1. Funciones Auxiliares

```
#include <iostream>
#include "list"
#include "map"
#include "algorithm"
#include "vector"
#include "fstream"
#include "tuple"

#define INFINITOPOSITIVO 10000
#define MENOSINFINITO -10000

typedef map<int, map<int,float>> Grafo;
typedef map<int,tuple<int,map<int,float>>> GrafoColoreado;

bool esGrafo(Grafo grafo, vector<int> &valores){
 int siz = grafo.size();
 for(auto iter = grafo.begin(); iter != grafo.end(); ++iter){
     if(iter->second.size() > siz) return false;
     valores.push_back(iter->first);
 }
 for(auto iter = grafo.begin(); iter != grafo.end(); ++iter){
     for(auto iter2 = iter->second.begin(); iter2 != iter->second.end(); ++iter2){
         if(find(valores.begin(), valores.end(), iter2->first) == valores.end())return false;
     }
 }
 return true;
}

int caminoMinimo(Grafo grafo, int a, int b, vector<int>& visitado, bool &flag){
 flag = false;
 if(find(visitado.begin(), visitado.end(),a) != visitado.end() or a == b){
     flag = false;
     return INFINITOPOSITIVO;
 }
 int resultado = 0;
 visitado.push_back(a);
 if(grafo[a].find(b) != grafo[a].end()){
     flag = true;
     resultado = grafo[a][b];
 }
 int menor = INFINITOPOSITIVO;
 for(auto iter = grafo[a].begin(); iter != grafo[a].end(); ++iter){
     if(flag) menor = min(menor,resultado);
     resultado = caminoMinimo(grafo,iter->first,b,visitado,flag) + iter->second;
 }
 if(flag) menor = min(menor,resultado);
 if(menor == INFINITOPOSITIVO) flag = false;
 else flag = true;
 return menor;
}

template <typename T>
vector<T> filtreRepeat(vector<T> valores){
 vector<T> resultado;
 for(int i = 0; i < valores.size(); i++){
     if(find(resultado.begin(),resultado.end(),valores[i]) == resultado.end()) resultado.push_back(↩
         valores[i]);
 }
 return resultado;
}

template <typename T>
int find(vector<T> valores, T a){
 for(int i = 0; i < valores.size(); i++){
     if(valores[i] == a)return i;
 }
 return -1;
}
```

2. Camino Simple

Consiste en estudiar la existencia de un camino entre dos vértices cualquiera.

```
bool hayCamino(Grafo grafo, int a, int b, vector<int>& visitado){
 if(find(visitado.begin(), visitado.end(),a) != visitado.end())return false;
 visitado.push_back(a);
 if(grafo[a].find(b) != grafo[a].end())return true;
```

```
    for(auto iter = grafo[a].begin(); iter != grafo[a].end(); ++iter){
        if(hayCamino(grafo,iter->first,b,visitado)) return true;
    }
    return false;
}
```

3. Conectividad Simple

Consiste en estudiar si el grafo es conexo, es decir, si existe al menos un camino entre cada par de vértices.

```
bool conexo(Grafo grafo){
    vector<int> valores;
    if(!esGrafo(grafo,valores)) return false;
    for(int i = 0; i < valores.size(); i++){
        vector<int> valTemp = valores;
        int temp = valores[i];
        valTemp.erase(valTemp.begin() + i);
        for(int j = 0; j < valTemp.size(); j++){
            vector<int> visitado;
            if(!hayCamino(grafo,valores[i],valTemp[j],visitado))return false;
        }
    }
     return true;
}
```

4. Detección de Ciclos

Consiste en estudiar la existencia de al menos un ciclo en el grafo.

```
bool ciclos(Grafo grafo){
    vector<int> valores;
    if(!esGrafo(grafo,valores)) return false;
    for(auto iter = grafo.begin(); iter != grafo.end(); ++iter){
        if(iter->second.find(iter->first) != iter->second.end()) return true;
    }
    return false;
}
```

