

我主要將此程式分成兩個小題：有向與無向

先拆解此題目：對於無向圖而言，此題目可以轉換為『maximum span tree』，因此使用 Kruskal's MST Algorithm，且因為此題目的 w 限定在 $-100 \sim 100$ 之間，因此可以使用 bucket sort 加速排序，且無向圖的邊數目可達到 20000000 之多，因此透過 bucket 避免過多的排序
爾後透過 union 方法去建構起 MST，最後即可得到所求
union 的方法是透過將 v-sets 的 id 對應起來並在每次 set union 時確定是否所有 v 都在 set 之內
(透過 `vector.size() == vertex_size` 判定)

資料結構上將 G, v, e 分開，並且在 G 裡面建構桶以及 edge/vertex 的對應表，並給一個 sets 的 `vector<vector<int>>` 儲存 sets 的資訊

而無向圖無論是否有權重都可以使用此方法解決此問題。

共用資料結構：

vertex 內部存有

id, set, bfs_color, bfs_d, dfs_t, in_edge array and out_edge array

因為一開始資料結構就選用 list 而非 array 的方式存儲，因此在 call mem 上面可能會稍慢，但好處是記憶體空間較為連續，搜索上速度應該差不多

edge 內有

$w, v1, v2, used, index$
(edge from $v1$ to $v2$)

透過將 edge 丟進丟出 vertex 快速建構並操作 G

edgeset 內部有

total_weight 以及 `vector<edge>`

作為快速操作使用（一開始有設計要遞迴多層，但後來發現效益極差，因此後來選擇遞迴兩層後，此功能略顯雞肋）

有向圖就顯的非常麻煩

資料結構相較於無向圖新增 unused_bucket（關鍵資料），temp_bucket, edge size, vertex size

解決有向圖的想法，我是先將它當成無向圖的解決方法建起 span tree 但是在發現

`v1.set == v2.set` 時不直接拋棄邊，而是使用 bfs 確認若加進這個邊是否會有環（採用 bfs 是因為其速度較為穩定，且比 topology sort 快，如果環的長度與圖的深度不對稱，也有較好的計算穩定性）

倘若有環就拋棄這個邊，無環則加入，是一個 pre-process 的 greedy 方法。

0 邊不放入，放入的話會有怪問題（0 邊是在後面的刪減可以進行嘗試）

透過此方法建立起的有向無環圖並非最小，因此需要透過加邊與拆邊進行。

使用迴圈從 unused_bucket (inside of this is edges) 挑選兩個不重複的邊（由大到小）加入圖，並透過 topological sort get cycle -> 逐步拆掉 source 以及 sink 的所有 edge 直到無法拆裝為止，剩下的邊即為 cycles，此方法可以有效的拆出所有環，但是速度較 bfs 慢。

透過此方法得到一個新的 Graph g ，並將該 g 內的所有環刪減邊，此嘗試刪減邊方法可快速將可能的邊用 bfs_u/bfs_d 測試。

倘若碰到有成功的邊/邊組且其權重（在刪減邊的階段就已經篩選過了）小於加入的邊，則將其更新到外圍的 G 並且將迴圈的 bucket 更新，並持續執行，直到 59s.

透過 cut edge and insert edge 方法對 G 快速加減邊並查找，並且透過排序以及 goto 跳出迴圈，將不可能的 edge set 排除，已達到加速運算。

因為是對 g 在環裡面挑選邊由小到大去減除，因此可以有效避免解到解後重複選到同個環

由於在試驗上，倘若 edge 數目太大，則有可能跑不完切一條邊的試驗，因此使用依序切邊以及根據整個 g -edge 以及 G 的大小去測試要放多少數目的 cut edge set 去跑。

透過調參數方法找到可能適合的遞迴深度後，在依序動態調整，得到區域的最佳解（且避免耗費過多時間）。