Part 1 Directed Graph

Implementation Details

Steps

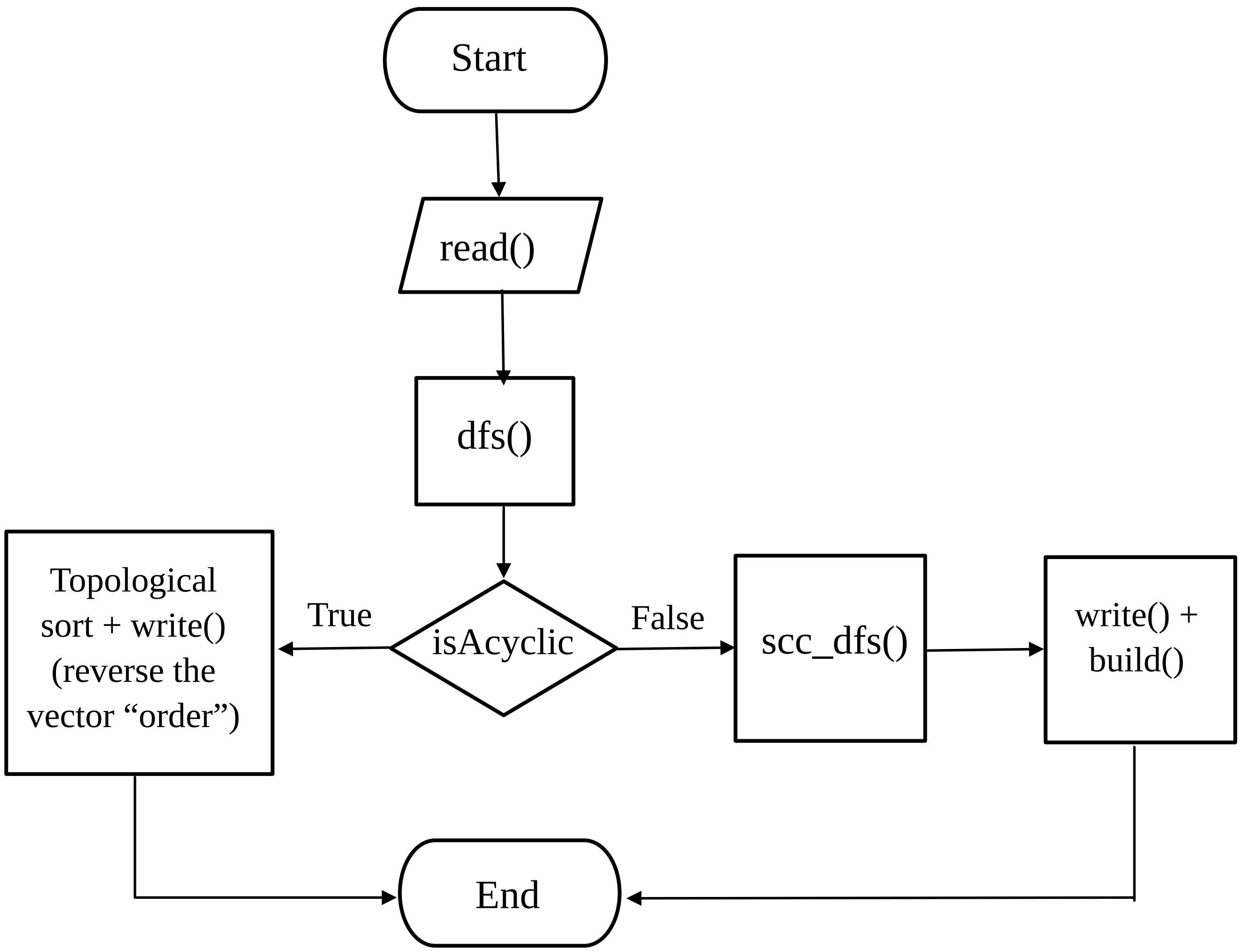
1. Read 函式讀取檔案輸入，第一行的兩個數字分別代表點的數目和路徑的數目，因此先建立一個 adjacency matrix，並根據接下來讀取到的輸入完善 adjacency matrix。
2. 進入到 Solve 函式後，初始化變數名稱為 visit 的 vector 為 0，為了紀錄等下 DFS 時造訪的次數，接著 visit 從索引值 i=0 開始，若找到 visit[i]=0，進入 DFS。
3. DFS 的目的是為了尋找此圖上是否有 SCC。跟平常找尋路徑時用的 DFS 類似，只是需要改寫一些地方，在 DFS 到此點時，先判斷其對應的 visit 值是否為 0，若為 0 將此點的 visit 值變成 1，並繼續搜尋下一個可到達的路徑；若為 1，則確定此圖有 SCC 的存在，將布林變數 isAcyclic 改為 false；若為 2，則因為此點已造訪且結束，所以不造訪此點。最後搜尋完所有可行的路徑後，在結束前將此點的 visit 值改為 2，並將此點推入 order 中，表示結束的先後順序。
4. 結束第一次的 DFS 後，判斷 isAcyclic 的值。

