15 Grafos (Búsqueda en profundidad)

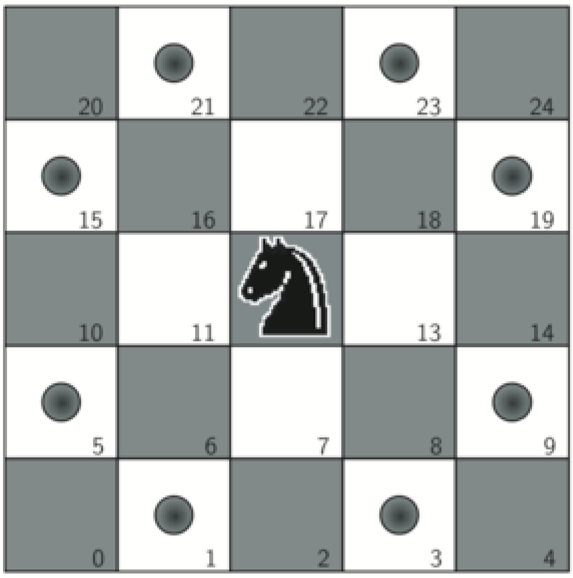
Graph.h

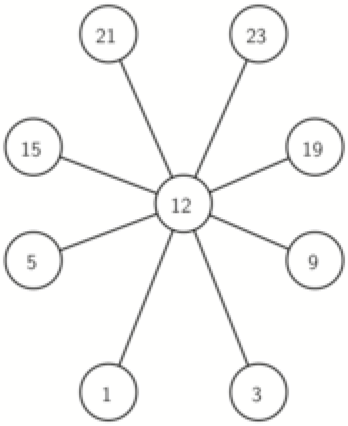
```
#include <iostream>
#include <vector>

using namespace std;

template<class T>
struct Edge;
template<class T>
class Vertex;

template<class T>
class Edge{
public:
   Vertex<T>* to;
   int weight;
   friend ostream &operator<<(ostream &out, Edge<T>* edge) {
       out << "To: " << edge->to->data;
       out << ", Weight: " << edge->weight << endl;
       return out;
   }
};

template<class T>
class Vertex{
public:
   T data;
   int inDegree;
   int outDegree;
   vector<Edge<T>*> connectedTo;

   Vertex<T>* predecessor;
   int distance;
   char color;

   int discovery;
   int finish;

   Vertex(const T& value);
   ~Vertex();

   void addNeighbor(Vertex<T>* to, int weight=0);
   int getWeight(const T& value);

   vector<Edge<T>*> getConnectedTo();
   friend ostream &operator<<(ostream &out, Vertex<T>* vertex) {
       out << vertex->data << endl;
       out << "In degree: " << vertex->inDegree << endl;
       out << "out degree: " << vertex->outDegree << endl;
       out << "Edges: " << endl;
       vertex->connectedTo.print();
       return out;
   }
};

template<class T>
class Graph {
public:
   int count;
   vector<Vertex<T>*> vertexList;
   Graph();
   ~Graph();
   Vertex<T>* addVertex(const T& value);
   Vertex<T>* getVertex(const T& value);
   void addEdge(const T& from, const T& to, int weight=0);
};

#endif //GRAPHS_GRAPH_H
```

Graph.cpp

```
#ifndef GRAPHS_GRAPH_CPP
#define GRAPHS_GRAPH_CPP

#include <algorithm>
#include "Graph.h"
template<class T>
Vertex<T>::Vertex(const T& value) {
   data = value;
   inDegree = 0;
   outDegree = 0;
   connectedTo = {};
   predecessor = 0;
   distance = 0;
   color = 'w';
   discovery = 0;
   finish = 0;
}

template<class T>
Vertex<T>::~Vertex() {

}

template<class T>
void Vertex<T>::addNeighbor(Vertex<T> *to, int weight) {
   Edge<T>* temp = new Edge<T>;
   temp->to = to;
   temp->weight = weight;

   outDegree++;
   to->inDegree++;
   connectedTo.push_back(temp);
}

template<class T>
int Vertex<T>::getWeight(const T &value) {
   for(int i=0; i < connectedTo.size(); i++){
       Edge<T>* temp = connectedTo.get(i);
       if(temp->to->data == value){
           return connectedTo.get(i)->weight;
       }
   }
   return NULL;
}

template<class T>
Graph<T>::Graph() {
   count = 0;
   vertexList = {};
}

template<class T>
Graph<T>::~Graph() {
}

template<class T>
Vertex<T>* Graph<T>::addVertex(const T &value) {
   Vertex<T>* newVertex = new Vertex<T>(value);
   vertexList.push_back(newVertex);
   count++;
   return newVertex;
}

template<class T>
void Graph<T>::addEdge(const T& from, const T& to, int weight) {
   Vertex<T>* fromVertex = getVertex(from);
   if(!fromVertex){
       fromVertex = addVertex(from);
   }
   Vertex<T>* toVertex = getVertex(to);
   if(!toVertex){
       toVertex = addVertex(to);
   }
   fromVertex->addNeighbor(toVertex, weight);
}

template<class T>
Vertex<T> *Graph<T>::getVertex(const T &value) {
   for(int i=0; i < vertexList.size();i++ ){
       if(vertexList[i]->data == value) return vertexList[i];
   }
   return NULL;
}

template<class T>
vector<Edge<T>*> Vertex<T>::getConnectedTo(){
    sort(connectedTo.begin(), connectedTo.end(), [](Edge<T>* a,
Edge<T>* b){ return a->to->data < b->to->data; });
    return connectedTo;
}
#endif
```

