ALGORITMOS Y ESTRUCTURAS DE DATOS

GRAPHS PROJECT

Integrantes:

José Miguel Sanchez Atahualpa

Milagros Valeria Yupanqui López

Julisa Lapa Romero

ÍNDICE:

1. Motivación

2. Estructura y Métodos del Grafo

3. Implementación del Parser

4. Algoritmos:

   - Kruskal
   - Prim
   - Breadth First Search
   - Depth First Search
   - Best First Search
   - Dijkstra
   - Floyd - Warshall
   - Bellman - Ford
   - A *

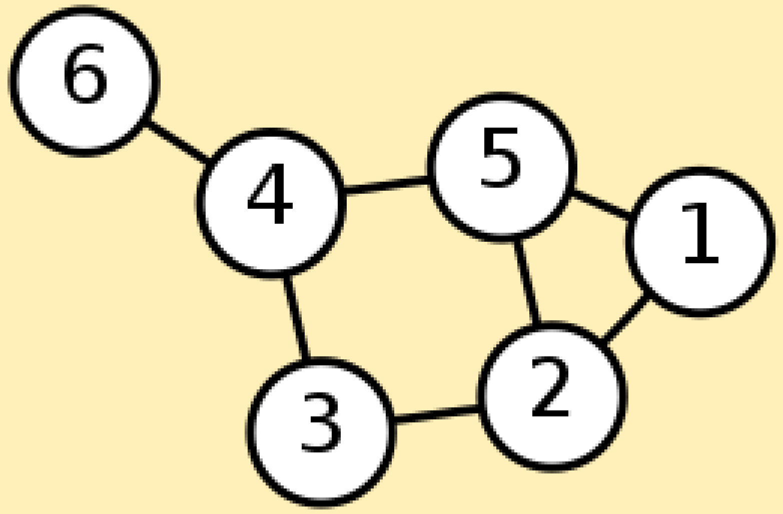
5. Anexos

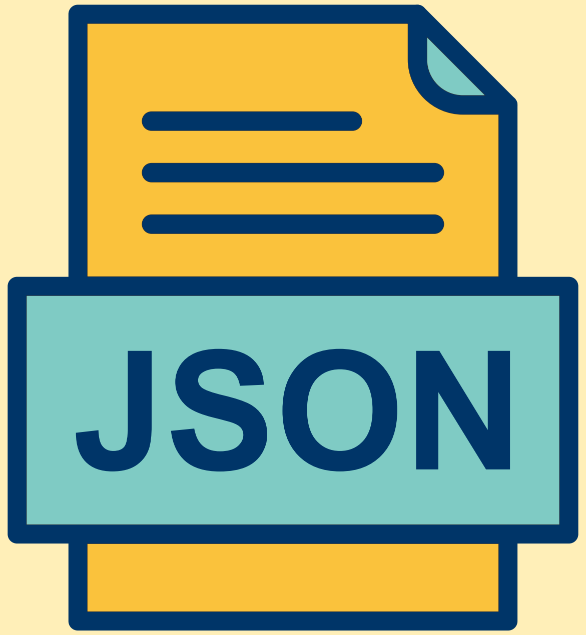
1.
MOTIVACIÓN

- **Implementar una estructura de datos de grafo con los métodos y algoritmos especificados por el docente.**

- La estructura Grafo debe ser dinámica(inserciones, eliminaciones, búsquedas, ...). Además, se debe implementar grafos Dirigidos y No Dirigidos. Finalmente, no se deben considerar loops ni multi-aristas.

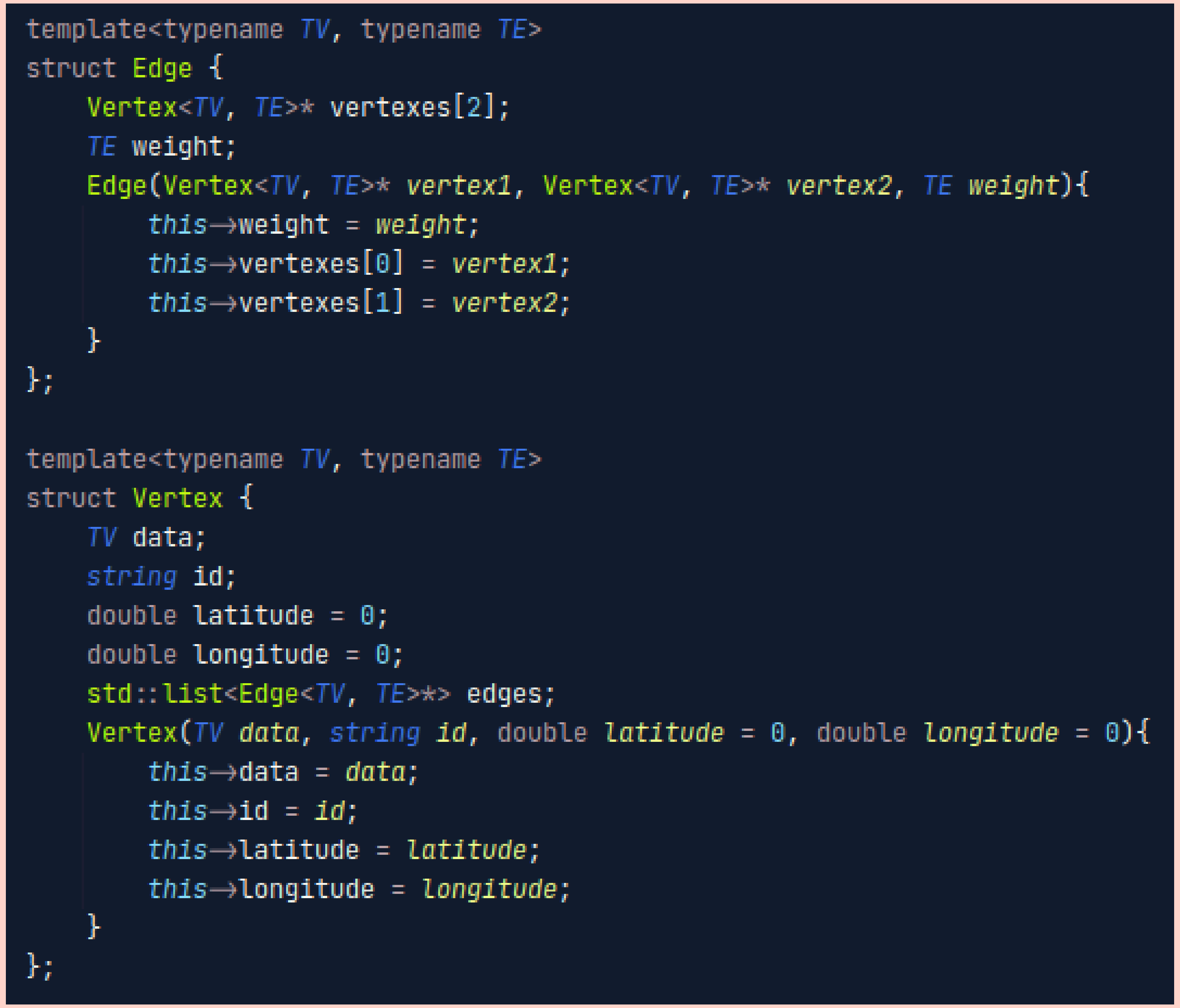
- **Implementar un file parser, que a partir de un archivo JSON construya un grafo de los aeropuertos de Perú y del mundo.**





