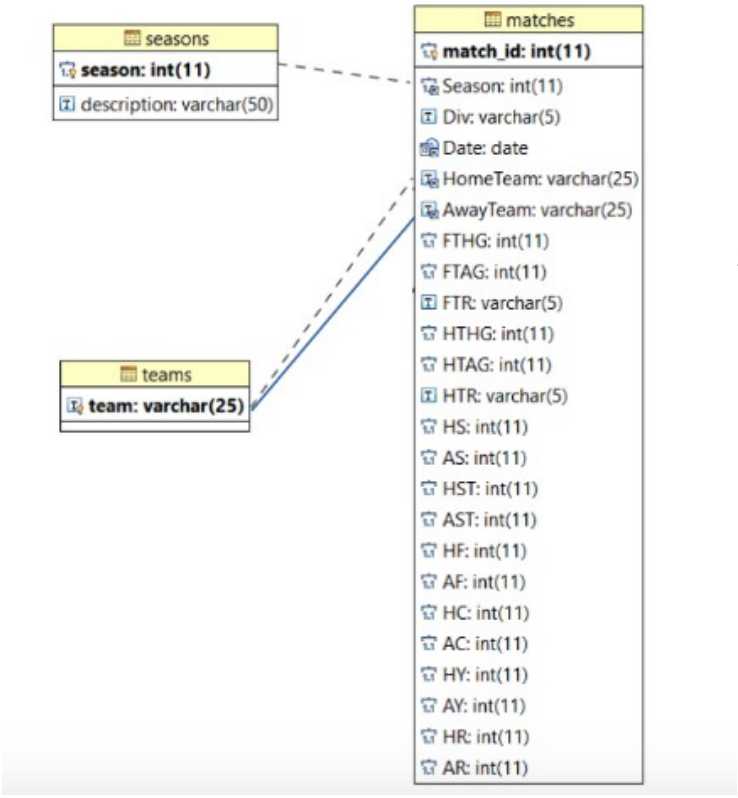
SERIE A




```
  private int id ;
  private Season season ;
  private String div ;
  private LocalDate date ;
  private Team homeTeam ;
  private Team awayTeam ;
  private int fthg ; // full time home goals
  private int ftag ; // full time away goals
  private String ftr ; // full time result (H, A, D)
  HTHG = Half Time(*) Home Team Goals                    (*)
tranne quanto la partita è assegnata a tavolino
          HTAG = Half Time(*) Away Team Goals
          HTR = Half Time(*) Result (H=Home Win,
                              D=Draw, A=Away Win)
          HS = Home Team Shots
           AS = Away Team Shots
           HST = Home Team Shots on Target
          AST = Away Team Shots on Target
          HHW = Home Team Hit Woodwork
          AHW = Away Team Hit Woodwork
          HC = Home Team Corners
          AC = Away Team Corners
          HF = Home Team Fouls Committed
          AF = Away Team Fouls Committed
          HO = Home Team Offsides
          AO = Away Team Offsides
          AR = Away Team Red Cards
          HY = Home Team Yellow Cards
          AY = Away Team Yellow Cards
          HR = Home Team Red Cards
```

```
*/
```

```
//---------------------------------------------------------
//  ---------------->   MAIN  <---------------
//---------------------------------------------------------

        SerieAController controller = loader.getController() ;
Model model=new Model();
controller.setModel(model);

//---------------------------------------------------------
// ------------------>  CONTROLLER  <---------------
//---------------------------------------------------------

Model model;

public void setModel(Model model) {

this.model=model;

//devo prendere le stagioni e metterle nella combobox e le inizializzo a 0
this.boxSeason.getItems().addAll(model.getSeasons());
this.boxSeason.setValue(this.boxSeason.getItems().get(0));

//devo prendere le Team e metterle nella combobox e le inizializzo a 0
this.boxTeam.getItems().addAll(model.getAllTeams());
this.boxTeam.setValue(this.boxTeam.getItems().get(0));
}
//---------------------------------------------------------
// ------------------>  DBCONNECT  <---------------
//---------------------------------------------------------

togliere password=root
```

```
//-------------------------------------------------------
// ------------------>  DAO  <---------------
//-------------------------------------------------------

/**
* @return VERICI GRAFO per SEASON, GOAL, TEAM
*/
 public List<Integer> getTotGOAL () {
 public List<Integer> getYears () {

 public Map<Integer,Season> MapSeasons() {
 public List<Season> listSeasons() {

 public Map<String,Team> MapTeams() {
 public List<Team> listTeams() {

 public Map<String,Team> MapTeamsForSeason(Season s) {
 public List<Team> listTeamsforSeason(Season s) {

 public List<Match> getAllMatches( Map<String, Team> mapSquadre,Map<Integer, Season> mapSeason) {
 public List<Match> getMatchesForSeason(Season stagione, Map<String, Team> mapSquadre) {

/**
* @return List MATCH
*/
 public List<Match> nSquadreNelleSTAGIONI(Integer s1,Integer s2,Map<String,Team> mapSquadre,Map<Integer,
 Season> mapSeason) {