5. Camino de Euler

Consiste en estudiar la existencia de un camino que conecte dos vértices dados usando cada arista del grafo exactamente una sola vez.

```
void relaciones(Grafo grafo, vector<tuple<int,int>> &resultado){
    for(auto iter = grafo.begin(); iter != grafo.end(); ++iter){
        for(auto iter2 = iter->second.begin(); iter2 != iter->second.end(); ++iter2){
            resultado.push_back(make_tuple(iter->first,iter2->first));
        }
    }
}

bool _caminoEuler(Grafo grafo,int x, int a, int b, vector<tuple<int,int>> restantes){
    auto temp = make_tuple(x,a);
    if(x != -1 and find(restantes.begin(),restantes.end(),temp) == restantes.end())return false;
    restantes.erase(restantes.begin() + find(restantes,temp));
    for(auto iter = grafo[a].begin(); iter != grafo[a].end(); ++iter){
        if(_caminoEuler(grafo,a,iter->first,b,restantes))return true;
    }
```

```
 return false;
}

bool caminoEuler(Grafo grafo, int a, int b){
 vector<int> valores;
 if(!esGrafo(grafo,valores))return false;
 vector<tuple<int,int>> rel;
 relaciones(grafo,rel);
 return _caminoEuler(grafo,-1,a,b,rel);
}
```

6. Camino de Hamilton

Consiste en estudiar la existencia de un camino que conecte dos vértices dados que visite cada nodo del grafo exactamente una vez.

```
bool _caminoHamilton(Grafo grafo, int a, int b, vector<int> restantes){
 if(find(restantes.begin(),restantes.end(),a) == restantes.end())return false;
 if(a == b and restantes.empty())return true;
 restantes.erase(restantes.begin() + find(restantes,a));
 for(auto iter = grafo[a].begin(); iter != grafo[a].end(); ++iter){
     if(_caminoHamilton(grafo,iter->first,b,restantes))return true;
 }
 return false;
}

bool caminoHamilton(Grafo grafo, int a, int b){
 vector<int> valores;
 if(!esGrafo(grafo,valores))return false;
 return _caminoHamilton(grafo,a,b,valores);
}
```

7. Conectividad Fuerte en Digrafos

Consiste en estudiar si hay un camino dirigido conectando cada par de vértices del digrafo. Inclusive se puede estudiar si existe un camino dirigido entre cada par de vértices, en ambas direcciones.

```
bool conexoFuerte(Grafo grafo){
 vector<int> valores;
 if(!esGrafo(grafo,valores)) return false;
 for(int i = 0; i < valores.size(); i++){
     vector<int> valTemp = valores;
     int temp = valores[i];
     valTemp.erase(valTemp.begin() + i);
     for(int j = 0; j < valTemp.size(); j++){
         vector<int> visitado;
         if(!hayCamino(grafo,valores[i],valTemp[j],visitado))return false;
     }
 }
  return true;
}
```

8. Clausura Transitiva

Consiste en tratar de encontrar un conjunto de vértices que pueda ser alcanzado siguiendo aristas dirigidas desde cada vértice del digrafo.

```
vector<int> clausuraTransitiva(Grafo grafo){
 vector<int> valores;
 vector<int> resultado;
 if(!esGrafo(grafo,valores)) return resultado;
 for(int i = 0; i < valores.size(); i++){
     vector<int> valTemp = valores;
     valTemp.erase(valTemp.begin() + i);
     bool flag = true;
     for(int j = 0; j < valTemp.size(); j++){
         vector<int> visitado;
         if(!hayCamino(grafo,j,i,visitado)){
             flag = false;
             break;
         }
     }
     if(flag)resultado.push_back(i);
 }
 return resultado;
}
```

9. Arbol de Expansión Mínima

Consiste en encontrar, en un grafo pesado, el conjunto de aristas de peso mínimo que conecta a todos los vértices.

```
void arbolDeEpansionMinima(Grafo &grafo){
 vector<int> valores;
 if(!esGrafo(grafo,valores))return;
 for(int i = 0; i < valores.size(); i++){
     for(int j = 0; j < valores.size(); j++){
         if(i != j){
             vector<int> visitado;
             bool flag = false;
             int menor = caminoMinimo(grafo,i,j,visitado,flag);
             if(flag) grafo[i][j] = menor;
         }
     }
 }
}
```

