This Graph cant be augmented further .

Thus the solution obtained is U1-I1 and U2-I2.

Further the input graph can be changed to obtain more solutions.

Code in Cpp:-

|  |
| --- |
| #include <bits/stdc++.h>  using namespace std;  const int N=1e2+5;  vector<int> parent(N);  vector<int> visited(N);  vector<vector<pair<int,int> > > adjacencyList(N);  vector<vector<pair<int,int> > > residualAdjacencyList(N);  vector<vector<int> > input(N);  vector<vector<pair<int,int> > > solution;  int totalAssignment=10000;  int finalEdge=10000;  bool comparTor(vector<vector<pair<int,int> >> ::iterator iteraTor1, vector<vector<pair<int,int>> > ::iterator iteraTor2)  {    vector<pair<int,int> > vector1=\*iteraTor1;  vector<pair<int,int> >vector2=\*iteraTor2;  if(vector1.size()!=vector2.size())  return false;  for(int onesolutionVariable =0; onesolutionVariable<vector1.size(); onesolutionVariable++)  {  if(vector1[onesolutionVariable].first==vector2[onesolutionVariable].first and vector1[onesolutionVariable].second == vector2[onesolutionVariable].second)  continue;  return false;  }  return true;  }  void removeDuplicates()  {  for(vector<vector<pair<int,int >>> :: iterator iteraTor =solution.begin();iteraTor != solution.end(); iteraTor++)  {  for(vector<vector<pair<int,int> >> :: iterator itr=solution.begin();itr!=iteraTor;itr++ )  {  if(comparTor(iteraTor,itr))  {  solution.erase(iteraTor);  iteraTor--;  break;  }  }  }  }  void reverseSort()  {  for (int iteraingVariable = solution.size() - 2; iteraingVariable >= 0; --iteraingVariable)  {  vector<pair<int,int> > oneSolution=solution[iteraingVariable];  int iteratingVariable2=iteraingVariable;  while (iteratingVariable2 < solution.size() - 1 && oneSolution.size() > solution[iteratingVariable2].size())  {  solution[iteratingVariable2]=solution[iteratingVariable2+1];  iteratingVariable2++;  }  solution[iteratingVariable2]=oneSolution;  }  }  void printSolution()  {    reverseSort();  for(int iteratingVariable=0;iteratingVariable<min(100,(int)solution.size());iteratingVariable++)  {  vector<pair<int,int > > oneSolution = solution[iteratingVariable];  for(pair<int,int> onePair : oneSolution)  {  cout<<onePair.first<<"th user is assigned "<<onePair.second<<"th interval"<<endl;  }  cout<<endl;  }  }  int bfsForFindingParent(int sourceVertex, int destinationVertex)  { //clearing the visited array  for(int node=0;node<N;node++)  visited[node]=0;  visited[sourceVertex]=1;  parent[sourceVertex]=-1;  queue<int> pendingNodes;  pendingNodes.push(sourceVertex);  while(!pendingNodes.empty())  {  int currentVertex=pendingNodes.front();  pendingNodes.pop();  for(pair<int,int> adjacentEdge : residualAdjacencyList[currentVertex])  {  int adjacentVertex=adjacentEdge.first;  int weightOfEdge=adjacentEdge.second;  if(visited[adjacentVertex]==0 and weightOfEdge>0)  {  pendingNodes.push(adjacentVertex);  visited[adjacentVertex]=1;  parent[adjacentVertex]=currentVertex;  }  }  }  return visited[destinationVertex];  }  void findingMaximumMapping(int noUser, int noInterval)  {  totalAssignment=0;  while( bfsForFindingParent(0,noUser+noInterval+2))  {  int possibleAssignment=INT\_MAX;  for(int node=noUser+noInterval+2;parent[node]!=-1;node=parent[node])  {  for(auto edge : residualAdjacencyList[parent[node]])  {  if(edge.first==node)  {  possibleAssignment=min(possibleAssignment,edge.second);  }  }  }  for(int node=noUser+noInterval+2;parent[node]!