Homework2 Report 109550171 陳存佩

Part 1 Sorting

1. Introduction (Introduction of different method/ algorithm)

A. ***Mergesort:*** a Divide and Conquer algorithm，先把陣列不斷對半分，分割到不能再更小後再把他們結合回去，結合的過程中進行排序。stable and out-of-place sort

B. ***Quicksort:*** a Divide and Conquer algorithm，找到 pivot 並確保比 pivot 小的數都在左邊，比 pivot 大的數都在右邊，持續做直到排序完畢，un-stable and in-place sort

2. Implement Details

A. ***Mergesort:*** 用遞迴寫，遞迴 function 叫 cut，當陣列的頭的 index 還小於尾時，就進行切割，從中點對半分，對半分完的兩個陣列再進去 cut 持續做切割直到不能再切，切完後再兩兩進入 merge 這個 function，建兩個空的陣列把左右兩個陣列暫存在裡面（此時這兩個陣列各自都是以排序好的），透過比大小的方式再依大小順序放回原本的陣列，如此一步一步把整個陣列再次組合起來並排序。

B. ***Quicksort:*** 用遞迴寫，遞迴 function 叫 quickSort，當陣列的頭的 index 還小於尾時，就進入 partition 這個 function，pivot 設為陣列最尾端，i 表示下一個小於 pivot 的數將要存放的位置（故只有當順利放入後 i 才會 ++），j 從陣列的頭逐一檢查哪個數字比 pivot 小，比較小的話就放入 i 索引值的位子，partition 做完的效果會是 pivot 會在正確位置，它的左比它小，它的右邊比它大。接著回到 quickSort，針對 pivot 兩邊的兩個陣列再分別進行 quickSort，重新尋找 pivot，一直重複直到排序完成。

3. Results

A. time complexity

Mergesort: O(nlogn)

```
void mergeSort::cut(int b, int e){ T(n)
    if (b < e) {
        int m; O(1)
        m = (b + e) / 2; O(1)
        this->cut(b, m); T(n/2)
        this->cut(m + 1, e); T(n/2)
        this->merge(b, m, e); O(n)
    }
}

void mergeSort::merge(int b, int m, int e){
    vector<int> temp1;
    vector<int> temp2;
    int i, j;
    for (i = b; i <= m; i++) { O(n)
        temp1.push_back( (this->v)[i] );
    }
    for (j = m+1; j <= e; j++) { O(n)
        temp2.push_back( (this->v)[j] );
    }
    i = j = 0;
    for (int k = b; k <= e; k++) { O(n)
        if (i < temp1.size() && j < temp2.size()) { //當temp1和2都還有東西
            if (temp1[i] < temp2[j]) { //放入temp1 (當它較小時)
                (this->v)[k] = temp1[i]; O(1)
                i++; O(1)
            }
            else if (temp1[i] >= temp2[j]) { //放入temp1 (當它較小時)
                (this->v)[k] = temp2[j]; O(1)
                j++; O(1)
            }
        }
        else {
            if ( j >= temp2.size() && i < temp1.size()) { //放temp1 (當temp2已經空了)
                (this->v)[k] = temp1[i]; O(1)
                i++; O(1)
            }
            else if ( i >= temp1.size() && j < temp2.size()) { //放temp2 (當temp1已經空了)
                (this->v)[k] = temp2[j]; O(1)
                j++; O(1)
            }
        }
    }
    temp1.clear(); O(n)
    temp2.clear(); O(n)
}
```

O(n)

T(1) = 1

T(n) = O(1) + O(1) + T(n/2) + T(n/2) + O(n)= 2*T(n/2) + O(n)

= 2(2*T(n/4) + O(n/2)) + O(n)= 4(T(n/4)) + 2*O(n) + O(n)

= $2^k T(n/2^k) + k*O(n) = 2^{\log n}*1+\log n*O(n)$ = O(nlogn)

Quicksort: O(nlogn)

```
void quickSort::sort(int b, int e){T(n)
    if (b < e) { //頭小於尾 O(1)
        int r = 0 ;O(1)
        r = this->partition(b, e); O(n) //pivot在經過partition後會跑到正確大小的位置
        this->sort(b, (r - 1));T(n/2)
        this->sort((r + 1), e);T(n/2)
    }
}

int quickSort::partition(int b, int e) {
    int pivot = (this->v)[e];O(1)
    int i = b - 1;O(1)
    for (int j = b; j <= e; j++) {O(n)
        if ((this->v)[j] <= pivot) {O(1)
            i++; //代表下一個小於pivot的數字的位置O(1)
            swap((this->v)[i], (this->v)[j]);O(1)
        }

    }
    return i;
}
```

