Problemes LP – Curs 2018-19
Tema 5: Grafs

Exercici 1 (nivell mig)

Volem crear una classe `Graf` amb *templates* per poder gestionar grafs amb qualsevol tipus d'informació als nodes i a les arestes. Per tant, la definició del *template* ha de tenir dos paràmetres, un pel tipus de la informació dels nodes i un altre pel tipus de la informació que es guarda a les arestes.
Aquesta classe `Graf` ha de tenir una funcionalitat molt similar a la que hem utilitzat a les sessions de classe (només amb alguns canvis que s'expliquen a continuació). Per tant, podeu reaprofitar bona part del codi que ja teniu. Ha de servir per poder guardar grafs dirigits o no dirigits i ponderats (amb informació associada a les arestes) o no. En aquest darrer cas assumirem que el tipus de les arestes és `int`.
A continuació detallem els mètodes públics que ha de tenir la classe (en tots els casos `TipusNode` fa referència al tipus del *template* per la informació associada als nodes i `TipusAresta`, al tipus per la informació associada a les arestes):

- ```
  Graf(vector<TipusNode>& nodes, vector<vector<int>> parellesNodes,
       bool dirigit);
  ```

  Constructor per grafs no ponderats que rep com a paràmetre un vector amb la informació dels nodes, un vector amb les parelles de nodes relacionats per una aresta i un booleà que indica si el graf és dirigit o no.

- ```
  Graf(vector<TipusNode>& nodes, vector<vector<int>> parellesNodes,
       vector<TipusAresta> valorsArestes, bool dirigit);
  ```

  Constructor per grafs ponderats. Apart dels paràmetres del constructor anterior, rep també un vector amb la informació associada a cadascuna de les arestes, especificades en el mateix ordre que al vector `parellesNodes`.

- ```
  bool afegeixAresta(const TipusNode& node1, const TipusNode& node2);
  ```

  Afegeix una aresta un graf no ponderat entre els dos nodes que es passen com a paràmetre (fixeu-vos que es passa la informació associada al node i no la posició del node com hem fet a les sessions de classe). S'ha de comprovar que els dos nodes existeixin dins del graf. Si algun dels dos no existeix, no s'afegeix l'aresta i s'ha de retornar `false`. Si l'aresta es pot afegir correctament, s'ha de retornar `true`.

- ```
  bool afegeixAresta(const TipusNode& node1, const TipusNode& node2,
       const TipusAresta& valorAresta);
  ```

  Afegeix una aresta un graf ponderat. La única diferència amb el mètode anterior està en la informació associada a l'aresta que es passa en el paràmetre `valorAresta`.

- ```
  bool eliminaAresta(const TipusNode& node1, const TipusNode& node2);
  ```

  Elimina l'aresta entre els dos nodes que es passen com a paràmetre (fixeu-vos que es passa la informació associada al node i no la posició del node com hem fet a les sessions de classe). S'ha de comprovar que els dos nodes existeixin dins del graf. Si algun dels dos no
  ```

existeix, no s'elimina l'aresta i s'ha de retornar `false`. Si l'aresta es pot afegir correctament, s'ha de retornar `true`.

- ```
  bool afegeixNode(const TipusNode& node);
  ```

  Afegeix un node al graf amb la informació associada que es passa com a paràmetre. S'ha de comprovar que el node no existís prèviament al graf. Si ja existeix s'ha de retornar `false`. Si el node es pot afegir correctament s'ha de retornar `true`.

- ```
  bool eliminaNode(const TipusNode& node);
  ```

  Elimina el node del graf amb la informació associada que es passa com a paràmetre. Elimina també totes les seves arestes. S'ha de comprovar que el node existeix al graf. Si no existeix s'ha de retornar `false`. Si el node es pot eliminar correctament s'ha de retornar `true`.

- ```
  bool getArestesNode(const TipusNode& node,
        vector<TipusNode>& nodesAdjacents) const;
  ```

  Recupera totes les arestes associades al node amb la informació que es passa com a paràmetre. Les arestes s'han de retornar al paràmetre `nodesAdjacents`. En aquest vector s'ha de guardar la informació de cadascun dels nodes relacionats amb el node actual a través d'alguna aresta. Només s'han de considerar les arestes que surten del node, no les que arriben al node.

- ```
  bool getArestesNode(const TipusNode& node,
        vector<TipusNode>& nodesAdjacents,
        vector<TipusAresta>& valorsArestes) const;
  ```

  Recupera totes les arestes associades a un node en un graf ponderat. És molt similar al mètode anterior, excepte en què, a més a més de la informació dels nodes relacionats a través d'alguna aresta, s'ha de retornar també la informació associada a cadascuna de les arestes al vector `valorArestes`.