Solución: Problema de la gira del caballo





Funciones auxiliares

```
int positionToId(int row, int col, int boardSize){
    return (row * boardSize) + col;
}

bool legalCoord(int x, int boardSize){
    if(x >= 0 && x < boardSize) return true;
    else return false;
}

list<tuple<int, int>> generateLegalMoves(int x, int y, int
boardSize){
    list<tuple<int, int>> newMoves;
    list<tuple<int, int>> moveOffsets;
    moveOffsets.push_back(tuple<int,int>(-1,-2));
    moveOffsets.push_back(tuple<int,int>(-1,2));
    moveOffsets.push_back(tuple<int,int>(-2,-1));
    moveOffsets.push_back(tuple<int,int>(-2,1));
    moveOffsets.push_back(tuple<int,int>(1,-2));
    moveOffsets.push_back(tuple<int,int>(1,2));
    moveOffsets.push_back(tuple<int,int>(2,-1));
    moveOffsets.push_back(tuple<int,int>(2,1));

    for(auto[row,col] : moveOffsets){
        int newX = x + row;
```

```
       int newY = y + col;
       if( legalCoord(newX, boardSize) && legalCoord(newY,
boardSize)){
           newMoves.push_back(tuple<int, int>(newX, newY));
       }
   }
   return newMoves;
}
Graph<int> KnightGraph(int boardSize){
   Graph<int> g;
   for(int row=0; row<boardSize; row++){
       for(int col=0; col<boardSize; col++){
           int nodeIdOrigin = positionToId(row, col, boardSize);
           list<tuple<int, int>> newPositions =
generateLegalMoves(row, col, boardSize);
           for(auto[r,c] : newPositions){
               int nodeIdDestination = positionToId(r, c,
boardSize);
               g.addEdge(nodeIdOrigin, nodeIdDestination);
           }
       }
   }
   return g;
}
```

Algoritmo gira del caballo

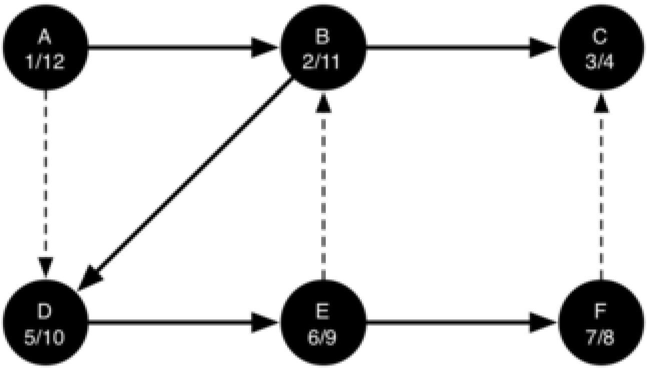
```
vector<Vertex<int>*> orderByAvailable(Vertex<int>* vertex){
   vector<Vertex<int>*> result;
   vector<tuple<int, Vertex<int>*>> listWeight;
   for(Edge<int>* v : vertex->connectedTo){
       if(v->to->color == 'w'){
           int c = 0;
           for(Edge<int>* w : v->to->connectedTo){
               if(w->to->color == 'w'){
                   c++;
               }
           }
           listWeight.push_back(tuple<int, Vertex<int>*>(c,
v->to));
       }
   }
   sort(listWeight.begin(), listWeight.end(), [](tuple<int,
Vertex<int>*> t1, tuple<int, Vertex<int>*>t2){
       return get<0>(t1) < get<0>(t2);
   });
   for(auto[c, v] : listWeight){
       result.push_back(v);
   }
   return result;
}
```

```
bool knightTour(int n, vector<int> &path, Vertex<int>* u, int
limit){
   bool done;
   u->color = 'g';
   path.push_back(u->data);
   if(n < limit){
       vector<Vertex<int>*> neighbors = orderByAvailable(u);
       int i = 0;
       done = false;
       while(i < neighbors.size() && !done){
           if( neighbors[i]->color == 'w' ){
               done = knightTour(n+1, path, neighbors[i], limit);
           }
           i++;
       }
       if(!done){
           path.pop_back();
           u->color = 'w';
       }
   }else{
       done = true;
   }
   return done;
}
```

main

```
int main() {
   Graph<int> g;
   int boardSize = 8;
   g = KnightGraph(boardSize);
   vector<int> path;
   bool done = knightTour(0, path, g.getVertex(4),
boardSize*boardSize-1);
   if(done) {
       for (int value: path) {
           cout << value << "\t ";
       }
   }else{
       cout << "No pudo alcanzar una solución";
   }

   return 0;
}
```