2.
ESTRUCTURA Y MÉTODOS DEL GRAFO

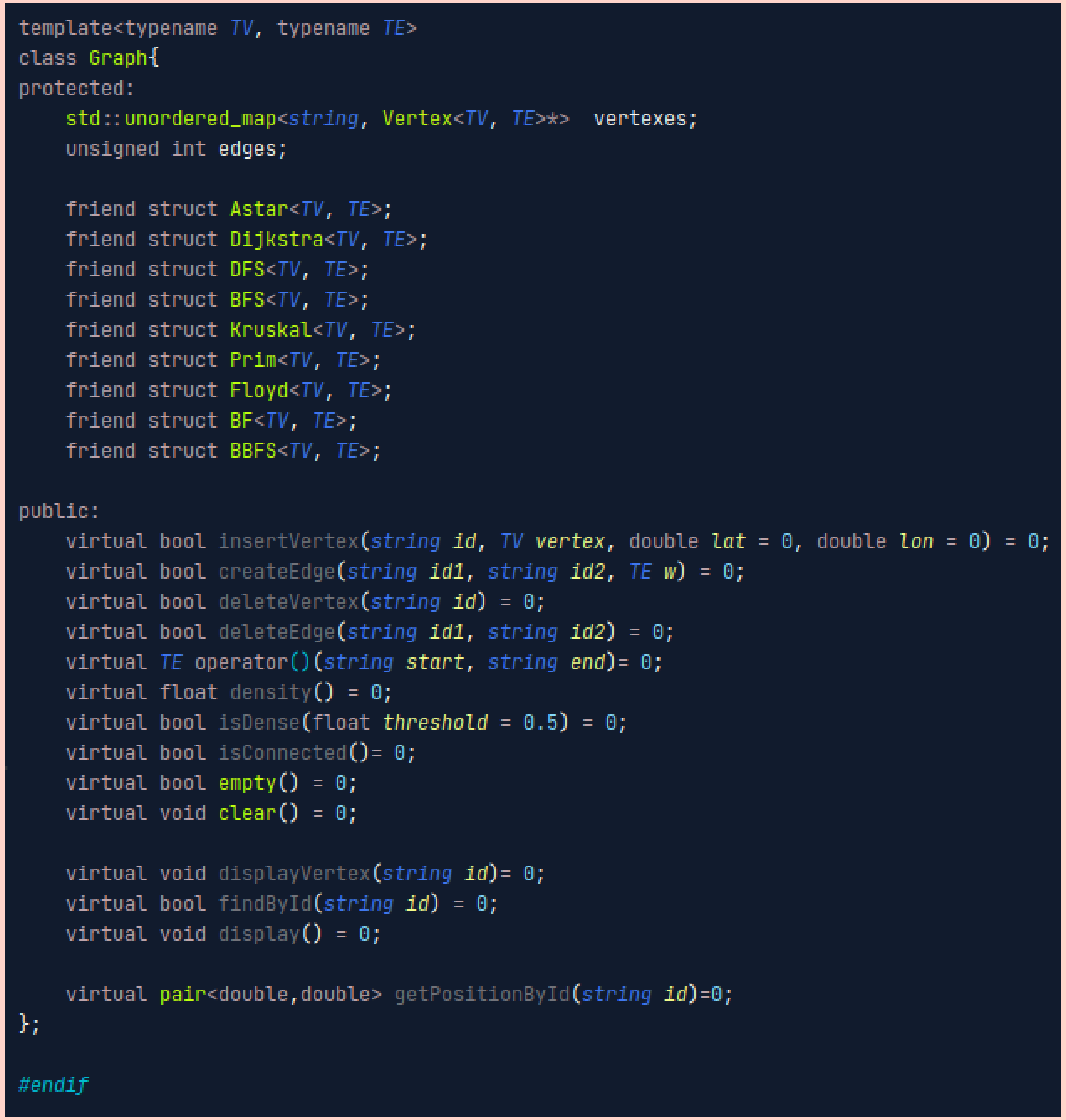
Componentes del Grafo:



```
template<typename TV, typename TE>
struct Edge {
    Vertex<TV, TE>* vertexes[2];
    TE weight;
    Edge(Vertex<TV, TE>* vertex1, Vertex<TV, TE>* vertex2, TE weight){
        this->weight = weight;
        this->vertexes[0] = vertex1;
        this->vertexes[1] = vertex2;
    }
};

template<typename TV, typename TE>
struct Vertex {
    TV data;
    string id;
    double latitude = 0;
    double longitude = 0;
    std::list<Edge<TV, TE>*> edges;
    Vertex(TV data, string id, double latitude = 0, double longitude = 0){
        this->data = data;
        this->id = id;
        this->latitude = latitude;
        this->longitude = longitude;
    }
};
```