 public List<Match> getPartiteByHomeTeam(Season stagione,Team homeTeam, Map<String, Team> mapSquadre) {   }
 public List<Match> getPartiteByAwayTeam(Season stagione,Team awayTeam, Map<String, Team> mapSquadre) { }
 public List<Match> getPartiteBySeason(Season stagione, Map<String, Team> mapSquadre) {    }
 public List<Match> getPartiteOVERBySeason(Season stagione, Map<String, Team> mapSquadre) {       }
 public List<Match> getPartiteByTeam(Season stagione,Team team, Map<String, Team> mapSquadre) {   }
 public List<Match> getMatchFinitiXaY_oopure_YaX( Map<String, Team> mapSquadre,Map<Integer, Season>
 mapSeason,Integer i,Integer i2) {
 public List<String> Squadre2Stagioni(Integer s,Integer s1) {
/**
* @return partite giocate finite con quel risultato dando goal cas e goal trasferta
*/
 public Integer nPartiteFiniteXaY(Integer i,Integer i2) {
 public Integer nPartiteFiniteXaY_InSTAGIONE(Integer i,Integer i2,Season s) {

/* @return  GOAL  tot, diff, casa, trasferta nelle due stagioni      */
 public Integer GoalTotali2Stagioni(Integer s,Integer s1) {

/*       @return numero partite o stagioni che hanno giocato contro  */
 public Integer nPartiteGiocateContro(Team  a,Team  a1,Map<String,Team> teams) {

/*       @return TOTgoalScontriDiretti------SUM diff reti----->TOTgoal team in casa----->TOTgoal team in trasferta
  */
 public Integer TOTgoalScontriDiretti(Team  a,Team  a1,Map<String,Team> teams) {

/**
 *  DATA LA STAGIONE
 * @return diff reti----->Goal in casa----->Goal in trasferta---->tot goal
 */
 public Integer diffRETI(Team  a,Team  a1,Season s,Map<String,Team> teams) {

/**
 * ALTRI PESI GRAFO
 */
 public Integer listNumeroPartiteVinteDaTeam(Season stagione,Team team, Map<String, Team> mapSquadre) {  }
 public Integer GOALTeamHomeInStagione(Season stagione,Team team, Map<String, Team> mapSquadre) {  }
/**
 * ALTRE MAPPE
 */
 public Map<Team, Integer> ListaGoalHomeInSeasonDESC(Season stagione, Map<String, Team> mapSquadre) {     }
 public Map<Team, Integer> ListaGoalAwayInSeasonDESC(Season stagione, Map<String, Team> mapSquadre) {     }

//-------------------------------------------------------
// ------------------>  GRAFI  <---------------
//-------------------------------------------------------

public void creaGrafoTEAM() {
 String s="";
 if (graphTEAM==null) graphTEAM=
 new SimpleWeightedGraph<Team,DefaultWeightedEdge>(DefaultWeightedEdge.class);

 Graphs.addAllVertices(graphTEAM, this.getAllTeams());//aggiungi vertici
 System.out.println("Grafo creato: " + graphTEAM.vertexSet().size() + " nodi");
```

```
 Map<Integer,Season>mapSEASON= dao.MapSeasons();
 Map<String,Team>mapTEAM= dao.MapTeams();
 //aggiungi archi
 for(Match mtemp: dao.getAllMatches(mapTeam, mapSeason)){
 Team home=mtemp.getHomeTeam();
 Team away=mtemp.getAwayTeam();
 if(home!=null && away!=null && !home.equals(away) ){
 //CALCOLO PESO n partite giocate
 Integer peso=  dao.nPartiteGiocateContro(home, away, mapTeam);
 //IMPOSTO ARCHI
 Graphs.addEdgeWithVertices(graphTEAM, home,away, peso);
 s+= home+" "+ away+" "+ peso +"\n";
 }
 }
 System.out.println("Grafo creato: " + graphTEAM.vertexSet().size() + " nodi, " + graphTEAM.edgeSet().size() + "
archi");
 System.out.println(s);
 }
public void creaGrafoGOAL() {