10. Caminos cortos a partir de un mismo origen

Consiste en encontrar cuales son los caminos más cortos conectando un vértice v cualquiera con cada uno de los otros vértices de un digrafo pesado. Este es un problema que por lo general se presenta en redes de computadoras, representadas como grafos.

```
void caminosCortosAPartirDeUnOrigen(Grafo &grafo, int a){
 vector<int> valores;
 if(!esGrafo(grafo,valores))return;
 for(int i = 0; i < valores.size(); i++){
     if(valores[i] != a){
         vector<int> visitado;
         bool flag = false;
         int menor = caminoMinimo(grafo,a,i,visitado,flag);
         if(flag) grafo[a][valores[i]] = menor;
     }
 }
}
```

11. Pareamiento

Dado un grafo, consiste en encontrar cual es el subconjunto más largo de sus aristas con las propiedades de que no haya dos conectados al mismo vértice. Se sabe que este problema clásico es resoluble en tiempo proporcional a una función polinomial en el número de vértices y de aristas, pero aun no existe un algoritmo rápido que se ajuste a grandes grafos.

```
vector<tuple<int,int>> _pareamiento(Grafo grafo, int a, vector<int> valores){
 vector<tuple<int,int>> resultado;
 if(valores.empty())return resultado;
 int mayor = 0;
 int actual = find(valores, a);
 valores.erase(valores.begin() + actual);
 for(auto iter = grafo[a].begin(); iter != grafo[a].end(); ++iter){
     int temp = find(valores, iter->first);
     vector<tuple<int,int>> v;
     if(temp != -1){
         valores.erase(valores.begin() + temp);
         v = _pareamiento(grafo,valores.front(),valores);
         v.push_back(make_tuple(a,iter->first));
         if(mayor < v.size() and v.size() != 0){
             mayor = v.size();
             resultado = v;
         }
     }
 }
 auto t = _pareamiento(grafo,valores.front(),valores);
 resultado.insert(resultado.end(),t.begin(),t.end());
 return resultado;
}

int pareamiento(Grafo grafo, vector<tuple<int,int>>& resultado){
 vector<int> valores;
 int mayor = 0;
 if(!esGrafo(grafo,valores))return 0;
 for(int i = 0; i < valores.size(); i++){
     vector<tuple<int,int>> temp = _pareamiento(grafo,valores[i],valores);
     if(mayor <  temp.size() and temp.size() != 0){
         mayor = temp.size();
         resultado = temp;
     }
 }
 resultado = filtreRepeat(resultado);
 return resultado.size();
}
```

12. Ciclos Pares en Digrafos

Consiste en encontrar en un digrafo un camino de longitud par. Este problema puede lucir simple ya que la solución para grafos no dirigidos es sencilla. Sin embargo, aun no se conoce si existe un algoritmo eficiente para resolverlo.

```
void _caminosPares(Grafo grafo, int a, vector<int> visitado, vector<int> &resultado, int peso){
 if(find(visitado.begin(), visitado.end(),a) != visitado.end())return;
 if(peso != 0 and peso % 2 == 0){
     resultado.push_back(a);
 }
 visitado.push_back(a);
 for(auto iter = grafo[a].begin(); iter != grafo[a].end(); ++iter){
     _caminosPares(grafo,iter->first,visitado,resultado,peso + iter->second);
 }
}

vector<tuple<int,int>> caminosPares(Grafo grafo){
 vector<int> valores;
 vector<tuple<int,int>> resultado;
 if(!esGrafo(grafo,valores))return resultado;
 for(int i = 0; i < valores.size(); i++){
     vector<int> res;
     vector<int> visitado;
     _caminosPares(grafo,valores[i],visitado,res,0);
     for(int j = 0; j < res.size(); j++){
```

```
            resultado.push_back(make_tuple(valores[i],res[j]));
        }
    }
    return resultado;
}
```