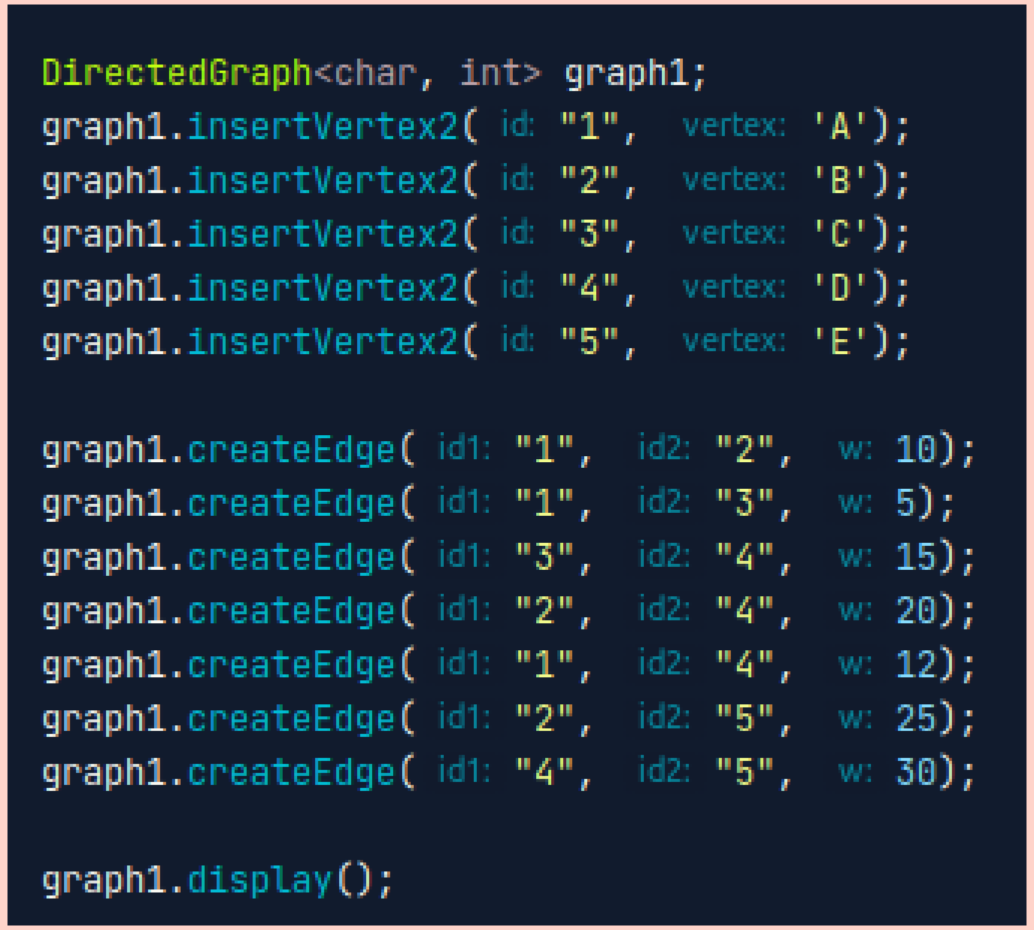
Componentes del Grafo:

Clase Padre :



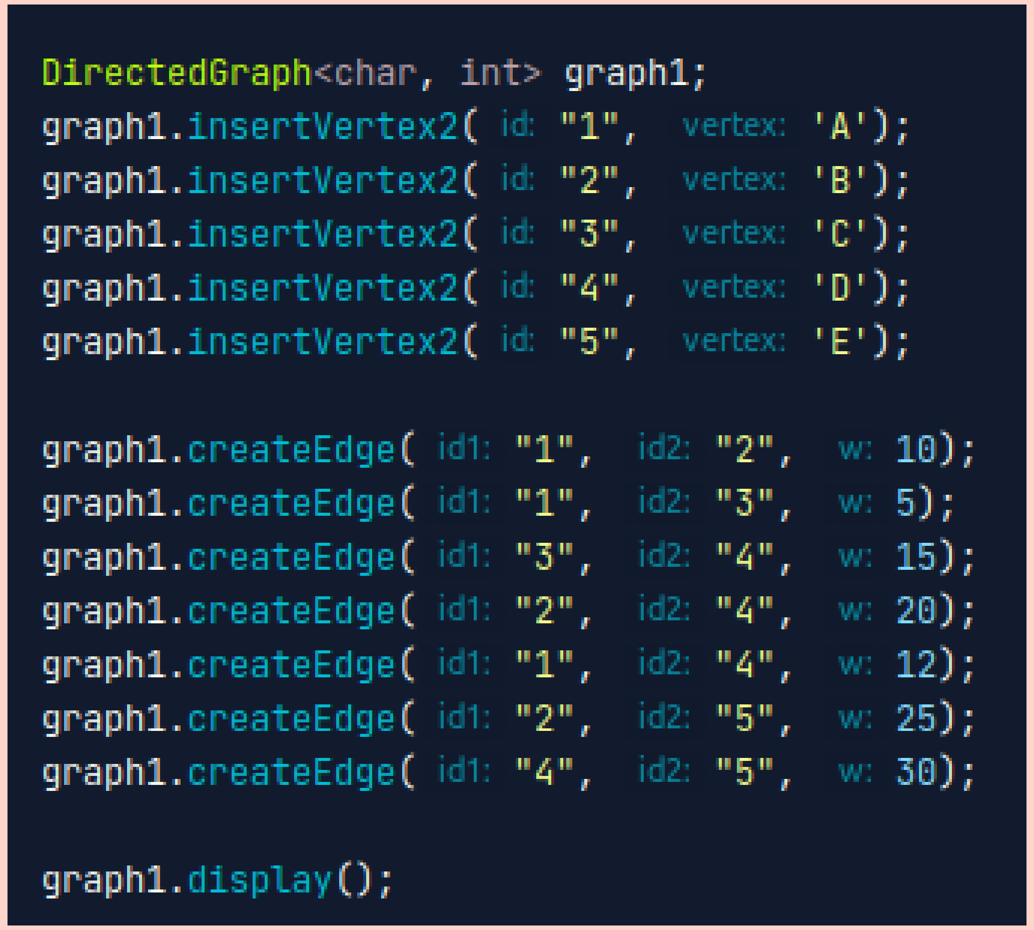
```
template<typename TV, typename TE>
class Graph{
protected:
    std::unordered_map<string, Vertex<TV, TE>*>  vertexes;
    unsigned int edges;

    friend struct Astar<TV, TE>;
    friend struct Dijkstra<TV, TE>;
    friend struct DFS<TV, TE>;
    friend struct BFS<TV, TE>;
    friend struct Kruskal<TV, TE>;
    friend struct Prim<TV, TE>;
    friend struct Floyd<TV, TE>;
    friend struct BF<TV, TE>;
    friend struct BBFS<TV, TE>;

public:
    virtual bool insertVertex(string id, TV vertex, double lat = 0, double lon = 0) = 0;
    virtual bool createEdge(string id1, string id2, TE w) = 0;
    virtual bool deleteVertex(string id) = 0;
    virtual bool deleteEdge(string id1, string id2) = 0;
    virtual TE operator()(string start, string end)= 0;
    virtual float density() = 0;
    virtual bool isDense(float threshold = 0.5) = 0;
    virtual bool isConnected()= 0;
    virtual bool empty() = 0;
    virtual void clear() = 0;

    virtual void displayVertex(string id)= 0;
    virtual bool findById(string id) = 0;
    virtual void display() = 0;

    virtual pair<double,double> getPositionById(string id)=0;
};

#endif
```