Búsqueda en profundidad general



Funciones

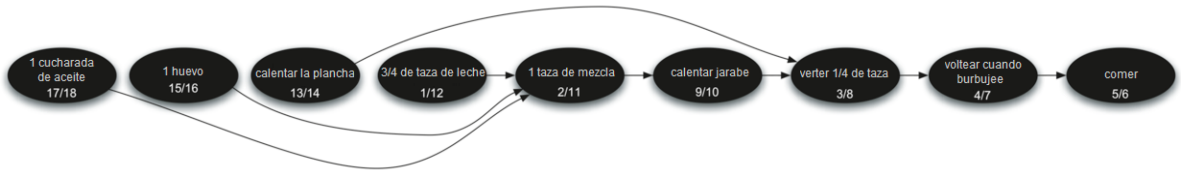
```
void traversal(Vertex<string>* vertex){
   while(vertex->predecessor){
       cout << vertex->data << endl;
       vertex = vertex->predecessor;
   }
   cout << vertex->data << endl;
}

void bep(Vertex<string>* curVertex, int &tiempo){
   curVertex->color = 'g';
   tiempo++;
   curVertex->discovery = tiempo;
   for(Edge<string>* neighbor: curVertex->getConnectedTo()){
       if(neighbor->to->color == 'w'){
           neighbor->to->predecessor = curVertex;
           bep(neighbor->to, tiempo);
       }
   }
   curVertex->color = 'b';
   tiempo++;
   curVertex->finish = tiempo;
}

void bep(Graph<string> &graph){
   int tiempo = 0;
   for(Vertex<string>* v: graph.vertexList){
       if(v->color == 'w'){
           bep(v, tiempo);
       }
   }
}
```

main

```
int main() {
   Graph<string> g;
   g.addVertex("A");
```

```
    g.addVertex("B");
    g.addVertex("C");
    g.addVertex("D");
    g.addVertex("E");
    g.addVertex("F");

    g.addEdge("A", "B");
    g.addEdge("A", "D");

    g.addEdge("B", "C");
    g.addEdge("B", "D");

    g.addEdge("D", "E");

    g.addEdge("E", "B");
    g.addEdge("E", "F");

    g.addEdge("F", "C");

    cout << "Realizando BEP desde vertice" << endl;
    bep(g);
    cout << "Imprimiendo recorrido" << endl;
    traversal(g.getVertex("F")); // palabra final*/
    return 0;
}
```

Ordenamiento topológico


1 cucharada de aceite 17/18
1 huevo 15/16
calentar la plancha 13/14
3/4 de taza de leche 1/12
1 taza de mezcla 2/11
calentar jarabe 9/10
verter 1/4 de taza 3/8
voltear cuando burbujee 4/7
comer 5/6

Funciones

```
//Las mismas que en búsqueda de profundidad general (anterior).
```

main

```
int main() {
    Graph<string> g;
    g.addVertex("3/4 de taza de leche");
    g.addVertex("calentar la plancha");
    g.addVertex("1 huevo");
    g.addVertex("1 cucharada de aceite");
    g.addVertex("1 taza de mezcla");
    g.addVertex("calentar jarabe");
    g.addVertex("verter 1/4 de taza");
    g.addVertex("voltear cuando burbujee");
    g.addVertex("comer");
    g.addEdge("3/4 de taza de leche","1 taza de mezcla");
    g.addEdge("1 huevo","1 taza de mezcla");
    g.addEdge("1 cucharada de aceite","1 taza de mezcla");
    g.addEdge("1 taza de mezcla","verter 1/4 de taza");
    g.addEdge("1 taza de mezcla","calentar jarabe");
    g.addEdge("calentar la plancha","verter 1/4 de taza");
    g.addEdge("verter 1/4 de taza","voltear cuando burbujee");
    g.addEdge("voltear cuando burbujee","comer");
    g.addEdge("calentar jarabe","comer");

    bep(g);

    vector<Vertex<string>*> vertexList = g.vertexList;
    sort(vertexList.begin(), vertexList.end(), [](Vertex<string>*
a, Vertex<string>* b){
        return a->finish > b->finish;
    });
    for(Vertex<string>* v : vertexList){
        cout << v->data << endl;
    }

    cout << "Recorrido" << endl;
    traversal(g.getVertex("comer"));

    return 0;
}
```