O(n)

T(1)=1

T(n) = O(1) + O(1) + T(n/2) + T(n/2) + O(n) = 2*T(n/2) + O(n)

= 2(2*T(n/4) + O(n/2)) + O(n) = 4(T(n/4)) + 2*O(n) + O(n)

= $2^kT(n/2^k) + k*O(n) = 2^{logn}*1+logn*O(n)$ = O(nlogn)

B. Execution Time

我用 20 個連續數字的 random、few unique、reversed、almost sorted 四種情況對兩個演算法做測試（詳細的測資附在報告最後一頁）

a. MergeSort: 預想最差狀況是 reversed，因為每次合併都要重新排序，預想最佳狀況是 almost sorted 但不如預期，可能是因為測資設計得不好。

i. Average Time: 0.0040825

ii.Fastest Time: 0.003 (few unique)

iii.Slowest Time: 0.00533 (reversed)

b. QuickSort: 預想最差狀況是 reversed，造成每次都選到最大或最小的 pivot，造成分割出來的陣列不平均，最佳狀況是 few unique，因為 pivot 剛好選到數列的中位數（見附錄測資），陣列分割平均

i. Average Time: 0.00383

ii.Fastest Time: 0.00333 (few unique)

iii.Slowest Time: 0.00433 (reversed)

C. Stability: MergeSort 是 stable，因為在切割的時候不會動到相對位置，合併的時候也是照大小合併，如果一樣大的話就是左邊先放，QuickSort 是 un-stable，舉例來說 input: 4 2 1 4* 3 (pivot 為 3) -> 2 4 1 4* 3 -> 2 1 4 4* 3 -> 2 1 3 4* 4，4 和 4*的位置互換了

4. Discussion

   A. *Which is better algorithm in which condition:* 當需要 stable 以及具有額外空間可利用的時候較適合使用 MergeSort，當不需要 stable 且沒有額外空間可使用的時候適合用 QuickSort，兩者的時間複雜度相同。
   B. *Challenges you encountered:* 一開始寫遞迴的時候有點卡住，後來透過用紙筆畫圖演練整個演算法的過程就好很多。

5. Conclusion

   A. *What did you learn from this homework:* 更加了解遞迴的思考與實作模式以及學會先用紙筆演練出演算法的過程再進行實作，也體會到設定洽當的遞迴終止條件的重要性。
   B. *How many time did you spend on this homework:* 一到兩天左右。
   C. *Feedback to TA:* 透過這次的實作更加了解 sort 的部分，也認識了算時間的相關函式。

Part 2 Minimum Spanning Tree

1. Introduction (Introduction of different method/ algorithm)

   A. ***Kruskal:*** a greedy algorithm，把重點放在 edge 上，從最小 cost 的 edge 開始看起，過程中避開 cycle
   B. ***Prim:*** a greedy algorithm，把重點放在 vertex 上，類似最短路徑的方法假設權重（除了起點其他設為無限），一樣從最小權重的 vertex 開始看起。

2. Implement Details

   A. ***Kruskal:*** 先建立一個 set struct 儲存 vertex，把每個 vertex 的 parent 都設成自己，每個 vertex 都是一個獨立集合，一開始都只有自己一個。接著把 edge 依 cost 排序，從最小的 cost 開始，找到 edge 兩端 vertex 的源頭 parents，檢查看看是否相同，這邊使用 findParent 遞迴，一個 vertex 的 parent 上面可能還有 parent（類似一顆 tree，源頭就是根），要一路尋找到該 vertex 的 parent 就是自己，就代表他是源頭，如果兩個 vertex 源頭不相同，則比較兩集合的內容數量，比較少的併到比較大的，合併的方法是把較小集合的源頭改成較大集合 vertex 的 parent，這樣就是把小樹併在大樹的分支下面，合併完的樹中的每個 vertex 使用 findParent 都會找到同樣的源頭。若 edge 兩端的 vertex 已在同個集合就不用合併，以免造成 cycle。
   參考網站：https://sites.google.com/site/zsgititit/home/jin-jiec-cheng-shi-she-ji-2/zui-xiao-sheng-cheng-shu