Grafo Dirigido:

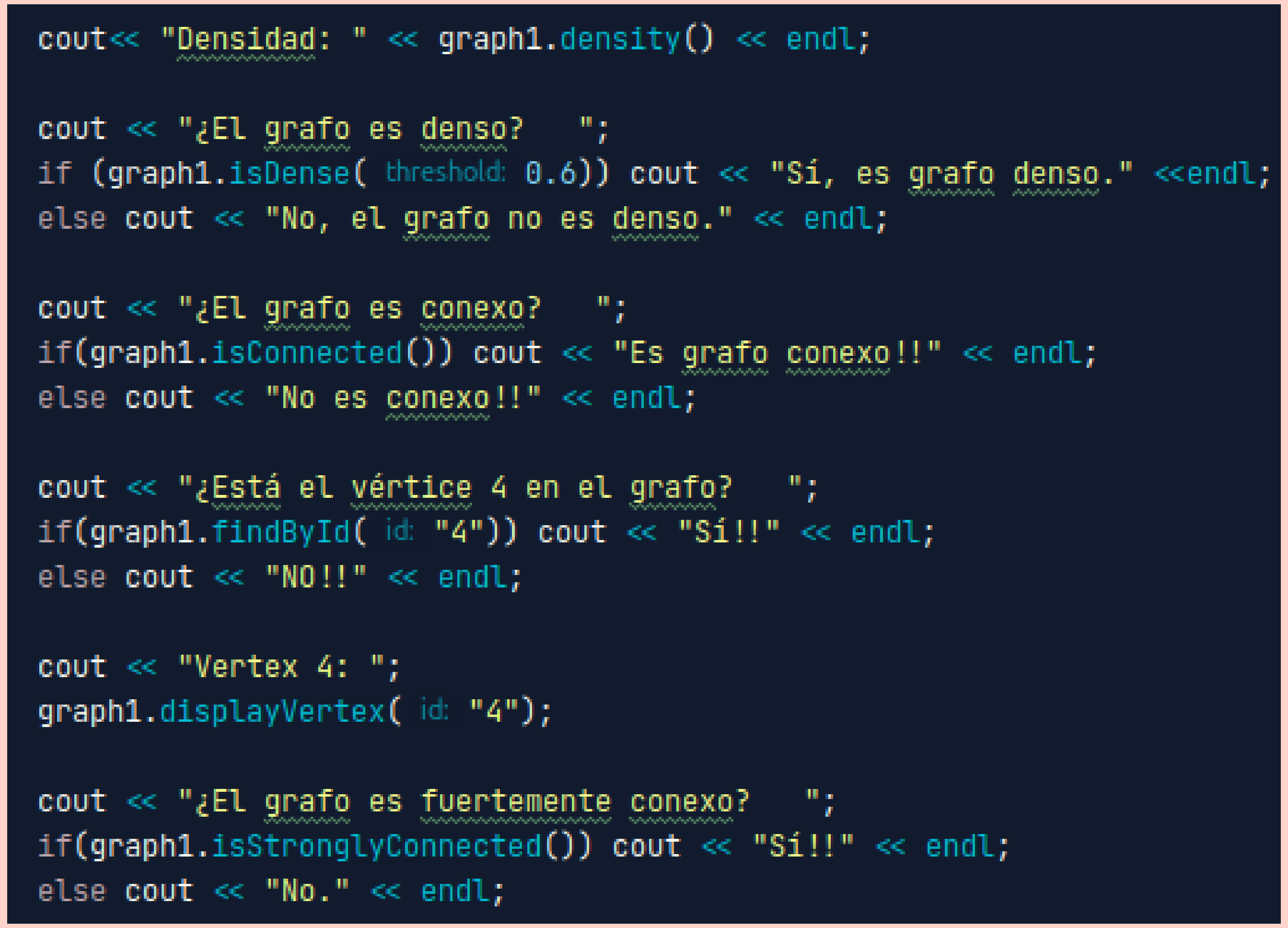


```
DirectedGraph<char, int> graph1;
graph1.insertVertex2( id: "1",  vertex: 'A');
graph1.insertVertex2( id: "2",  vertex: 'B');
graph1.insertVertex2( id: "3",  vertex: 'C');
graph1.insertVertex2( id: "4",  vertex: 'D');
graph1.insertVertex2( id: "5",  vertex: 'E');

graph1.createEdge( id1: "1",  id2: "2",  w: 10);
graph1.createEdge( id1: "1",  id2: "3",  w: 5);
graph1.createEdge( id1: "3",  id2: "4",  w: 15);
graph1.createEdge( id1: "2",  id2: "4",  w: 20);
graph1.createEdge( id1: "1",  id2: "4",  w: 12);
graph1.createEdge( id1: "2",  id2: "5",  w: 25);
graph1.createEdge( id1: "4",  id2: "5",  w: 30);

graph1.display();
```