- ```
  void mostra() const;
  ```

  Mostra la informació del graf exactament de la mateixa forma que està fet al codi que hem fet servir a les sessions de classe.
  ```

Exercici 2 (nivell mig) – EXERCICI AVALUABLE

Volem modificar el codi de la classe `Graf` representada amb llistes de nodes adjacents per guardar les arestes (enlloc de la matriu d'adjacència) que vam explicar a classe i que us vam deixar a Caronte com a part del material de la sessió 14.

En aquest codi (que us deixem a Caronte per començar l'exercici) hi volem afegir mètodes per trobar tots els cicles de tres nodes del graf, els graus de cadascun dels nodes i els camins per anar d'un node a un altre dins del graf. Tots aquests mètodes ja els hem fet a classe, però per la representació de les arestes amb la matriu d'adjacència. Ara els volem adaptar a la representació amb llistes de nodes adjacents. Tots els mètodes han de funcionar correctament tant en grafs dirigits com en grafs no dirigits.

En concret, els mètodes que s'han d'implementar són els següents:

- ```
  vector<vector<string>> cicles();
  ```

  Retorna tots els cicles de 3 nodes que tingui el graf. Els cicles s'han de retornar en un vector on a cada posició es guarda un vector de 3 elements amb l'etiqueta de cadascun dels tres nodes del cicle.

- ```
  int grauOutNode(string node);
  ```

  Retorna el grau de sortida del node que es passa com a paràmetre. El grau de sortida és el nombre d'arestes que surten del node.

- ```
  int grauInNode(string node);
  ```

  Retorna el grau d'entrada del node que es passa com a paràmetre. El grau d'entrada és el nombre d'arestes que arriben al node.

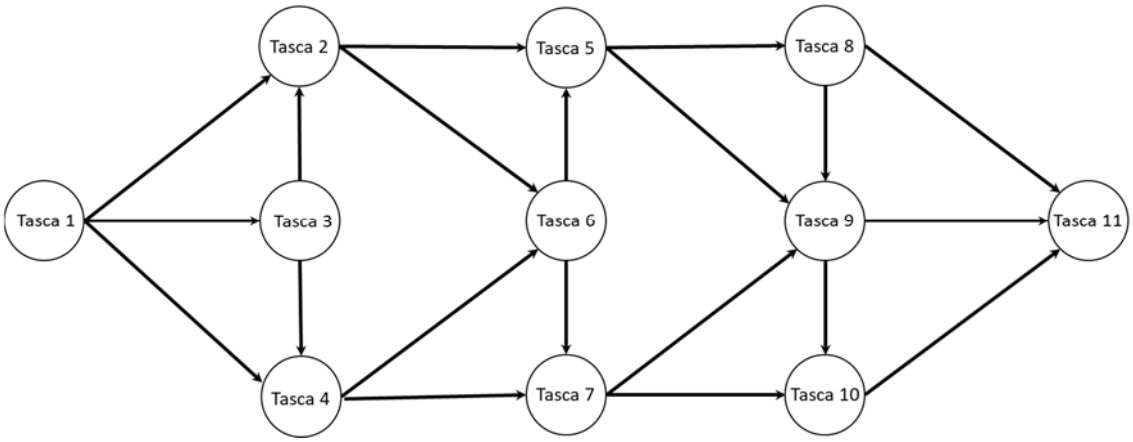
- ```
  void camins(string node1, string node2,
        vector<list<string>>& camins);
  ```

  NO AVALUABLE(*) - Retorna tots els camins que podem seguir dins del graf per anar del `node1` al `node2`, que es passen com a paràmetres. A diferència del mètode que vam fer a classe per la representació amb la matriu d'adjacència, aquí no volem mostrar tots els camins per pantalla sinó retornar-los per referència al paràmetre `camins`. Haurem de retornar un vector, on a cada posició del vector hi guardem un dels possibles camins entre els dos nodes. Dins del vector, cada camí es guarda com una llista amb les etiquetes de tots els nodes del camí.

  (*) Aquest mètode és no avaluable. Forma part del test que us deixem a Caronte, de forma que podreu veure si funciona correctament, però el resultat del test d'aquest mètode no es tindrà en compte per calcular la nota final de l'exercici.

Exercici 3 (nivell mig) – EXERCICI AVALUABLE

Suposem que tenim un graf que representa les dependències entre les tasques necessàries per realitzar un projecte, similar a l'exemple que hem utilitzat en el recorregut en profunditat (DFS) de la sessió 15.