B. ***Prim:*** 先建一個 vertex 的 struct，放入一些基本的參數（像是 weight, parent, visit, index），建立 adjacency matrix，當還有 vertex 尚未被 visit 過就進行以下動作：找出還沒被 visit 過的最小權重 vertex 設為 target，檢查每個和 target 相鄰且沒被 visit 過的 vertex，如果該 vertex 目前的權重大於 edge cost，則更新 vertex 的權重和 parent，最後把 target 設為 visit，最後檢查是否所有 vertex 都被 visit 過，若沒有的話繼續進入 while 迴圈。

3. Results (time complexity)

Kruskal: O(ElogE)

```
set* s;
s = new set[v_num];
for (int i = 0; i < v_num; i++) {                                   O(V)
    s[i].parent = i;O(1)
    s[i].num = 1; O(1)
}

sort(e, e + e_num, compare); //sort by e's cost use compare funciton   O(ElogE)
for (int i = 0; i < e_num; i++) {O(E)
    if (findParent(s, e[i].a) != findParent(s, e[i].b)) {O(V)如果兩個vertex最上面的parent不一樣
        if (s[e[i].a].num >= s[e[i].b].num) {O(1)a's subsets >= b's
            s[findParent(s, e[i].b)].parent = s[e[i].a].parent; O(V)把b最上面的parent的parent接到a的parent
            s[e[i].a].num++; O(1)
        }
        else {                                                      O(EV)
            s[findParent(s, e[i].a)].parent = s[e[i].b].parent; O(V)
            s[e[i].b].num++; O(1)
        }
        total_cost += e[i].cost; O(1)
    }
}

int findParent(set* s,int i) {T(V)return the toppest parent
    int ans = i;O(1)
    if (i != s[i].parent) {O(1)
        ans = findParent(s, s[i].parent); T(V-1)
    }
    return ans;
}
```

O(V)
+O(ElogE)
+O(EV)
= **O(ElogE)**

T(1) = O(1)

T(v) = O(1)+O(1)+T(v-1) = O(1)+O(1)+2*T(v-2) = O(1)+O(1)+k*T(v-k) = (v-1)*T(1) = O(v)

Prim: O(V²)

4. Discussion

A. *Which is better algorithm in which condition:* 他們都是貪婪演算法，當 edge 較少時使用 Kruskal，因為它是以 edge 為焦點，當 vertex 較少時使用 Prim，因為它是以 vertex 為焦點。

B. *Challenges you encountered:* 在寫 Kruskal 時，處理 disjoint set 時花了很多心思，想了蠻多種方法去檢測兩個 vertex 是否在同一個集合內，後來查到了用類似 tree 的方式去表達不同 vertex 的 parent 關係。

5. Conclusion

A. *What did you learn from this homework:* disjoint set 實作的方法，也透過實作確切體認到教授上課說的 Kruskal 以 edge 為焦點，Prim 以 vertex 為焦點的概念，因為在實作 Kruskal 時都是以 edge 為出發點考慮，主要資料也是透過建 edge struct 來儲存，實作 Prim 時則都是以 vertex 為主。

