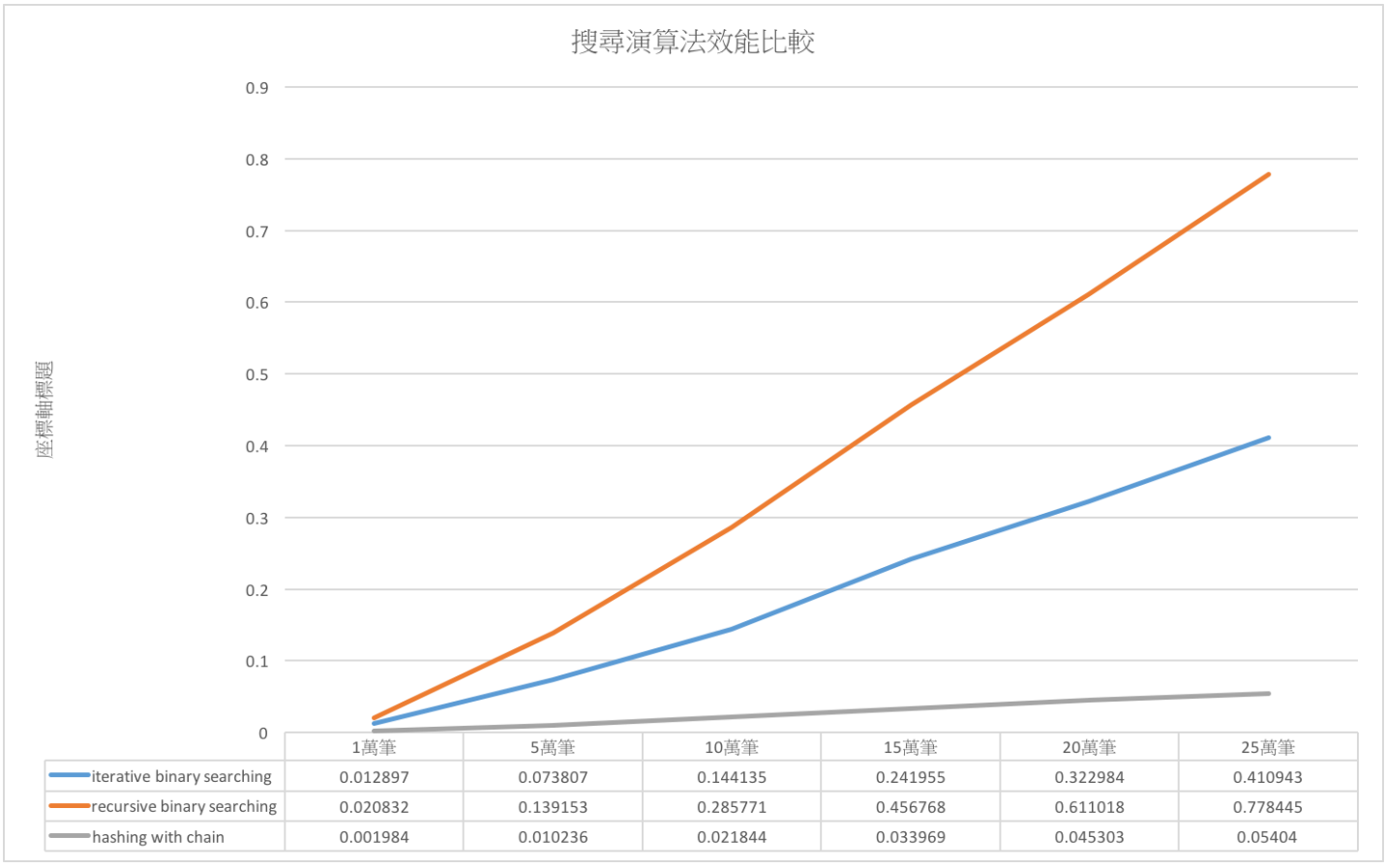
上述實作中可發現不管是迭代二元搜尋或者遞迴二元搜尋**都必須先將資料做排序**；比起二元搜尋法，雜湊表的方式僅需直接將每一筆資料都先計算雜湊值並記憶於雜湊表中，不需要一有新的資料進來時便需要大費周章將所有資料排序一次。因此二元搜尋比起雜湊表還來得耗時許多。

如上圖表可見比起二元搜尋演算法初始化時得先對資料排序，建立雜湊表的方式所耗費的時間成本來得低需多。



	1萬筆	5萬筆	10萬筆	15萬筆	20萬筆	25萬筆
iterative binary searching(search)	0.000659	0.003605	0.006665	0.012995	0.014615	0.017949
recursive binary searching(search)	0.009192	0.07002	0.138817	0.232848	0.306374	0.385925
hashing with chain(search)	0.000027	0.000166	0.000294	0.000384	0.000901	0.001013

而實際測試時，三種演算方式過程在對同樣的目標字串、同樣的輸入資料進行測試發現，在搜尋過程中雜湊表搜尋耗時明顯佔據了優勢。而迭代二元搜尋比起遞迴方式效能來得好許多，因為在遞迴的過程中會在行程（Process）的堆疊（Stack）上佔用大量空間存放函數返回指標、取回函數指標，這期間耗費大量時間成本在程式計數器（PC, aka Program Counter）上，詳細可參考 x86 Calling Convention。



	1萬筆	5萬筆	10萬筆	15萬筆	20萬筆	25萬筆
iterative binary searching	0.012897	0.073807	0.144135	0.241955	0.322984	0.410943
recursive binary searching	0.020832	0.139153	0.285771	0.456768	0.611018	0.778445
hashing with chain	0.001984	0.010236	0.021844	0.033969	0.045303	0.05404

由前面資訊得知沒有意外的總體效能評比如上圖表所示，雜湊表搜尋方式完整從建立雜湊表到搜尋的過程所

耗費的時間成本相當的低廉。而迭代二元搜尋因為不需要大量的呼叫過程而壓倒性的比遞迴二元搜尋來的效能優異。