   4-1. 若為 true，則進行 DFS 尋找 DAG 的拓樸排序並輸出，而經過觀察後，發現 order 中的順序反過來後便是正確的輸出，因此我

們便能得到此 DAG 的拓樸排序。

4-2. 若為 false，進行第二次的 DFS，名為 scc_dfs 的自訂函式，進入 DFS 的順序則為 order 的最後一個到最前，且整個圖的路徑方向要改為相反的，在第二次 DFS 時，找到不能再找為止前所經過的點皆屬於同一個環上，因此進入 build()函式將這些點集合成一個大點，並依序編為 0、1、……n 的編號。接著尋找這些集合的點之間是否有相連的路徑，並將其合併成一大條(weight 相加)，最後依照題目輸入那樣，輸出點的數量、路徑數、路徑。

Flow Chart



Discussion

Discover

- 在進行完第一次 DFS 後，若圖為 DAG，且以 DFS 方式進行拓樸排序輸出時，發現輸出順序是第一次 DFS 結束時間由大排到小的結果。

Challenge Encountered

- 一開始並不清楚如何分辨一張圖是否為 DAG 或者是 SCC，且對這兩者的概念還很模糊，因此完全沒有想法，不知從何處下手。後來 google 相關資料有了一定概念後，便開始邊做邊測試，過程中逐漸對 SCC 和 DAG 有了更深入的了解。
- 測試輸出結果時，發現拓樸排序跟正確輸出不同，於是用紙筆寫出每一步，最終找到排序的問題，且在手寫下來後，發現與我的 vector “order”反過來的順序一樣，因此直接省略重新找拓樸排序的函式，直接將 vector “oreder”由後往前輸入檔案。

Optimized

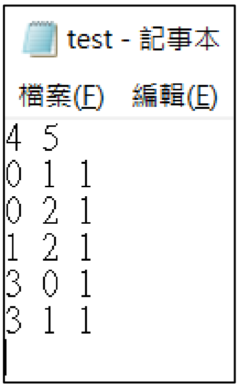
- 最後的步驟找各個 SCC 間連接的路徑時，原本想用 adjacency matrix 一行一行去進行判斷，但發現這樣做的時間複雜度會來到 O(V^2)，V=點的數量。

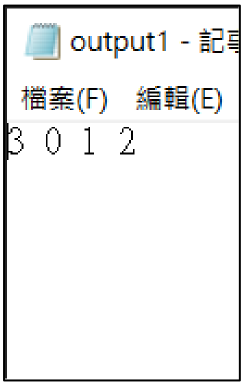
  優化辦法：另外開一個 vector<pair<pair<int, int>, int>>>存放兩點的權重，因此在合併路徑時，只需尋訪過 vector 一次，時間複雜 O(E)，