B. *How many time did you spend on this homework:* 兩三天，disjoint set 的部分想很久。

C. *Feedback to TA:* 實作真的可以讓自己對於這個演算法的印象加深很多，蠻喜歡這個作業的！

Part 3 Shortest Path

1. Introduction (Introduction of different method/ algorithm)

   A. ***Dijkstra:*** 可以用來計算有正權重、方向性 graph 的最小路徑，但不能處理負權重的情況。

   B. ***Bellman-Ford:*** 可以用來處理所有 graph（包含負權重），並能夠偵測出 negative cycle（當有負環時沒有最小路徑），實作方法簡單但是時間複雜度較高。

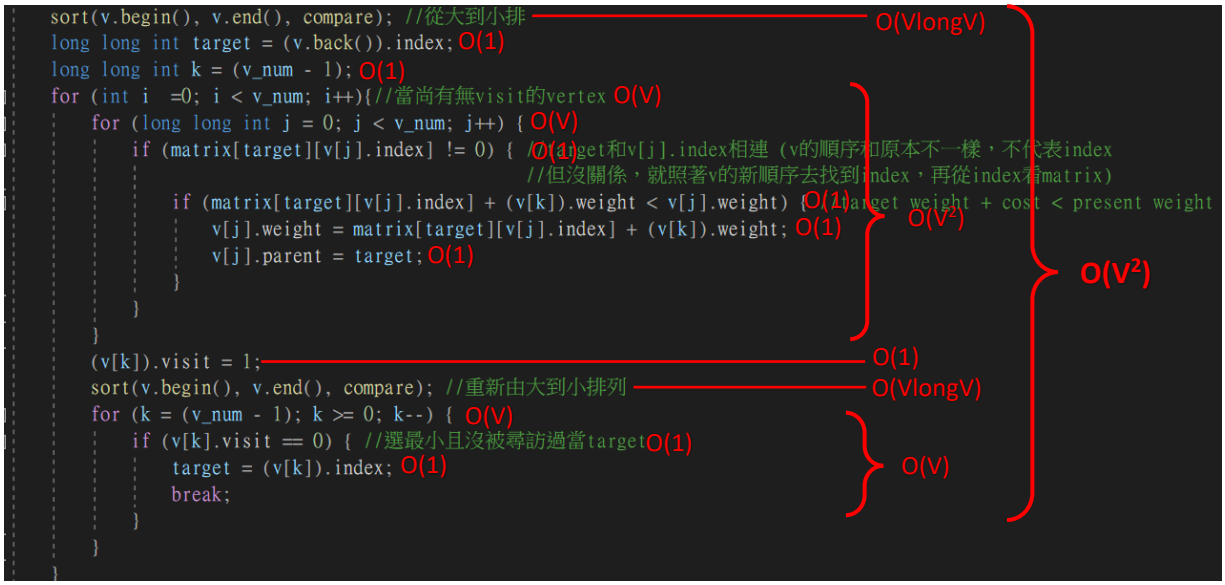
2. Implement Details

   A. ***Dijkstra:*** 建立 vertex struct 來存放 vertex 的各種性質，先設所有 vertex 權重為無限（這邊設為 100 萬），原點的權重設為 0，並建立一個 adjacency matrix 來儲存 graph 內每條路徑的 cost，重複以下動作直到 vertex 被尋訪完為止：每次都取出權重最小的 vertex（以下稱為目標），找到和它相鄰的 vertex（在 matrix 內的對應位置不為零），若該相鄰 vertex 現有的權重大於目標加上 cost，則更新相鄰 vertex 的權重跟 parent，處理完所有相鄰的 vertex 後把目標設為已參訪過，繼續找下個目標。

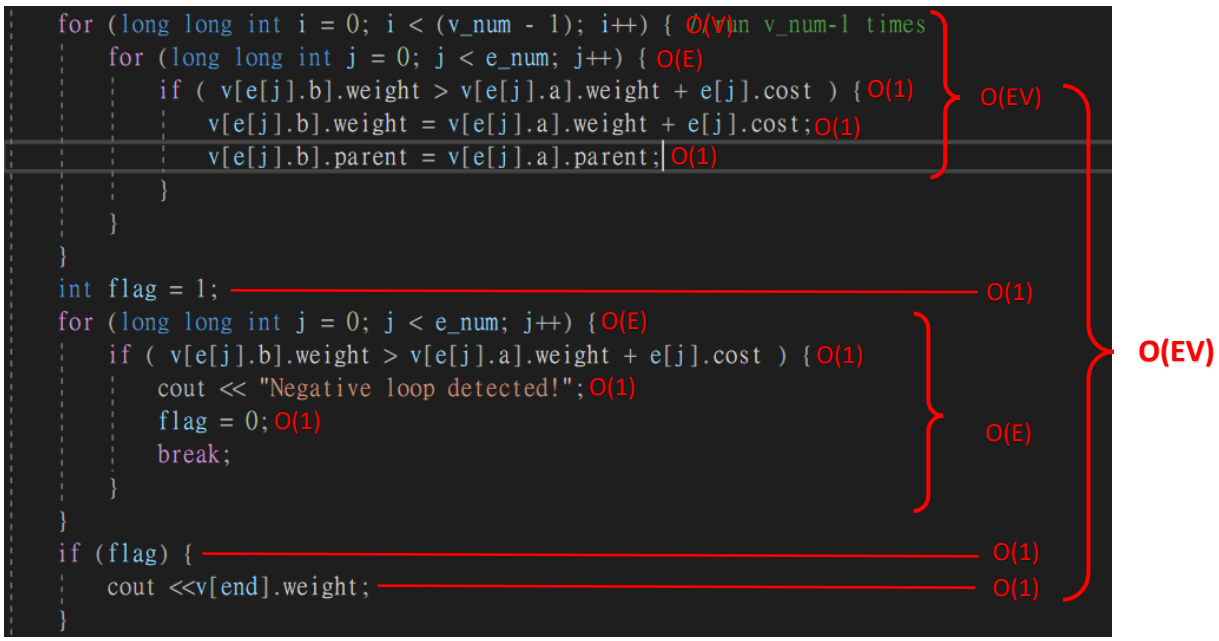
   B. ***Bellman-Ford:*** 建立 vertex 和 edge struct 來存放他們的各種性質，先設所有 vertex 權重為無限（這邊設為 100 萬），原點的權重設為 0，再依序輸入每個 edge 的起點終點以及 cost，主要程式是雙層迴圈，外層迴圈要做 vertex 的數目-1 次，以確保已經找到最小路徑，內層迴圈是尋訪每個 edge，如果 edge 終點的權重大於起點加上 cost，則更新該終點的權重跟 parent。結束上述迴圈後，進行負環測試，如果這時候權重還可以變更小就表示有負環，此時沒有最短路徑。