En aquest graf dirigit cada node representa una de les tasques a realitzar. Una aresta entre dos nodes indica una dependència entre les dues tasques. Per exemple, l'aresta entre la Tasca 1 i la Tasca 2 indica que per poder fer la Tasca 2, primer hem d'haver completat la Tasca 1.

a) Volem modificar l'algorisme de recorregut en profunditat (DFS) perquè a més a més del recorregut del graf ens retorni en quin ordre hem de realitzar les tasques per complir amb les relacions de dependència que ens marquen les arestes. En aquest exemple, un possible ordre de les tasques seria: Tasca 1, Tasca 3, Tasca 2, Tasca 4, Tasca 6, Tasca 5, Tasca 7, Tasca 8, Tasca 9, Tasca 10, Tasca 11.

   L'ordre de les tasques es pot deduir a partir del recorregut en profunditat, tal com està explicat al vídeo del DFS que teniu publicat a Caronte. Repasseu el vídeo per veure com ho heu de fer.

   Bàsicament, cada cop que en el recorregut ja no puguem seguir expandint un node i haguem de tornar enrere guardarem aquest node en una pila que és la que ens donarà l'ordre de les tasques. En l'exemple, el primer node que ja no podem expandir més i afegirem a la pila és la Tasca 11, després la Tasca 10, la Tasca 9, la Tasca 8 i així successivament. Com podem observar, el DFS ens permet obtenir l'ordre invers en què hem de fer les tasques. Per tant, si ho guardem a una pila, quan traiem els elements de la pila obtindrem l'ordre correcte.

   En el codi de la classe Graf us demanem que afegiu una versió sobrecarregada del mètode que calcula el DFS per poder retornar també la pila que ens guarda l'ordre invers de les tasques, amb aquesta capçalera:

```
void DFS(string nodeInicial, queue<string>& recorregut,
                stack<string>& ordre);
```

b) **NO AVALUABLE(*)** – En un graf com l'anterior que representa les dependències entre tasques, la detecció de cicles en el graf ens serveix per comprovar si la planificació que hem fet és correcta o no. Si el graf conté algun cicle vol dir que hi ha una dependència cíclica entre vàries tasques i per tant, la planificació no és realitzable.

Volem implementar un mètode que ens permeti detectar si hi ha algun cicle en un graf dirigit com el de l'exemple anterior. Per poder detectar cicles hem d'aplicar l'algorisme DFS i comprovar si en algun moment del recorregut tornem a un node que ja hem visitat prèviament en el camí actual del recorregut. Per això haurem d'introduir dues modificacions a l'algorisme del DFS:

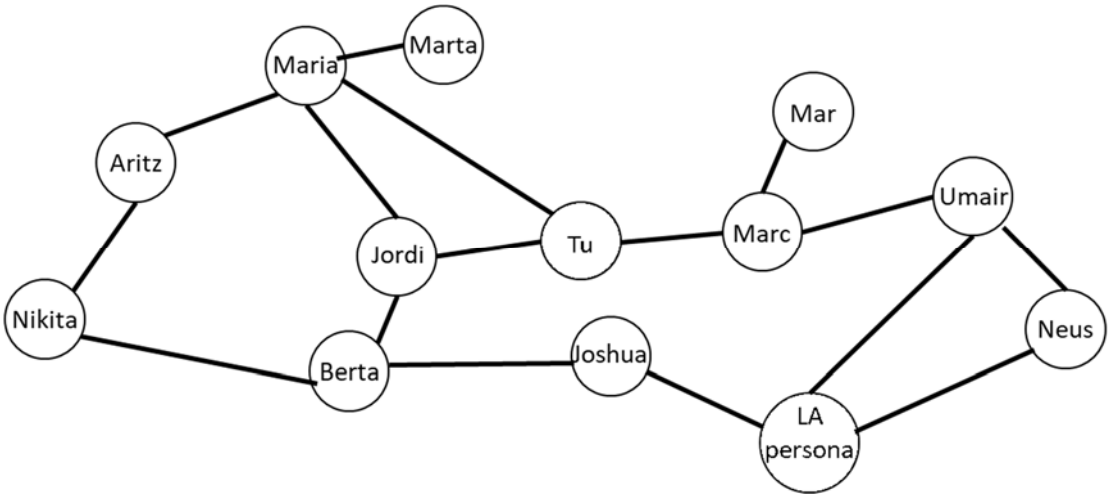
- Haurem de guardar en cada moment de l'algorisme quins nodes formen part del camí que segueix el recorregut des del node inicial. Tinguem en compte que cada cop que visitem un nou node l'hem d'afegir al camí actual, però que quan ja no podem seguir expandint un node i tornem enrere, aquest node deixa d'estar en el camí actual. Si quan expandim un node en el recorregut anem a parar a un node que està marcat com a part del camí actual vol dir que hi ha un cicle al graf.
- El recorregut no té un únic node inicial. Haurem d'escollir un primer node del graf i fer el recorregut des d'aquest node, però quan acabem de fer el recorregut des d'aquest primer node, si encara hi ha nodes per visitar, n'haurem d'escollir un altre (dels no visitats) i tornar a començar. D'aquesta forma, ens assegurem que explorem tot el graf per buscar els cicles.