 String s="";
 if (graphGOAL==null) graphGOAL=
 new SimpleWeightedGraph<Integer,DefaultWeightedEdge>(DefaultWeightedEdge.class);
 Graphs.addAllVertices(graphGOAL, this.getTotGOAL());//aggiungi vertici
 Map<String,Team> mapTEAM =dao.MapTeams();
 Map<Integer,Season>mapSEASON= dao.MapSeasons();
 //aggiungi archi
 for(Match mtemp:dao.getAllMatches(mapTeam, mapSeason)){
 Integer home= mtemp.getFthg();
 Integer away= mtemp.getFtag();
 if(home!=null && away!=null && !home.equals(away) ){
 //CALCOLO PESO DIFFERENZA RETI
 Integer peso=  dao.nPartiteFiniteXaY(home, away);
 //IMPOSTO ARCHI

 Graphs.addEdgeWithVertices(graphGOAL, home, away,peso);
 s+= home+"  "+ away +" "+ peso+"\n";
 }
 }

 System.out.println("Grafo creato: " + graphGOAL.vertexSet().size() + " nodi, " +
 graphGOAL.edgeSet().size() + " archi");
 System.out.println(s);
 }


public void creaGrafoSEASON() {
 String s="";
 if (graphSeason==null)  graphSeason=new SimpleWeightedGraph<Integer,DefaultWeightedEdge>(DefaultWeightedEdge.class);
 Graphs.addAllVertices(graphSeason, dao.getYears());//aggiungi vertici
 System.out.println("Grafo creato: " + graphSeason.vertexSet().size() );

 //aggiungi archi

 for(Integer s1:graphSeason.vertexSet()){
 for(Integer s2:graphSeason.vertexSet()){
 if(s1!=null && s2!=null&& !s1.equals(s2) ){
 //IMPOSTO ARCHI
 //numero squadre presenti in entrambe le stagioni
 List<String> m=dao.Squadre2Stagioni(s1, s2);
 Double peso= (double) m.size();
 //goal
 Integer peso2= dao.GoalTotali2Stagioni(s1, s2);
 Graphs.addEdgeWithVertices(graphSeason,s1, s2, peso);
 s+= s1+" "+ s2+" "+peso+"\n";
 }
 }
 }

 System.out.println("Grafo creato: " + graphSeason.vertexSet().size() + " nodi, " +
 graphSeason.edgeSet().size() + " archi");
 System.out.println(graphSeason);
 System.out.println(s);
 }
```

```
//-------------------------------------------------------
// ----------------->  GET BEST/WORST  <---------------
//-------------------------------------------------------

 public Team getBestTeam() {
 Team best=null ;
 int max = Integer.MIN_VALUE ;
 for(Team d: this.graph.vertexSet()) {
 int peso = 0 ;
 //outgoingEdgesOf(d)) e incomingEdgesOf(d)) perchè è directed...se no solo outgoingEdgesOf(d))
 for(DefaultWeightedEdge e: graph.outgoingEdgesOf(d)) {      peso += graph.getEdgeWeight(e) ;}
 for(DefaultWeightedEdge e: graph.incomingEdgesOf(d)) {      peso -= graph.getEdgeWeight(e) ;      }
 if(peso>max) {
 max = peso ;
 best = d ;     }
 }      return best ;
 }

 public Team getWorstTeam() {
 Team loser=null ;
 int peggiore=Integer.MAX_VALUE;
 for(Team d: this.graph.vertexSet()) {
 int peso = 0 ;
 for(DefaultWeightedEdge e: graph.outgoingEdgesOf(d)) {      peso += graph.getEdgeWeight(e) ;}
 for(DefaultWeightedEdge e: graph.incomingEdgesOf(d)) {      peso -= graph.getEdgeWeight(e) ;}
 if(peso<peggiore) {
 peggiore = peso ;
 loser = d ;    }
 }      return loser ;
 }

//-------------------------------------------------------
// ------------------> VISITE  <----------------
//-------------------------------------------------------


1)  Escludendo gli aeroporti con zero rotte, determinare se nel grafo ottenuto è possibile da ogni aeroporto
 raggiungere ogni altro aeroporto.----->>> fortemente connesso?