3. Results (time complexity)

Dijkstra: O(V²)



Bellman-Ford: O(EV)



4. Discussion

A. *Which is better algorithm in which condition:* 當出現負權重時只能使用 Bellman-Ford，也只有 Bellman-Ford 可以偵測 negative cycle，但是當 edge 數太多時，Bellman-Ford 時間複雜度會飆升，若又沒有負權重，就較適合用 Dijkstra

B. *Challenges you encountered:* 程式碼寫完後發現太大的測資就無法處理，後來發現是因為一開始我處理「設定權重為無限大」時，我是設無限為 edge cost 的最大值+1，但我後來發現權重會比 edge cost 還要大，因為還要考慮 edge cost 加上 vertex 的權重，故後來設成更大的數確保不會超過。

C. *Describe how you detect the negative loop in the Bellman-Ford Algorithm:* 當進行 vertex 數-1 次的迴圈後（若沒有負環的話此時權重都不會再改變），進行負環測試，如果這時候權重還可以變更小就表示有負環（會不斷讓權重變小），此時沒有最短路徑

D. *If you need to print out the path of the shortest path, describe how it can be done?* 善用 vertex 內的 parent 參數，先找到終點 vertex 的 parent，再找到這個 parent 的 parent，一路往回推可以找到起點。（各個 vertex 中的 parent 就是要完成這個最短路徑該 vertex 上個經過的 vertex）。

E. *Compare the time complexity of the two algorithms:* Dijkstra 的複雜度是 $O(V^2)$，Bellman-Ford 是 O(EV)，當 edge 數量太多時用 Bellman-Ford 就會較沒效率。

5. Conclusion

A. What did you learn from this homework? 雖然虛擬碼看起來很簡單，但是實作其實需要注意蠻多小細節，像是 Dijkstra 中，vertex 經過排序後 vector 的 index 就和原本代表的不一樣了，需要再多新增一個參數紀錄原本的 index，因為 adjacency Matrix 適用原本的 index。設成無限那邊也需要多花點心思設定適合的邊境條件（見 4B）。

B. How many time did you spend on this homework? 兩三天，較大測資沒有過的問題思考了很久才想到解決辦法（見 4B）。

C. Feedback to TA

D. 覺得 part3 是這次 H.W.2 最難的部分。

附錄：sorting 測資

random

20

1 8 20 5 13 6 10 16 2 18 4 12 9 7 11 19 17 15 3 14

0

merge: 0.004

quick: 0.004

few unique

20

1 2 3 2 2 1 3 1 2 1 3 2 1 2 3 3 1 2 1 2

0

merge: 0.003

quick: 0.00333

reversed

20

20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

0

merge: 0.00533

quick: 0.00433

almost sorted

20

1 3 2 4 5 7 6 8 9 10 12 11 13 15 16 14 17 18 20 19

0

merge: 0.004

quick: 0.00366

因為發現每次使用 time 函式跑出來的結果會有小幅誤差，故上面數據都是同樣測資丟三次取平均的結果