Algorithm Programming Assignment #3 **Report**

B07901021 潘世軒

1. **Data structure**
2. **class** Graph
3. {
4. **public**:
5. Graph();
6. Graph(**char**, **int**);
7. **void** E\_construct(**int**, **int**, **int**);
8. **void** test();
9. **void** cyclebreak(fstream&);
10. **void** sort\_E();
11. **void** MST\_Kruskal();
12. **void** dir\_find\_delete\_edge(vector<Edge>&);
13. **void** add\_edge\_back(vector<Edge>&);
14. **void** output\_u(fstream&);
15. **void** output\_d(fstream&);
16. **private**:
17. **void** Makeset(**int**);
18. **void** Union(**int**, **int**);
19. **int** Findset(**int**);
20. **void** Link(**int**, **int**);
21. **void** MergeSort(vector<Edge>&);
22. **void** MergeSortSubVector(vector<Edge>&, **int**, **int**);
23. **void** Merge(vector<Edge>&, **int**, **int**, **int**, **int**);
25. **bool** DFS\_cycle\_detecting(Edge&);
26. **bool** DFS\_visit(**int**, **int**, vector<**char**>&);
28. **void** DFS\_cycle\_detecting\_check();
29. **void** DFS\_visit\_check(**int**, vector<**char**>&);
31. **char** type;
32. **int** n;
33. // weight[i][j] means the weight of edge(i,j)(-100~100)
34. // if no edges between (i,j), weight[i][j] = 666
35. vector<vector<**int**> > weight;
36. vector<vector<**int**> > A;
37. vector<Edge> E;
38. vector<vector<**int**> > adj;
39. vector<Vertex> V;
40. };

使用一個叫做Graph的資料結構，裏頭包含vertex, edge，type 是用來表達這個graph是undirected 還是directed的，n則是用來表達vertice數目(有時候直接用n取代V.size比較方便)，weight則是用來表達edge weight(如comment)，在我們進行cycle breaking後，會將刪除某些edge的結果儲存在A，最後output的時候將A和weight進行比對來輸出刪掉的邊，最後又多創了一個adj的二維陣列，以便跑DFS。Graph裡的一大堆函式會在下方的Algorithm中解釋。

1. **Algorithm**

在本次作業中，我將幾乎所有的function都定義在Graph裡面，一方面是我如果我要在別的地方使用這個演算法，可以直接呼叫graph的member function，感覺比較直覺，一方面是這樣是直接改graph裡的各種屬性，不需要用外部function寫call by referenc。，根據輸入建立好Graph後，直接跑G.cyclebreak(fout) 開始進行拆解cycle。

1. **void** Graph::cyclebreak(fstream& fout){
2. **if**(type == 'u'){
3. // undirected:
4. // (1) find the maximum spanning tree
5. // (2) compare the MST to origin edge and find the deleted edge
6. MST\_Kruskal();
7. output\_u(fout);
8. }
9. **else** **if**(type == 'd'){
10. // directed:
11. // (1) treat all edges as undirected edges and find the minimum spanning tree
12. // (2) compare the MST to origin edge and find the deleted
13. // (3) trying add back edges with "positive weight" in decreasing order
14. // (4) run DFS to check whether the edge added will cause cycle in the Graph
15. // (5) if not, add it back
16. // (6) compare the final result to origin edge and find the deleted edge
18. MST\_Kruskal();
19. vector<Edge> d;
20. dir\_find\_delete\_edge(d);
21. add\_edge\_back(d);
22. output\_d(fout);
23. }
24. **else**{
25. cout << "type error" << endl;
26. }
27. }
28. Undirected graph

Undirected graph的部分，要求刪掉的edge weight總和越小越好，並且所有頂點都要被連接，因此我將課本中的MST-Kruskal稍微修改，改成找Maximum spanning tree，這樣代表要刪掉的的邊weight總和會是min，改動的地方只有將edge根據weight sort時的順序改為由大到小(先嘗試把大的加進MST中以找到Maximum spanning tree)，將要刪掉的邊記在上述的A矩陣裡再和weight比對就可以找到哪些邊是要被刪掉的。

1. Directed graph

一開始想要使用的方法是跑DFS去detect cycle，並刪掉這個cycle中最小weight的edge，但是後來覺得這個方法似乎有點不切實際，因為也許會找到很多cycle，每個cycle又要繞整圈去找，感覺比較麻煩，因此不採用這個方法。

後來選用如上方code所示的方法，先將所有的directed edge都當作是無向的去跑MST (max)，跑完後的結果，可以確保剩下來的edge絕對不會造成cycle，接下來將這些被刪掉的edge，由大排到小開始嘗試一一把weight>0的edge加進去，加進一條邊後，check他加進去會不會造成新的cycle (偵測辦法：若(u, v)加入後會造成cycle，這個cycle一定會包含(u, v)，因此從v開始跑DFS，如果跑得到u的話就代表有cycle存在！)，如果不會的話就代表我可以安心的把這個edge加進去(並更新上述A矩陣以及adj矩陣)，當把所有positive edge都加進去後，比對A矩陣和weight就可以寫output了。