2) lista degli aereoporti connessi
3) numero degli aereoporti connessi
4) trovo (max) degli aereoporti connessi
5) trovo (max size) aereoporti connessi
6) trovo 5 achi con peso minimo di (max size)
6)Permettere all'utente di selezionare, da un menu a tendina, una delle squadre presenti nel grafo,
 e premere il bottone "Connessioni Squadra".
-------------------------------------

 private ConnectivityInspector<Airport,DefaultWeightedEdge> ci;

 /* Escludendo gli aeroporti con zero rotte, determinare se nel grafo ottenuto è possibile da ogni aeroporto
 raggiungere ogni altro aeroporto.----->>> fortemente connesso?  */

        public String isConnesso () {
                ci = new ConnectivityInspector<>(this.graph);
            if (ci.isGraphConnected()){ return "Il grafo--> fortemente connesso";   }
                else return "il grafo---> non fortemente connesso";
        }

 /* lista degli aereoporti connessi */

         public List<Set<Team>> getConn(){
                        ci = new ConnectivityInspector<>(this.graph);
                        return ci.connectedSets();
                }

 /* numero degli aereoporti connessi */
                public int getNumberConn(){
                        return this.getConn().size();
                }

 /* trovo (max) degli aereoporti connessi */
                public Set<Team> MAXconnesso() {
                        Set<Team> MAX=null;
                        Double size=0.0;
                        for(Set<Team> atemp :this.getConn()){
                                if(atemp.size()>size){
                                        size = (double) atemp.size();
                                        MAX=atemp;
                                }
                         }
```

```
                        return MAX;
                }

 /* trovo (max size) aereoporti connessi */
        public int MAXconnessioni() {
                        return this.MAXconnesso().size();
                }

 /* trovo 5 achi con peso minimo di (max size) */

  public List<EdgePeso> get5ArchiMinWeight (){

                  List<EdgePeso>  edgePeso= new  ArrayList<>();
                  // avevo pensato di sostituire i vertici del grafo con il set trovato Max connesso per poi
   // richiamare il metodo CreaGrafo()
                 Set<Team> new_vertex =this.MAXconnesso();
                 for (Team a1: new_vertex){
                  for (Team a2: new_vertex){
                         if (! a1.equals(a2)&& a1!=null&&a2!=null){
                                 //peso
                                       DefaultWeightedEdge e=graph.getEdge(a1, a2);
                                              if(e!=null ){
                                                      graph.getEdgeWeight(e);
                                                      edgePeso.add(new EdgePeso(e,graph.getEdgeWeight(e)) );
                                              }
                                 }
                  }
                 }
                 //peso minimo se ti chiede max devi cambiare nella classe EDGE PESO
                 Collections.sort(edgePeso);
                 return edgePeso.subList(0, 5);
             }

// Permettere all'utente di selezionare, da un menu a tendina, una delle squadre presenti nel grafo,
// e premere il bottone "Connessioni Squadra".