E=邊的數目，the worst case: O(E)= O(V^2)，但多數情況仍比原本想法還要有效率。

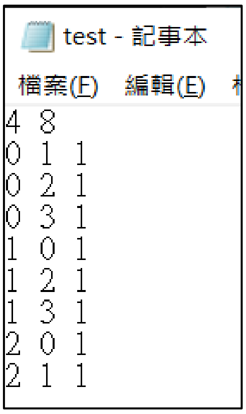
Results

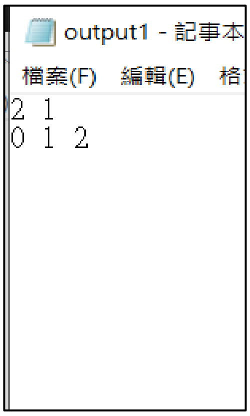
- Test Input & Test Output：輸出與範例結果吻合





- Test Input & Test Output：輸出與範例結果吻合





Part 2 Shortest Path

Implementation Details

Steps

1. Read()函式讀取輸入的側資，並根據測資建立相應的 adjacency matrix，若兩點之間無連線時，將長度設為一個極大值(1e9)。

2. 進入到 Solve()函式，利用兩種演算法來找尋從 0 到最後一個點的最短路徑，分別為 Dijkstra 和 Bellman-Ford：

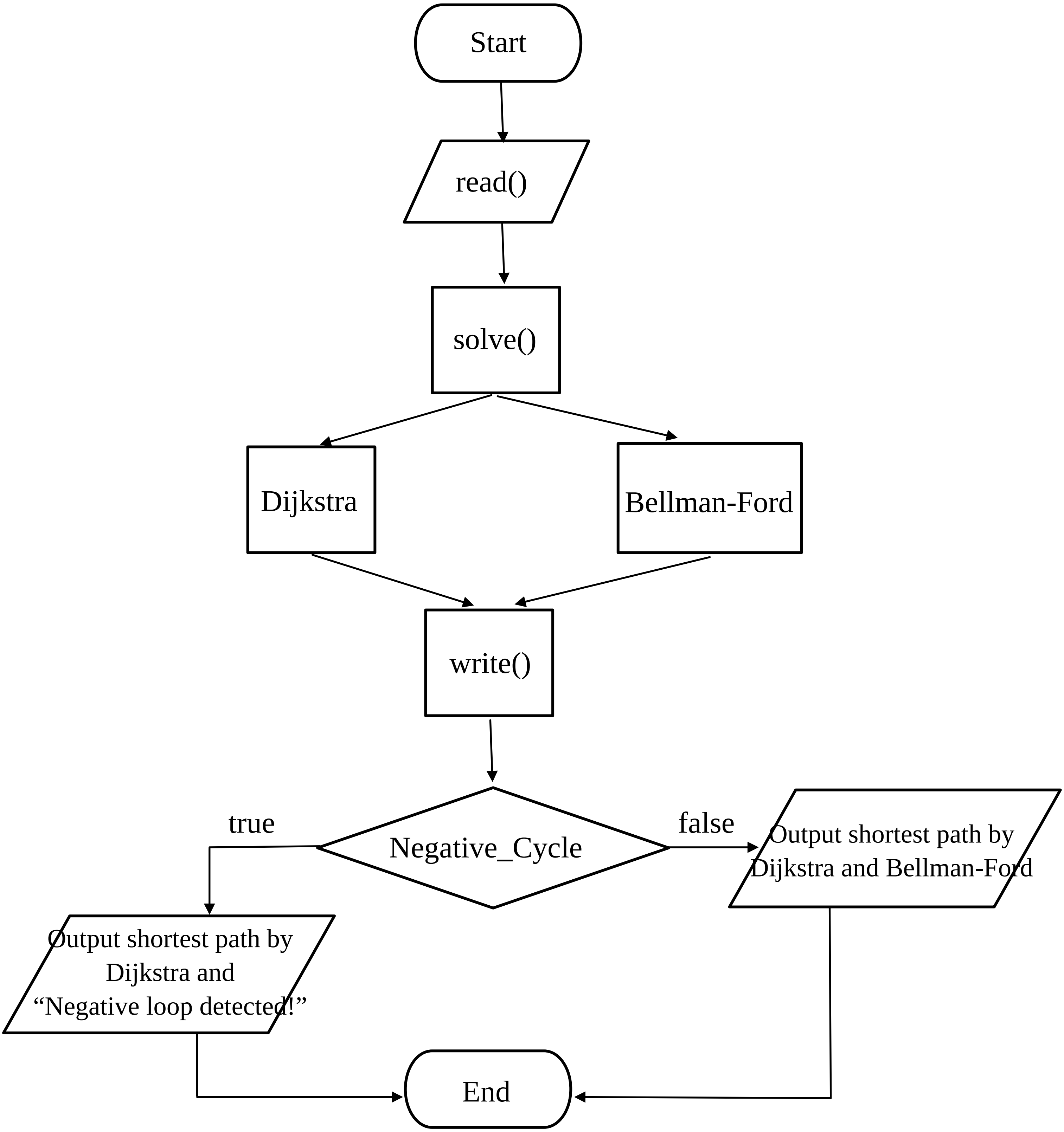
   2-1. Dijkstra：考慮到此演算法無法計算負權重的情況，在進行此演算法時，將所有的邊都進行絕對值一次，確保邊不會是負數。