Grafo Dirigido:

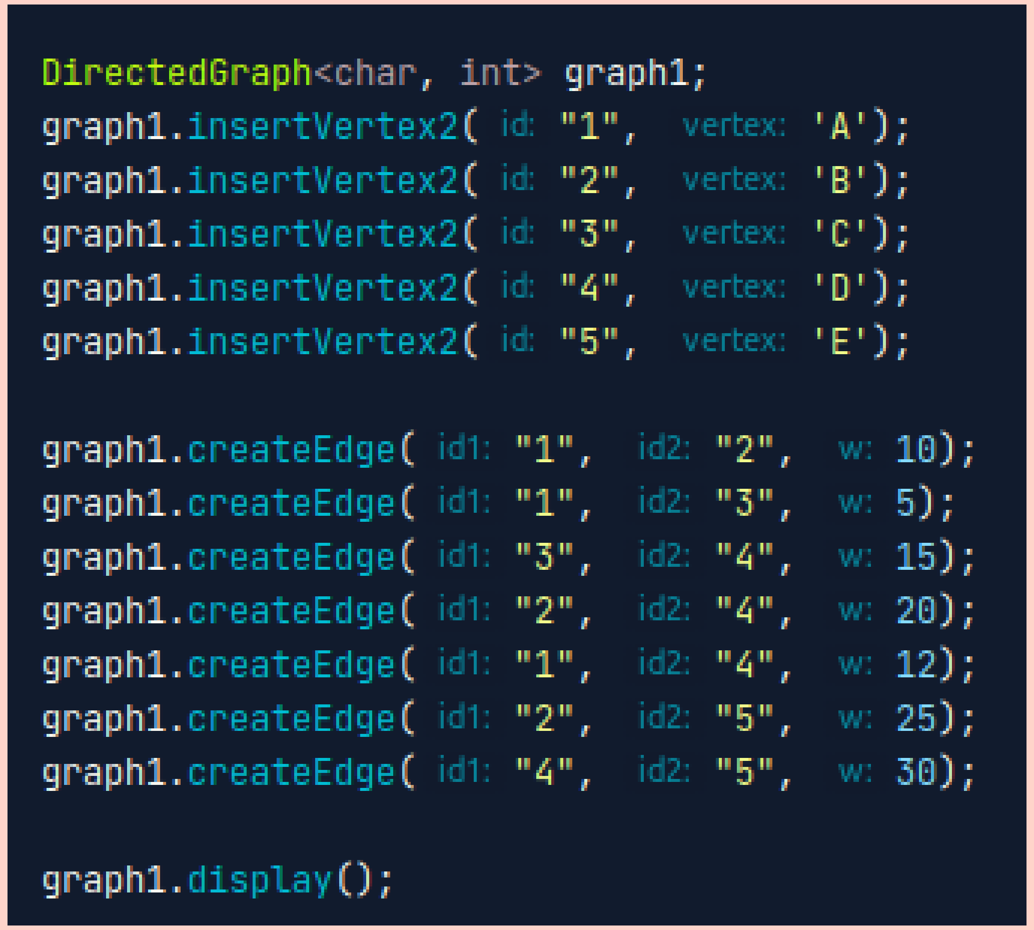


```
DirectedGraph<char, int> graph1;
graph1.insertVertex2( id: "1",  vertex: 'A');
graph1.insertVertex2( id: "2",  vertex: 'B');
graph1.insertVertex2( id: "3",  vertex: 'C');
graph1.insertVertex2( id: "4",  vertex: 'D');
graph1.insertVertex2( id: "5",  vertex: 'E');

graph1.createEdge( id1: "1",  id2: "2",  w: 10);
graph1.createEdge( id1: "1",  id2: "3",  w: 5);
graph1.createEdge( id1: "3",  id2: "4",  w: 15);
graph1.createEdge( id1: "2",  id2: "4",  w: 20);
graph1.createEdge( id1: "1",  id2: "4",  w: 12);
graph1.createEdge( id1: "2",  id2: "5",  w: 25);
graph1.createEdge( id1: "4",  id2: "5",  w: 30);

graph1.display();
```