 public Set<Team> getConn(String name){
                ConnectivityInspector<Team,DefaultWeightedEdge> ci = new ConnectivityInspector<>(this.graphTEAM);
 return ci.connectedSetOf(mapTeam.get(name));
 }

//-------------------------------------------------------
// ------------------> VISITE <---------------
//-------------------------------------------------------

1) raggiungibili data Stagione e Team di partenza (dijkstra)
2) raggiungibili da query data Airline
3) tutti raggiungibili calcolando punti  data stagione
4) CAMMINO MINIMO e TEMPO TOT CAMMINO
5) successori
6) predecessori
7) raggiungibili (visita in  profondità)---> breathfirstInterator
8) numero di raggiungibili (visita in  profondità)
9) BELLMANcalcolaPercorso(Team partenza)
10) VICINI CONNESSI


/*Si permetta all'utente di selezionare un team tra quelli raggiunti della stagione, e determinare
 tutti gli altrin team da esso raggiungibili con viaggi di una o piU partite . L'elenco deve essere ordinato per
 punti crescente  rispetto al team di partenza. */

        public List<TeamPunteggio> getDestinations(Season season, Team start) {

                        List<TeamPunteggio> list = new ArrayList<>();
                        for (Team end : this.getAllTeamsBySeason(season)) {
                                DijkstraShortestPath<Team, DefaultWeightedEdge> dsp =
  new DijkstraShortestPath<>(graph, start, end);
                                GraphPath<Team, DefaultWeightedEdge> p = dsp.getPath();
                                if (p != null) {

 // p.getEdgeList().size() è il numero di partite da
 //aggiungere dopo p.getWeight() e aggiungi un int nella classe

                                list.add(new TeamPunteggio(end, (int) p.getWeight()));
                                }
                        }
                        //ordinato per punti crescente  rispetto al team di partenza
                        list.sort(new Comparator<TeamPunteggio>() {
                                @Override
                                public int compare(TeamPunteggio o1, TeamPunteggio o2) {
```

```
                                        return Double.compare(o1.getPunteggio(), o2.getPunteggio());
                                }
                        });
                return list;
                }


/**
 * CAMMINO MINIMO e TEMPO TOT CAMMINO
 */

                public String calcolaPercorso(Team p, Team a) {
 //             this.creaGrafo();
                String s="";
                double tottime=0;
                DijkstraShortestPath<Team, DefaultWeightedEdge> dsp = new DijkstraShortestPath<>(graph, p,a);
                GraphPath<Team, DefaultWeightedEdge> path = dsp.getPath();
                        if(path == null)  return null;
                        List<Team> vertici = new ArrayList<>();

                        tottime = ((path.getEdgeList().size()) * 30 -60) + path.getWeight();  //- StazP e StazA

                        for(DefaultWeightedEdge e : path.getEdgeList()){
                                vertici.add(graph.getEdgeTarget(e));
                                }
                s+= "Percorso: "+ vertici.toString() + "\n"+ "Tempo tot: " + tottime;
                System.out.println(s);
                return s;

        }
//SUCCESSORI

        public List<Team> trovaSucessori(Team s) {
                        List<Team> successori = new ArrayList<Team>();
                        successori.addAll(Graphs.successorListOf(graph,s));

                        return successori;
                }

//PREDECCESSORI

        public List<Team> trovaPredecessori(Team s) {
                        List<Team>predecessori = new ArrayList<Team>();
                        predecessori.addAll(Graphs.predecessorListOf(graph,s));
                        return predecessori;
                }




//      quanti e quali altri stati sono raggiungibili(attraverso uno o piu archi)

        List<Team> vicini = new ArrayList<Team>();

        public List<Team> getRaggiungibiliInAmpiezza(Team partenza)
                {
                        BreadthFirstIterator<Team, DefaultWeightedEdge> visita =
                                        new BreadthFirstIterator<Team, DefaultWeightedEdge>( this.graph, partenza);

                        while(visita.hasNext()) //Finchè ogni nodo ha un successore
                        {
                                Team s = visita.next(); //Prendo il successore