13. Asignación

Este problema se conoce también como pareamiento bipartito pesado. Consiste en encontrar un pensamiento perfecto de peso mínimo en un grafo bipartito. Un grafo bipartito es aquel cuyos vértices se pueden separar en dos conjuntos, de tal manera que todas las aristas conecten a un vértice en un conjunto con otro vértice en el otro conjunto.

```
int minRelacion(map<int,float> g, int a){
    int menor = 10000;
    int index = 0;
    for(auto iter = g.begin(); iter != g.end(); ++iter){
        if(iter->second < menor){
            menor = iter->second;
            index = iter->first;
        }
    }
    return index;
}

vector<tuple<int,int,int>> _asignacion(Grafo grafo, int a, vector<int> valores){
    vector<tuple<int,int,int>> resulclique grafostado;
    if(valores.empty())return resultado;
    int actual = find(valores, a);
    valores.erase(valores.begin() + actual);
    int index = minRelacion(grafo[a],a);
    int temp = find(valores, index);
    map<int,float> tt = grafo[a];
    while(!tt.empty() and temp == -1){
        tt.erase(index);
        index = minRelacion(tt,a);
        temp = find(valores, index);
    }
    vector<tuple<int,int,int>> v;
    if(!tt.empty()){
        valores.erase(valores.begin() + temp);
        v = _asignacion(grafo,valores.front(),valores);
        v.push_back(make_tuple(a,index,grafo[a][index]));
        resultado.insert(resultado.end(),v.begin(),v.end());
    }
    return resultado;
}

int asignacion(Grafo grafo, vector<tuple<int,int,int>>& resultado){
    vector<int> valores;
    int menor = 10000;
    if(!esGrafo(grafo,valores))return 0;
    for(int i = 0; i < valores.size(); i++){
        vector<tuple<int,int,int>> v = _asignacion(grafo,valores.front(),valores);
        int suma = 0;
        for(auto iter = v.begin(); iter != v.end(); ++iter){
            suma += get<2>(*iter);
        }
        if(menor > suma){
            menor = suma;
            resultado = v;
        }
    }
    resultado = filtreRepeat(resultado);
    return resultado.size();
}
```

14. EL camino más largo

Consiste en encontrar cual es el camino más largo que conecte a dos nodos dados en el grafo.

```
int caminoMasLargo(Grafo grafo,int a, int b, vector<int> visitado, bool flag){
 flag = false;
 if(find(visitado.begin(), visitado.end(),a) != visitado.end() or a == b){
     flag = false;
     return -1;
 }
 int resultado = -1;
 visitado.push_back(a);
 if(grafo[a].find(b) != grafo[a].end()){
     flag = true;
     resultado = grafo[a][b];
 }
 int mayor = -1;
 for(auto iter = grafo[a].begin(); iter != grafo[a].end(); ++iter){
     if(flag) mayor = max(mayor,resultado);
     resultado = caminoMasLargo(grafo,iter->first,b,visitado,flag) + iter->second;
 }
 if(flag)mayor = max(mayor,resultado);
 if(mayor == -1) flag = false;
 else flag = true;
 return mayor;
}
```

15. Colorabilidad

Consiste en estudiar si existe alguna manera de asignar k colores a cada uno de los vértices de un grafo, de tal forma de que ninguna arista conecte dos vértices del mismo color.

```
bool verificarColoracion(GrafoColoreado grafo, int a){
 int color = get<0>(grafo[a]);
 for(auto iter = get<1>(grafo[a]).begin(); iter != get<1>(grafo[a]).end(); ++iter){
     if(get<0>(grafo[iter->first]) == color)return false;
 }
 return true;
}

bool coloracion(Grafo grafo){
 vector<int> valores;
 if(!esGrafo(grafo,valores))return false;
 GrafoColoreado grafoColoreado;
 for(int i = 0; i < valores.size(); i++){
     grafoColoreado[valores[i]] = make_tuple(0,grafo[valores[i]]);
 }
 for(int i = 0; i < valores.size(); i++){
     if(get<0>(grafoColoreado[valores[i]]) == 0)get<0>(grafoColoreado[valores[i]]) = 1;
     if(!verificarColoracion(grafoColoreado,valores[i]))return false;
     int color = 0;
     if(get<0>(grafoColoreado[valores[i]]) == 1) color = 2;
     else color = 1;
     for(auto iter = get<1>(grafoColoreado[valores[i]]).begin(); iter != get<1>(grafoColoreado[↩
         valores[i]]).end(); ++iter){
         get<0>(grafoColoreado[iter->first]) = color;
     }
 }
 return true;
}
```