1. **Findings**

這次作業比較困難的部分處理有向圖的部分，但我認為只要想到一個合理的演算法就可以交了，因為其實我覺得最後刪掉的總weight，其實和刪除或者加入edge的順序有關係，我嘗試過將上方紅字的這個排序改成quicksort(unstable)，讓同樣weight的東西先後順序可能改變，導致directed case中有些會變好，有些結果則會變差，但很難去評斷這個順序到底怎麼排比較好，因此最後就隨便取一種sort方法來做，並祈禱測資對我有利。

我認為另一個難題是這個graph到底要怎麼建立，課本中的演算法多半寫的很簡單，看看就過去了，但實際上要寫成code時卻會先卡一陣子這些資料結構到底要怎麼實現，我覺得我這次寫的graph的coding沒有很漂亮，因為有些private member其實是可以移除的，但是我後來寫法是需要什麼就把它加進去(比方說DFS需要adj matrix，就加，MST需要vertex.rank, p 這些東西，就寫一個class vertex，加入graph中)，但感覺如果客家一點的話應該是有很多東西可以捨去的，總之最後有算出答案了就先維持這樣吧XD，有空在來修的好看一點。

最後，因為這次作業比較難用肉眼驗證正確性，因此我自己大爆寫了一個確定答案是否正確的cpp檔，讀入原本的input file，以及算出來的output file，可以確認是否有cycle，是否所有vertex都走的到，以及是否誤刪根本沒有的邊，也有把這個check 的檔案傳給幾個同學並請他們試用看看有沒有bug，但目前還不知道這個checker到底會不會有問題XD，只能說我跑的結果丟去checker是沒有問題的。為了展現誠意，把checker丟在後面給看報告的人過目(其實可以看前面註解就好)，我覺得我在做的事情很符合工程師該做的事情，設計了一個東西不知道其對或錯，因此再設計一個東西來驗證其正確性，感覺蠻酷的！