                                vicini.add(s); //e lo metto nella lista
                        }
                        return vicini;
                }
        public int raggiungibili(){
                return vicini.size();
        }



 //BELLMAN---> UGUALE A disktra ma tiene conto peso negativo

        public String BELLMANcalcolaPercorso(Team partenza) {
                String ritorno = new String();

                BellmanFordShortestPath<Team, DefaultWeightedEdge> bellman =
                                new BellmanFordShortestPath<Team, DefaultWeightedEdge>(graph, partenza);
```

```
                Map<Team, Double> mappa = new HashMap();

                for (Team arrivo: graph.vertexSet()) {
                        if (!partenza.equals(arrivo)) {
                                double peso = bellman.getCost(arrivo);
                                mappa.put(arrivo, peso);
                        }
                }

                for (Team f: mappa.keySet()) {
                        ritorno = ritorno + f.getTeam() + " = " + mappa.get(f)+ " \n" ;
                }

                return ritorno;
        }

//VICINI CONNESSI
  public List<Team> trovaViciniConnessi(Team s) {
 List<Team> connessi = new ArrayList<Team>();
  connessi.addAll(Graphs.neighborListOf(graph,s));
 return  connessi;
 }

//-------------------------------------------------------
// ------------------> RICORSIONE  <---------------
//-------------------------------------------------------


/**
 * RICORSIONE 1  ------->  GET OTTIMO Team di tutti le partite
 *-----------------------------------------------------------------------
 */

 //GET OTTIMO di tutti le gare
 private List<Team> all;
 private Map<String,Integer> FantaTeam;
 private Map<String,List<Team>> mapTeam;
 private List<String> best2;

 //METODO FINALE

 public List<String> getOttimo(){
 List<String> parziale=new ArrayList<>();
 best2=new ArrayList<>();
 recursive(0,parziale);
 return best2;
 }

 //RICORSIONE

 private void recursive(int livello, List<String> parziale) {

 all=this.getAllTeams(); //di tutti le gare
 for(String s: parziale){
 for(Team p:mapTeam.get(s)){
 all.removeAll(Graphs.neighborListOf(graph, p));
 }
 }
 if(best2.isEmpty()|| parziale.size()<best2.size()){
 best2.clear();
 best2.addAll(parziale);
 System.out.println(best2);
 }
 for(String s:FantaTeam.keySet()){
 if(parziale.isEmpty() || s.compareTo(parziale.get(parziale.size()-1))>0){
 parziale.add(s);
 recursive(livello+1,parziale);
 parziale.remove(s);
 }
 }
 }
```

```
/**
------------------------------------------------------------
 * RICORSIONE 2
*/
 private List<Team> best;
 private List<DefaultWeightedEdge> bestArchi;

public List<Team> getLongestPath(){
 List<DefaultWeightedEdge> parzialeArchi= new ArrayList<>();
 bestArchi = new ArrayList<>();
 best = new ArrayList<>();
 recursive(parzialeArchi);
 for(DefaultWeightedEdge edge: bestArchi){
 best.add(graph.getEdgeSource(edge));
 }
 best.add(graph.getEdgeTarget(bestArchi.get(bestArchi.size()-1)));
 System.out.println("best: "+best.toString());
 return best;
 }

private void recursive(List<DefaultWeightedEdge> parzialeArchi) {
 // condizione terminazione == tutte le possibili ==> termina da solo
 //controllo se la dim di parziale è > best aggiorno best
 //System.out.println(parzialeArchi.toString()+"\n");

 if(parzialeArchi.size() >= bestArchi.size()){
 bestArchi.clear();
 bestArchi.addAll(parzialeArchi);
 System.out.println("bestArchi: "+bestArchi.toString()+"\n");
 }
 or(DefaultWeightedEdge edge : graph.edgeSet()){
 if(graph.getEdgeWeight(edge)==1){
 if(!parzialeArchi.contains(edge) && (parzialeArchi.size()==0  ||
graph.getEdgeTarget(parzialeArchi.get(parzialeArchi.size()-1)).equals(graph.getEdgeSource(edge)))){
 parzialeArchi.add(edge);
 recursive(parzialeArchi);
 parzialeArchi.remove(edge);
 }
 }
 }

 }

/**
  * RICORSIONE 3 ---->DOMINO
  * ------------------------------------------------------------------------
  * a. Quando l'utente seleziona la funzione "Domino", occorre ricercare e stampare la più lunga sequenza di partite
  * "concatenate", in cui ciascuna squadra sconfitta in una partita diviene vincitrice nella partita successiva.
  * Ad esempio, la sequenza potrà essere: TeamA, TeamC, TeamK, se nella partita TeamA-TeamC è risultato vincitore
TeamA,
  * e nella partita TeamC-TeamK è risultato vincitore TeamC.