=-1 ;node=parent[node])  {    for(pair<int,int>& edge : residualAdjacencyList[parent[node]])  {  if(edge.first==node)  {  edge.second-=possibleAssignment;    }  }    for(pair<int,int>& edge : residualAdjacencyList[node])  {  if(edge.first==parent[node])  edge.second+=possibleAssignment;  }  }  totalAssignment+=possibleAssignment;    }  }  void initialisingAdjacencyList(int noUser ,int noInterval)  {  for(int node=0;node<=noUser+noInterval+2;node++)  {  residualAdjacencyList[node].clear();  adjacencyList[node].clear();  }  for(int user=1;user<=noUser;user++)  {  adjacencyList[0].push\_back(make\_pair(user,1));  residualAdjacencyList[0].push\_back(make\_pair(user,1));  //reverse edge with zero weight  adjacencyList[user].push\_back(make\_pair(0,0));  residualAdjacencyList[user].push\_back(make\_pair(0,0));  }  for(int interval =noUser+1;interval<=noUser+noInterval;interval++)  {  adjacencyList[interval].push\_back(make\_pair(noUser+noInterval+1,1));  residualAdjacencyList[interval].push\_back(make\_pair(noUser+noInterval+1,1));  adjacencyList[noUser+noInterval+1].push\_back(make\_pair(interval,0));  residualAdjacencyList[noUser+noInterval+1].push\_back(make\_pair(interval,0));  }  residualAdjacencyList[noUser+noInterval+1].push\_back(make\_pair(noUser+noInterval+2,finalEdge));  residualAdjacencyList[noUser+noInterval+2].push\_back(make\_pair(noUser+noInterval+1,0));  adjacencyList[noUser+noInterval+1].push\_back(make\_pair(noUser+noInterval+2,finalEdge));  adjacencyList[noUser+noInterval+2].push\_back(make\_pair(noUser+noInterval+1,0));    }  void userIntervalMapping(int noUser, int noInterval)  {    initialisingAdjacencyList(noUser,noInterval);    for(int user=1;user<=noUser;user++)  {  for(int interval : input[user])  {  adjacencyList[user].push\_back(make\_pair(interval+noUser,1));  residualAdjacencyList[user].push\_back(make\_pair(interval+noUser,1));  // //adding reverse edge  adjacencyList[interval+noUser].push\_back(make\_pair(user,0));  residualAdjacencyList[interval+noUser].push\_back(make\_pair(user,0));  }  }    findingMaximumMapping(noUser,noInterval);  vector<pair<int,int> > oneSolution;  for(int user=1;user<=noUser;user++)  {  int \* freqWeight= new int[105]{};  for(pair<int,int> edge : adjacencyList[user]) freqWeight[edge.first]+=edge.second;  for(pair<int,int> edge : residualAdjacencyList[user]) freqWeight[edge.first]-=edge.second;  for(int frequency =noUser+1;frequency<=noUser+noInterval;frequency++)  {  if(freqWeight[frequency]==1)  {  oneSolution.push\_back(make\_pair(user,frequency-noUser));  }  }  }  sort(oneSolution.begin(),oneSolution.end());  solution.push\_back(oneSolution);  removeDuplicates();  }  void userReverseIntervalMapping(int noUser, int noInterval)  {  initialisingAdjacencyList(noUser,noInterval);  for(int user=1;user<=noUser;user++)  {  for(int interval : input[user])  {  adjacencyList[noUser+1-user].push\_back(make\_pair(interval+noUser,1));  residualAdjacencyList[noUser+1-user].push\_back(make\_pair(interval+noUser,1));  // //adding reverse edge  adjacencyList[interval+noUser].push\_back(make\_pair(noUser-user+1,0));  residualAdjacencyList[interval+noUser].push\_back(make\_pair(noUser-user+1,0));  }  }    findingMaximumMapping(noUser,noInterval);  vector<pair<int,int> > oneSolution;  for(int user=1;user<=noUser;user++)  {  int \* freqWeight= new int[105]{};  for(pair<int,int> edge : adjacencyList[user]) freqWeight[edge.first]+=edge.second;  for(pair<int,int> edge : residualAdjacencyList[user]) freqWeight[edge.first]-=edge.second;  for(int frequency =noUser+1;frequency<=noUser+noInterval;frequency++)  {  if(freqWeight[frequency]==1)  {  oneSolution.push\_back(make\_pair(noUser-user+1,frequency-noUser));  }  }  }  sort(oneSolution.begin(),oneSolution.end());  solution.push\_back(oneSolution);  removeDuplicates();  }  void userIntervalReverseMapping(int noUser, int noInterval)  {  initialisingAdjacencyList(noUser,noInterval);  for(int