Podeu trobar una explicació més detallada de com utilitzar el DFS per detectar cicles en un graf al següent enllaç: https://www.geeksforgeeks.org/detect-cycle-in-a-graph/

(*) Aquest mètode és no avaluable. Forma part del test que us deixem a Caronte, de forma que podreu veure si funciona correctament, però el resultat del test d'aquest mètode no es tindrà en compte per calcular la nota final de l'exercici.

Exercici 4 (nivell mig) – EXERCICI AVALUABLE

Suposem que tenim un graf que representa les relacions entre persones en una xarxa social, similar a l'exemple que hem utilitzat en el recorregut en amplada (BFS) de la sessió 15.



En aquest graf no dirigit cada node representa una persona. Una aresta entre dos nodes indica un contacte entre les dues persones. Hem vist a la sessió 15 que el recorregut en amplada (BFS) permet trobar el camí més curt entre dos nodes del graf.

a) Volem modificar l'algorisme BFS que hem vist a classe perquè ens retorni el número de passos mínim per poder anar d'un node inicial a un node final del graf. En el graf de la xarxa social anterior representa a quina distància (en número de contactes entre elles) estan dues persones.
Per fer aquesta modificació haurem d'afegir un vector que guardi per cada node un comptador indicant quants passos han estat necessaris per arribar-hi. Cada cop que visitem un nou node actualitzarem el seu comptador a partir del número de passos del node anterior en el recorregut.
En el codi de la classe Graf us demanem que afegiu una versió sobrecarregada del mètode que calcula el BFS per poder retornar el número de passos entre el node inicial i el node final:

```
int BFS(string nodeInicial, string nodeFinal,
                            queue<string>& recorregut);
```

Tingueu en compte que no fa falta generar tot el recorregut BFS. En el moment en què arribem al node final ja podem parar el recorregut. Si el node final que es passa com a paràmetre no existeix al graf, s'ha de retornar -1 com a resultat de la funció.

b) Volem tornar a modificar l'algorisme BFS per mostrar el camí més curt entre dos nodes del graf, és a dir, quins contactes hem de seguir dins de la xarxa per arribar a una determinada persona.
Per fer aquesta modificació haurem de guardar en un vector, per cada node del graf, quin és el seu node anterior en el recorregut BFS. És a dir, cada cop que durant el recorregut

visitem un nou node guardarem a la posició corresponent del vector quin és el node a partir del qual hi hem arribat. D'aquesta forma quan arribem al node final, podem recuperar cap enrere el camí que hem seguit fins arribar-hi, i guardar-lo en una pila.
La capçalera del mètode BFS modificat serà la següent:

```
void BFS(string nodeInicial, string nodeFinal,
         queue<string>& recorregut, stack<string>& cami);
```

Igual que abans, tingueu en compte que no fa falta generar tot el recorregut BFS. En el moment en què arribem al node final ja podem parar el recorregut. Si el node final que es passa com a paràmetre no existeix al graf, s'ha de retornar la pila buida.

c) **NO AVALUABLE(*)** – Volem fer una tercera modificació a l'algorisme BFS perquè ens retorni tots els nodes que estan a una certa distància màxima del node inicial. És a dir, hem de fer el recorregut BFS, però enlloc de parar quan arribem al node final, o quan hem recorregut tots els nodes, hem de parar quan el nº de passos des del node inicial fins al node actual del recorregut sigui més gran que la distància que li passem com a paràmetre a la funció.
La capçalera d'aquesta versió modificada serà la següent:

```
void BFS(string nodeInicial, int distancia,
         queue<string>& recorregut);
```

(*) Aquest mètode és no avaluable. Forma part del test que us deixem a Caronte, de forma que podreu veure si funciona correctament, però el resultat del test d'aquest mètode no es tindrà en compte per calcular la nota final de l'exercici.