  * b. Suggerimento: dal punto di vista del grafo, occorre trovare un cammino (aperto),
  * di lunghezza massima, che attraversi unicamente archi con peso +1. Gli archi con peso 0 oppure -1 non
  *  devono essere considerati.
  *  c. Si noti che il cammino potrebbe non essere semplice
  * (cioè lo stesso vertice potrebbe comparire più volte).
  *  Ad esempio: A-B, B-C, C-D, D-A, A-E, E-F. In questo caso A ha vinto su B e su E, ma ha perso da D.
  *  In ogni caso, ogni arco può essere utilizzato una sola volta (ad esempio, abbiamo potuto usare A-E,
  *  ma non sarebbe stato lecito ripetere di nuovo A-B).
  *  d. Al termine della ricerca, si stampi tale cammino.
  *
  */

//metodo
private void scegli(List<DefaultWeightedEdge> parziale, int livello, List<DefaultWeightedEdge> best, Team squadra) {

 // lo svolgo sugli archi perchè se no non considero
 //alcune partite, in quanto la squadra svolge piu partite sia in casa che in trasferta

 if(parziale.size() > best.size()){    best.clear();             //se parziale >cammino MAX  -> sostituisco
 best.addAll(parziale);
 System.out.println(best.toString());}

 else{
   for(DefaultWeightedEdge e: graph.outgoingEdgesOf(squadra)){    //ARCHI USCENTI graph.outgoingEdgesOf
 if(graph.getEdgeWeight(e)==1){
 if(!parziale.contains(e))  {
  parziale.add(e);
 scegli(parziale,livello+1,best,graph.getEdgeTarget(e));  //graph.getEdgeTarget
 // LANCIA RICORSIONE sui team
```

```
parziale.remove(e);
}
 }
   }
}
}
//implementazione metodo
public List<DefaultWeightedEdge> trovaSequenza(){
List<DefaultWeightedEdge> best=new ArrayList<DefaultWeightedEdge> ();
for( Team squadra: graph.vertexSet()){
List<DefaultWeightedEdge> parziale=new ArrayList<DefaultWeightedEdge>();
scegli(parziale,0,best,squadra);
}
return best;
}

/**
 * RICORSIONE 4 ---> DREAM TEAM
 * ----------------------------------------------------------------------
 * a. Facendo click sul pulsante “DreamTeam”, individuare un dream team.
 * b. Definiamo come team un gruppo di K squadre. La dimensione K del team viene stabilita dall’utente con
 * l’apposita casella di testo (k).
 * c. Il tasso di sconfitta di un team è definito come il numero totale di vittorie di una qualsiasi squadra non
 * appartenente al team su una qualsiasi squadra appartenente al team.
 * d. Un dream team è un team di K squadre che abbia il minimo tasso di sconfitta.
 * Suggerimento: utilizzare un algoritmo ricorsivo per esplorare gli insiemi di K squadre.
 * Suggerimento 2: effettuare delle prove con valori di K piccoli (1, 2 o 3).
 */


private List<Team> costruttori;
private List<Team> best;
private int K;
private int valore;

 public List<Team> dreamTeam(int k){
List<Team> parziale=new ArrayList<>();
best=new ArrayList<>();
this.K=k;
valore=Integer.MIN_VALUE;
recursive(0,parziale);
return best;
}

private void recursive(int livello, List<Team> parziale) {
if(parziale.size()==K){
if(this.getValoreTeam(parziale)>valore){
valore=this.getValoreTeam(parziale);
best.clear();
best.addAll(parziale); }
return;
}
for(Team ctemp:graph.vertexSet()){
if(parziale.isEmpty()){
parziale.add(ctemp);
recursive(livello+1,parziale);
parziale.remove(ctemp);
}
}
}
private int getValoreTeam(List<Team> parziale) {
Set<Team> sconfitti=new HashSet<Team>();
for(Team ctemp:parziale){
for(Team perdente:Graphs.successorListOf(graph, ctemp)){
if(!parziale.contains(perdente)){
sconfitti.add(perdente);
}
}
}
return sconfitti.size();
}
```