1. // this is the checker for PA3, NTUEE, algorithm, 2020 fall
2. // produced by b07901021 SHIH-HSUAN PAN in 2020/12/20
3. // compile:    g++ correct.cpp -o correct
4. // run:        ./correct <original\_input\_file> <output\_file\_you\_produced>
5. //
6. // (1) the program will first read the original input file and construct graph
7. //     (only consider the edge without considering the weight)
8. // (2) then the output file you produced will be readed
9. // (3) according to your output file, some edges in the graph will be removed
10. //     also, the program will check whether the edge you removed is in the original input file
11. //     however, the order of undirected edges doesnt follow the order in the original input file
12. // (4) then the program will run DFS-smiliary algorithm to check the existence of cycles
13. // (5) treating all directed edges as undirected, then start traverse all vertices to see
14. //     whether all vertices are weakly connected
15. //
16. // summary:
17. // the program will check
18. // (1) whether you delete some edges not in the original input file
19. // (2) whether all vertices are weakly connected
20. // (3) whether there is any cycle in your graph aftering removing edges you specified
21. //
22. // if there is any problem about this checker, please send email to me to discuss about it
23. // email: b07901021@ntu.edu.tw
25. #include <cstring>
26. #include <fstream>
27. #include <vector>
28. #include <iostream>
29. #include <algorithm>
31. **using** **namespace** std;
33. **class** Graph
34. {
35. **public**:
36. Graph();
37. Graph(**char** , **int**);
38. **void** E\_construct(**int**, **int**);
39. **void** E\_delete(**int**, **int**);
40. **void** DFS\_cycle\_detecting\_check();
41. **void** DFS\_visit\_check(**int**, vector<**char**>&, vector<**int**>&);
42. **void** weakly\_connected\_check();
43. **void** traverse(**int**, vector<**char**>&, vector<vector<**int**> >&);
44. **private**:
45. **char** type;
46. **int** n;
47. vector<vector<**int**> > adj;
48. };
50. Graph::Graph(**char** t, **int** num){
51. type = t;
52. n = num;
53. **for**(**int** i = 0; i < n; i++){
54. vector<**int**> blank;
55. adj.push\_back(blank);
56. }
57. }
58. **void** Graph::E\_construct(**int** u, **int** v){
59. adj[u].push\_back(v);
60. **if**(type == 'u') adj[v].push\_back(u);
61. }
63. **void** Graph::E\_delete(**int** u, **int** v){
64. **if**(type == 'd'){
65. cout << "deleting edge(" << u << ", " << v << ")" << endl;
66. vector<**int**>::iterator it = find (adj[u].begin(), adj[u].end(), v);
67. **if** (it != adj[u].end()) adj[u].erase(it);
68. **else** cout << "wrong!! your output file deletes edge not in original input file" << endl;
69. }
70. **else**{
71. vector<**int**>::iterator it1 = find (adj[u].begin(), adj[u].end(), v);
72. vector<**int**>::iterator it2 = find (adj[v].begin(), adj[v].end(), u);
73. **if** (it1 != adj[u].end() && it2 != adj[v].end()) {
74. adj[u].erase(it1);
75. adj[v].erase(it2);
76. // cout << "deleting edge(" << u << ", " << v << ")" << endl;
77. }
78. **else**{
79. cout << "wrong!! your output file deletes edge not in original input file" << endl;
80. }
81. }
82. }
84. **void** Graph::DFS\_cycle\_detecting\_check(){
85. vector<**char**> color(n, 'w');
86. vector<**int**> pi(n, -1);
87. **for**(**int** i = 0; i < n; i++){
88. **if**(color[i] == 'w')
89. DFS\_visit\_check(i, color, pi);
90. }
91. }
92. **void** Graph::DFS\_visit\_check(**int** u, vector<**char**>& color, vector<**int**>& pi){
93. color[u] = 'g';
94. **for**(**int** i = 0; i < adj[u].size(); i++){
95. **int** v = adj[u][i];
96. **if**(color[v] == 'w'){
97. pi[v] = u;
98. DFS\_visit\_check(v, color, pi);
99. }
100. **else** **if**(color[v] == 'g' && type == 'd'){
101. cout << "wrong!! there are cycles in your graph" << endl;
102. }
103. **else** **if**(color[v] == 'g' && type == 'u'){
104. **if**(v != pi[u])
105. cout << "wrong!! there are cycles in your graph" << endl;
106. }
107. }
108. color[u] = 'b';
109. }
110. **void** Graph::weakly\_connected\_check(){
111. vector<vector<**int**> > adj\_matrix;
112. vector<**char**> color(n, 'w');
113. **for**(**int** i = 0; i < n; i++){
114. vector<**int**> temp(n, 0);
115. adj\_matrix.push\_back(temp);
116. }
117. **for**(**int** i = 0; i < n; i++){
118. **for**(**int** j = 0; j < adj[i].size(); j++){
119. **int** u = i;
120. **int** v = adj[i][j];
121. adj\_matrix[u][v] = 1;
122. adj\_matrix[v][u] = 1;
123. }
124. }
125. traverse(0, color, adj\_matrix);
126. **for**(**int** i = 0; i < n; i++){
127. **if**(color[i] == 'w'){
128. cout << "wrong!! all vertices are not weakly connected" << endl;
129. **break**;
130. }
131. }
133. }
134. **void** Graph::traverse(**int** u, vector<**char**>& color, vector<vector<**int**> >& m){
135. color[u] = 'g';
136. **for**(**int** i = 0; i < m[u].size(); i++){
137. **if**(m[u][i] == 1){
138. **int** v = i;
139. **if**(color[v] == 'w'){
140. traverse(v, color, m);
141. }
142. }
143. }
144. color[u] = 'b';
145. }
147. **int** main(**int** argc, **char**\* argv[])
148. {
149. **if**(argc != 3) {
150. cout << "wrong arguments" << endl;
151. **return** 0;
152. }
154. //////////// read the input file /////////////
156. cout << "reading..." << endl;
157. **char** buffer[200];
158. fstream fin(argv[1]);
159. fstream fin\_d(argv[2]);
160. **char** type; // 'u' means undirected edge
161. // 'd' means directed edge
162. fin >> type;
163. **int** m, n; // n: total number of vertices
164. // m: total number of edges
165. fin >> n >> m;
167. vector<**int**> data;
168. **int** num;
169. **while** (fin >> num)
170. data.push\_back(num); // data[3i] will be the start point of an edge.
171. // data[3i+1] will be the end point of an edge.
172. // data[3i+2] will be the weight.
174. cout << "processing original input file..." << endl;
175. Graph G(type, n);
176. **for**(**int** i = 0; i < m; i++){
177. **int** u = data[3\*i];
178. **int** v = data[3\*i+1];
179. **int** w = data[3\*i+2];
180. G.E\_construct(u, v);
181. }
183. cout << "processing output file..." << endl;
184. vector<**int**> data\_d;
185. **int** num\_d, total;
186. fin\_d >> total;
187. cout << "sum of deleted edge weight = " << total << endl;
188. **while** (fin\_d >> num\_d){
189. data\_d.push\_back(num\_d);
190. }
191. **for**(**int** i = 0; i < data\_d.size(); i+=3){
192. **int** u = data\_d[i];
193. **int** v = data\_d[i+1];
194. **int** w = data\_d[i+2];
195. G.E\_delete(u, v);
196. }
198. cout << "checking cycle existance" << endl;
199. G.DFS\_cycle\_detecting\_check();
201. cout << "checking connected" << endl;
202. G.weakly\_connected\_check();
204. cout << "checking finished" << endl;
205. }