user=1;user<=noUser;user++)  {  for(int interval : input[user])  {  adjacencyList[user].push\_back(make\_pair(noInterval-interval+noUser+1,1));  residualAdjacencyList[user].push\_back(make\_pair(noInterval-interval+noUser+1,1));  // //adding reverse edge  adjacencyList[noInterval-interval+noUser+1].push\_back(make\_pair(user,0));  residualAdjacencyList[noInterval-interval+noUser+1].push\_back(make\_pair(user,0));  }  }      findingMaximumMapping(noUser,noInterval);  vector<pair<int,int> > oneSolution;  for(int user=1;user<=noUser;user++)  {  int \* freqWeight= new int[105]{};  for(pair<int,int> edge : adjacencyList[user]) freqWeight[edge.first]+=edge.second;  for(pair<int,int> edge : residualAdjacencyList[user]) freqWeight[edge.first]-=edge.second;  for(int frequency =noUser+1;frequency<=noUser+noInterval;frequency++)  {  if(freqWeight[frequency]==1)  {  oneSolution.push\_back(make\_pair(user,noInterval+1-(frequency-noUser)));    }  }  }  sort(oneSolution.begin(),oneSolution.end());  solution.push\_back(oneSolution);  removeDuplicates();  }  void userReverseIntervalReverseMapping(int noUser, int noInterval)  {  initialisingAdjacencyList(noUser,noInterval);  for(int user=1;user<=noUser;user++)  {  for(int interval : input[user])  {  adjacencyList[noUser-user+1].push\_back(make\_pair(noInterval+1-interval+noUser,1));  residualAdjacencyList[noUser-user+1].push\_back(make\_pair(noInterval+1-interval+noUser,1));  // //adding reverse edge  adjacencyList[noInterval+1-interval+noUser].push\_back(make\_pair(noUser-user+1,0));  residualAdjacencyList[noInterval-interval+noUser+1].push\_back(make\_pair(noUser-user+1,0));  }  }  findingMaximumMapping(noUser,noInterval);  vector<pair<int,int> > oneSolution;    for(int user=1;user<=noUser;user++)  {  int \* freqWeight= new int[105]{};  for(pair<int,int> edge : adjacencyList[user]) freqWeight[edge.first]+=edge.second;  for(pair<int,int> edge : residualAdjacencyList[user]) freqWeight[edge.first]-=edge.second;  for(int frequency =noUser+1;frequency<=noUser+noInterval;frequency++)  {  if(freqWeight[frequency]==1)  {  oneSolution.push\_back(make\_pair(noUser+1-user,noInterval+1-(frequency-noUser)));    }  }  }  sort(oneSolution.begin(),oneSolution.end());  solution.push\_back(oneSolution);  removeDuplicates();  }  void userToIntervalMapping(int noUser ,int noInterval)  { userIntervalMapping(noUser,noInterval);  userReverseIntervalMapping(noUser,noInterval);  userIntervalReverseMapping(noUser,noInterval);  userReverseIntervalReverseMapping(noUser,noInterval);  finalEdge=totalAssignment-1;  }  int main()  {  #ifndef ONLINE\_JUDGE    freopen("input.txt", "r", stdin);    freopen("output.txt", "w", stdout);    #endif  time\_t start, end;  time(&start);  // enter the number of users  int noUser;  cin >> noUser;  //enter the number of intervals  int noInterval;  cin >> noInterval;  //taking input    for(int user=1;user<=noUser;user++)  { //enter number of intervals for ith user  int noOfIntervals;  cin >> noOfIntervals;  while(noOfIntervals--)  {  int favourableInterval;  cin >> favourableInterval;  input[user].push\_back(favourableInterval);    }  }  while(solution.size()<4 and totalAssignment>0)  userToIntervalMapping(noUser,noInterval);    printSolution();    time(&end);  double time\_taken = double(end - start);    cout<<"calculation of total time taken by program " << fixed << time\_taken << setprecision(5);  cout << " sec " << endl;    return 0;  } |

Pros of Solution:

1)Solution is very efficient .Time complexity of solution is O(noUsers\*(noINterval)^2).It can produce results of noUser<=500 within 1 sec.

Cons of Solution

1) For finding more than 1 solution we have to manually change the graph.