   vector “abs_d”是記錄 0 到每一點的最短距離，一開始將每一個初始化為極大值(1e9)，除了 abs_d[0]=0。每一次從 abs_d 中找最小的路徑值且還未造訪過的，利用此點去更新看是否到其他點又會有更短的路徑，反覆循環次數最多不超過頂點數量，最終就能得出點 0 到所有點的最短路徑。

   2-2. Bellman-Ford：此演算法可以計算負權重，但可能會出現負環，造成最短路經出現負無限的情況，因此以三層巢狀迴圈進行完一輪的最短路徑後，還需再以兩層的巢狀迴圈檢查是否還能再找到最小路徑，若可以則代表此圖有負環的情況，將返回 true 值給變數 negative_Cycle，若未能再找到更小的值，代表並無負環，返回 false 值給變數 negative_Cycle。

3. Wriite()函式是輸出結果到檔案上，第一行為 Dijkstra 後最短路徑長的結果，第二行為 Bellman-Ford，但在輸出第二行之前，先判斷變數 negative_Cycle，若為 true，輸出”Negative loop detected!”，若為 false，輸出 0 到最後一點的最短路徑長。

Flow Chart


Start
read()
solve()
Dijkstra
Bellman-Ford
write()
Negative_Cycle
true
false
Output shortest path by Dijkstra and Bellman-Ford
Output shortest path by Dijkstra and "Negative loop detected!"
End

Discussion

Time Complexity

- Dijkstra：O(V^2)
- Bellman-Ford：O(V*E) = O(V^3)

Which is Better Algorithm in Which Condition

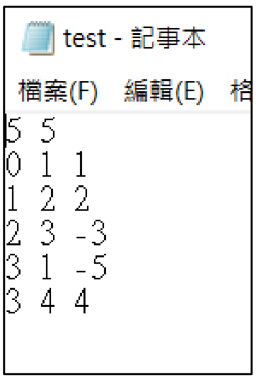
- Doesn’t Exist Negative Edges：Dijkstra 會比 Bellman-Ford 更有效率
- Exists Negative Edges：因為 Dijkstra 無法計算，因此 Bellman-Ford 會是唯一的選擇。

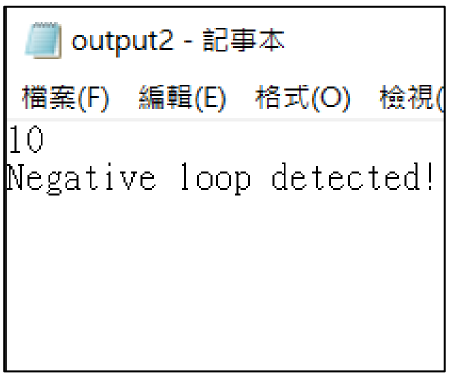
Optimize and Discover

- 在實作 Dijkstra 演算法時，利用 priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>>找未被造訪過且離出發點最近的點，便不用再利用窮舉的方式找出距離最小值，雖然時間複雜度一樣是 O(V^2)，但效率更好。

Results

- Test Input & Test Output：輸出與範例結果吻合


test - 記事本
檔案(F) 編輯(E) 格
5 5
0 1 1
1 2 2
2 3 -3
3 1 -5
3 4 4


output2 - 記事本
檔案(F) 編輯(E) 格式(O) 檢視(
10
Negative loop detected!