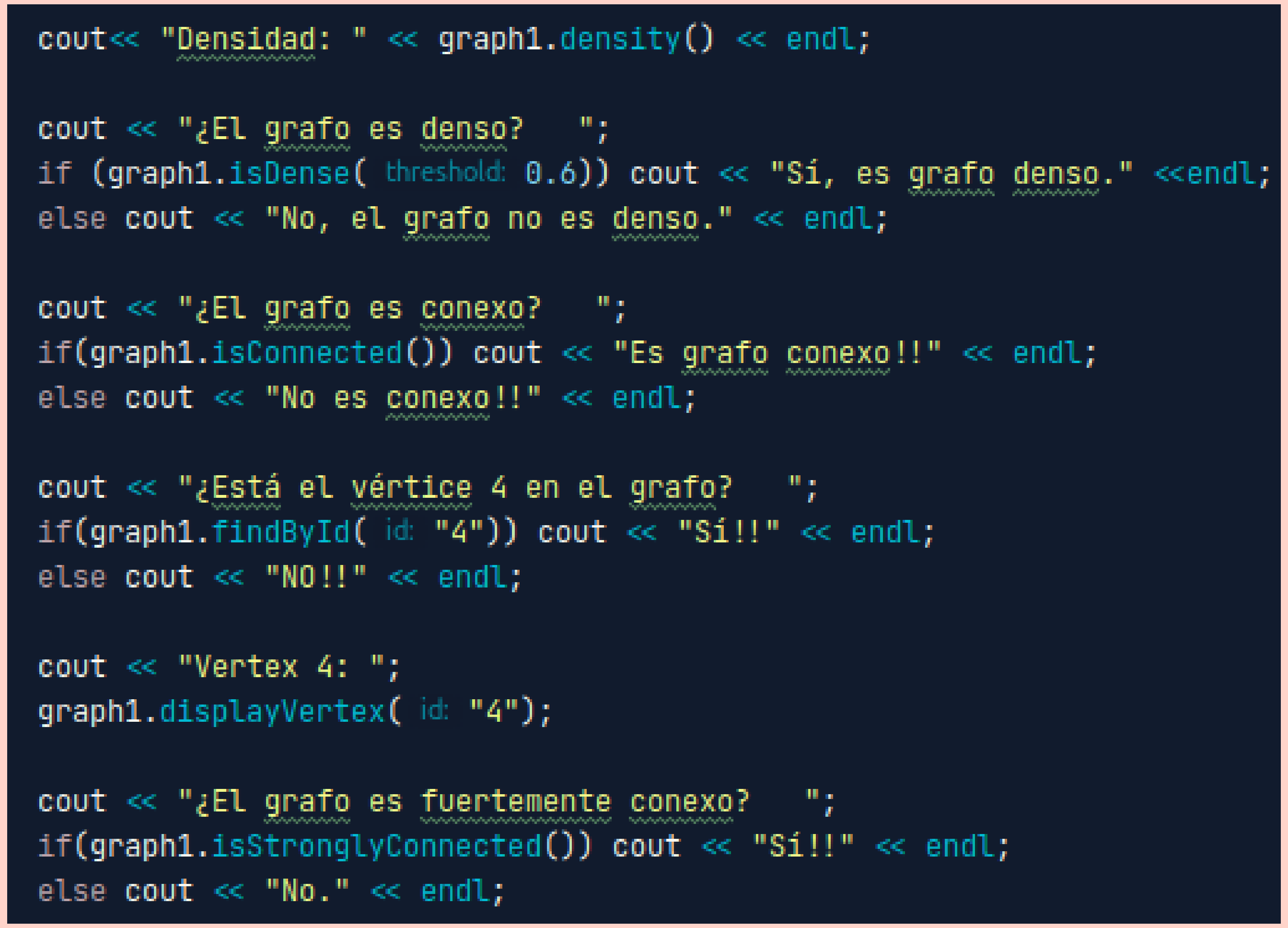


```
cout<< "Densidad: " << graph1.density() << endl;

cout << "¿El grafo es denso?   ";
if (graph1.isDense( threshold: 0.6)) cout << "Sí, es grafo denso." <<endl;
else cout << "No, el grafo no es denso." << endl;

cout << "¿El grafo es conexo?   ";
if(graph1.isConnected()) cout << "Es grafo conexo!!" << endl;
else cout << "No es conexo!!" << endl;

cout << "¿Está el vértice 4 en el grafo?   ";
if(graph1.findById( id: "4")) cout << "Sí!!" << endl;
else cout << "NO!!" << endl;

cout << "Vertex 4: ";
graph1.displayVertex( id: "4");

cout << "¿El grafo es fuertemente conexo?   ";
if(graph1.isStronglyConnected()) cout << "Sí!!" << endl;
else cout << "No." << endl;
```

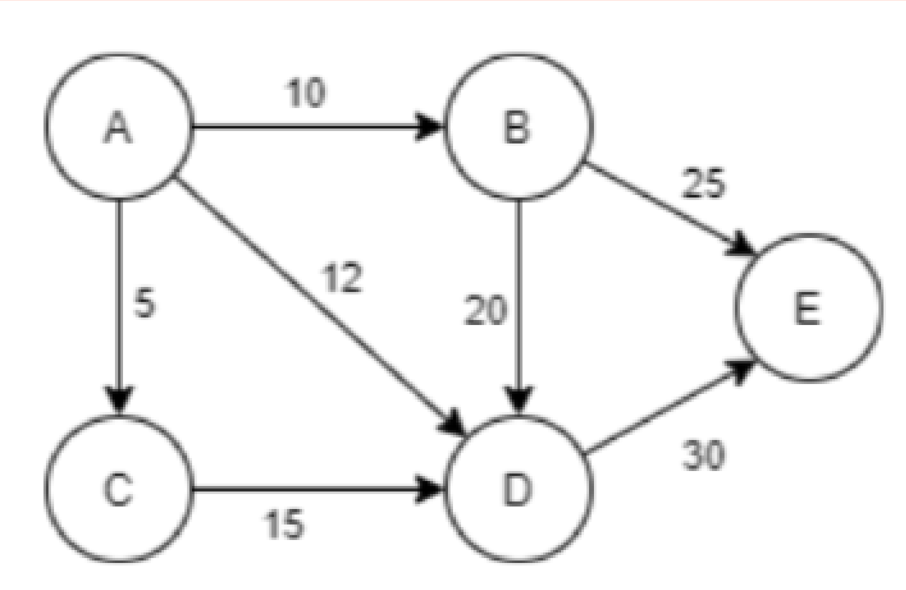
Grafo Dirigido:

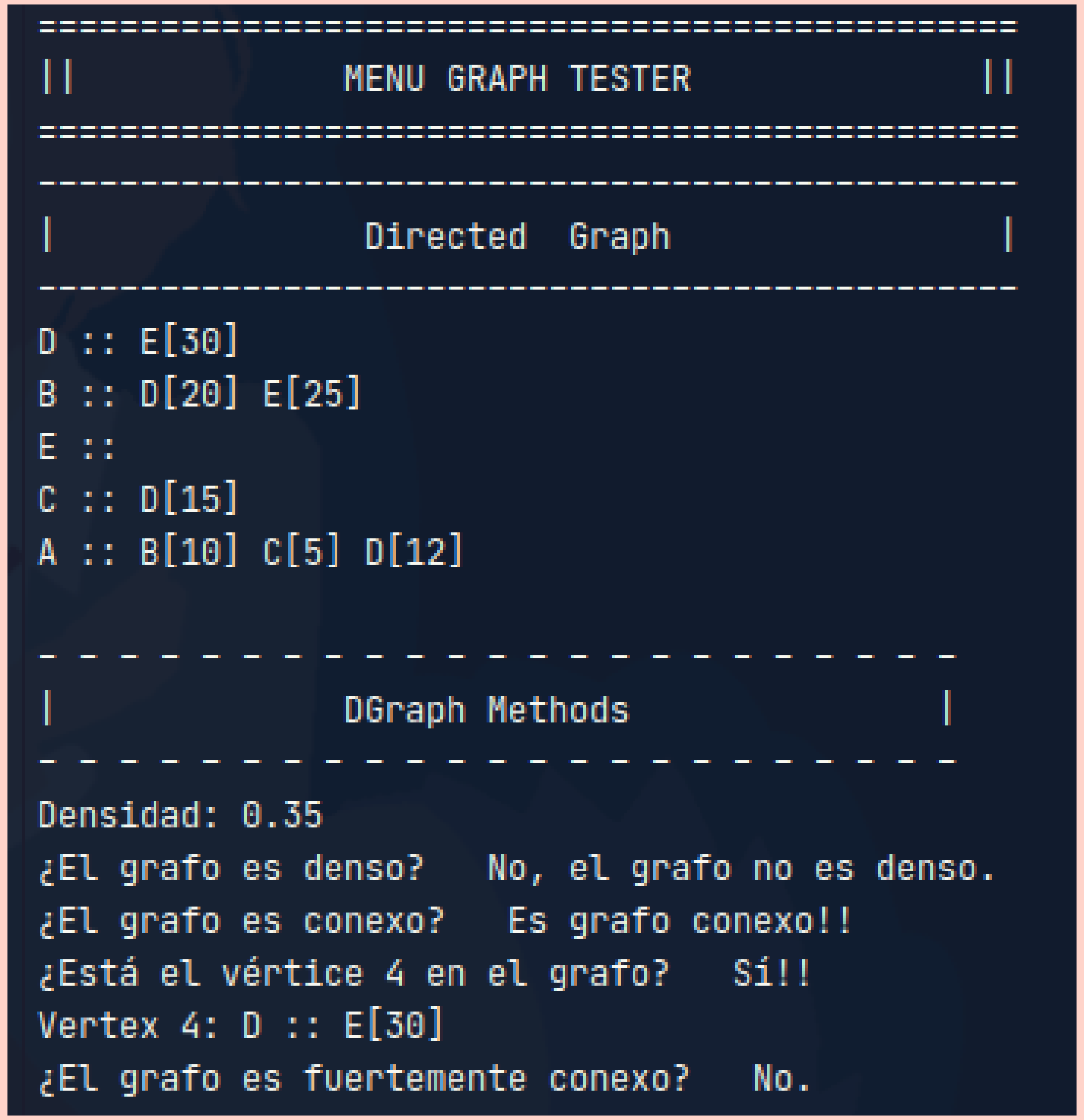


```
DirectedGraph<char, int> graph1;
graph1.insertVertex2( id: "1",  vertex: 'A');
graph1.insertVertex2( id: "2",  vertex: 'B');
graph1.insertVertex2( id: "3",  vertex: 'C');
graph1.insertVertex2( id: "4",  vertex: 'D');
graph1.insertVertex2( id: "5",  vertex: 'E');

graph1.createEdge( id1: "1",  id2: "2",  w: 10);
graph1.createEdge( id1: "1",  id2: "3",  w: 5);
graph1.createEdge( id1: "3",  id2: "4",  w: 15);
graph1.createEdge( id1: "2",  id2: "4",  w: 20);
graph1.createEdge( id1: "1",  id2: "4",  w: 12);
graph1.createEdge( id1: "2",  id2: "5",  w: 25);
graph1.createEdge( id1: "4",  id2: "5",  w: 30);

graph1.display();
```



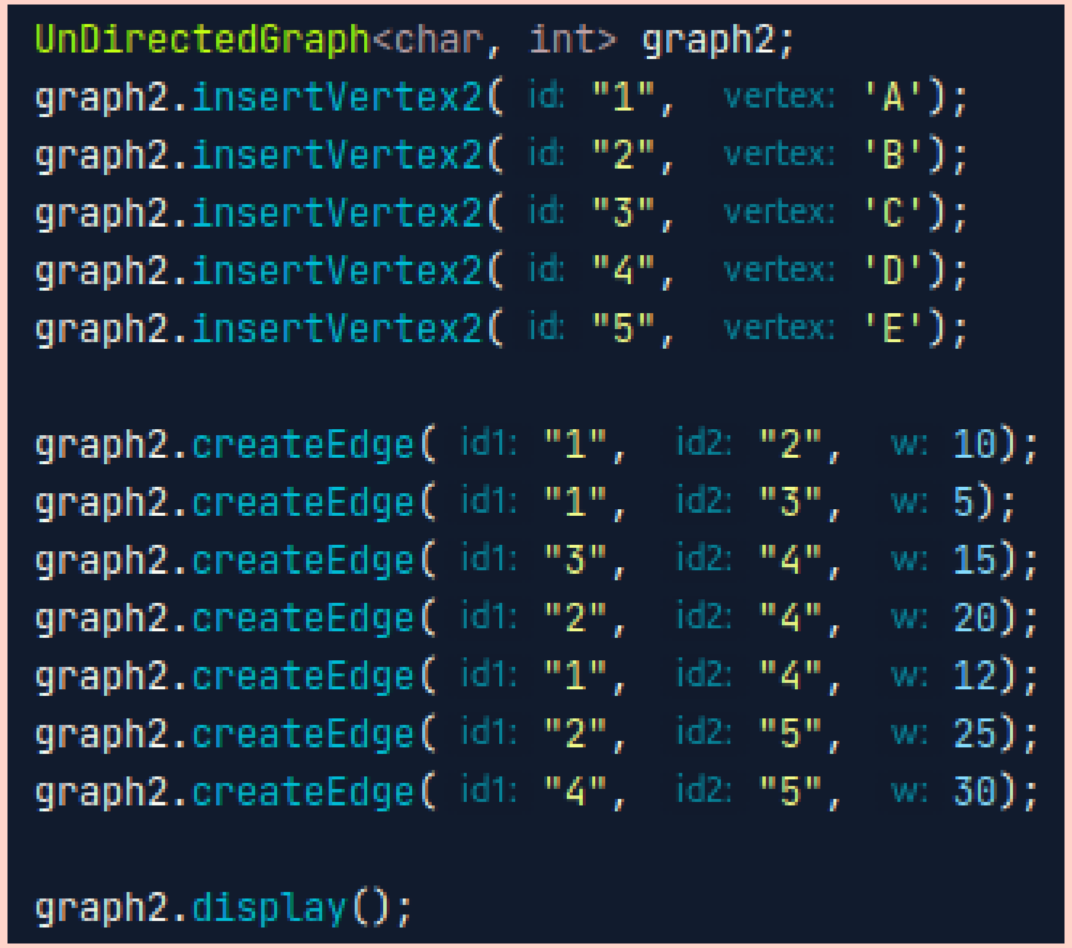
```
cout<< "Densidad: " << graph1.density() << endl;

cout << "¿El grafo es denso?   ";
if (graph1.isDense( threshold: 0.6)) cout << "Sí, es grafo denso." <<endl;
else cout << "No, el grafo no es denso." << endl;

cout << "¿El grafo es conexo?   ";
if(graph1.isConnected()) cout << "Es grafo conexo!!" << endl;
else cout << "No es conexo!!" << endl;

cout << "¿Está el vértice 4 en el grafo?   ";
if(graph1.findById( id: "4")) cout << "Sí!!" << endl;
else cout << "NO!!" << endl;

cout << "Vertex 4: ";
graph1.displayVertex( id: "4");

cout << "¿El grafo es fuertemente conexo?   ";
if(graph1.isStronglyConnected()) cout << "Sí!!" << endl;
else cout << "No." << endl;
```





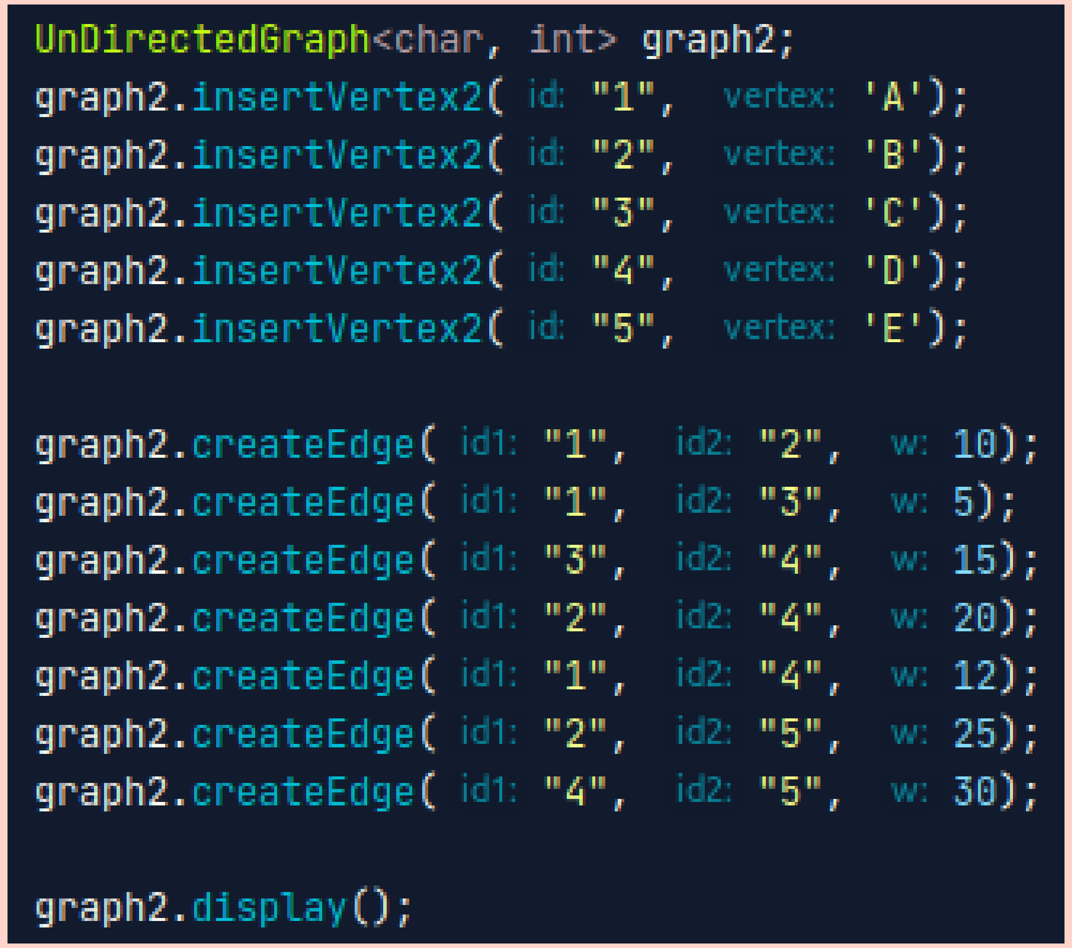
```
================================================
||                MENU GRAPH TESTER           ||
================================================
------------------------------------------------
|                 Directed  Graph              |
------------------------------------------------
D :: E[30]
B :: D[20] E[25]
E ::
C :: D[15]
A :: B[10] C[5] D[12]

- - - - - - - - - - - - - - - - - - - - - - - -
|                DGraph Methods              |
- - - - - - - - - - - - - - - - - - - - - - - -
Densidad: 0.35
¿El grafo es denso?   No, el grafo no es denso.
¿El grafo es conexo?   Es grafo conexo!!
¿Está el vértice 4 en el grafo?   Sí!!
Vertex 4: D :: E[30]
¿El grafo es fuertemente conexo?   No.
```

Grafo No Dirigido:

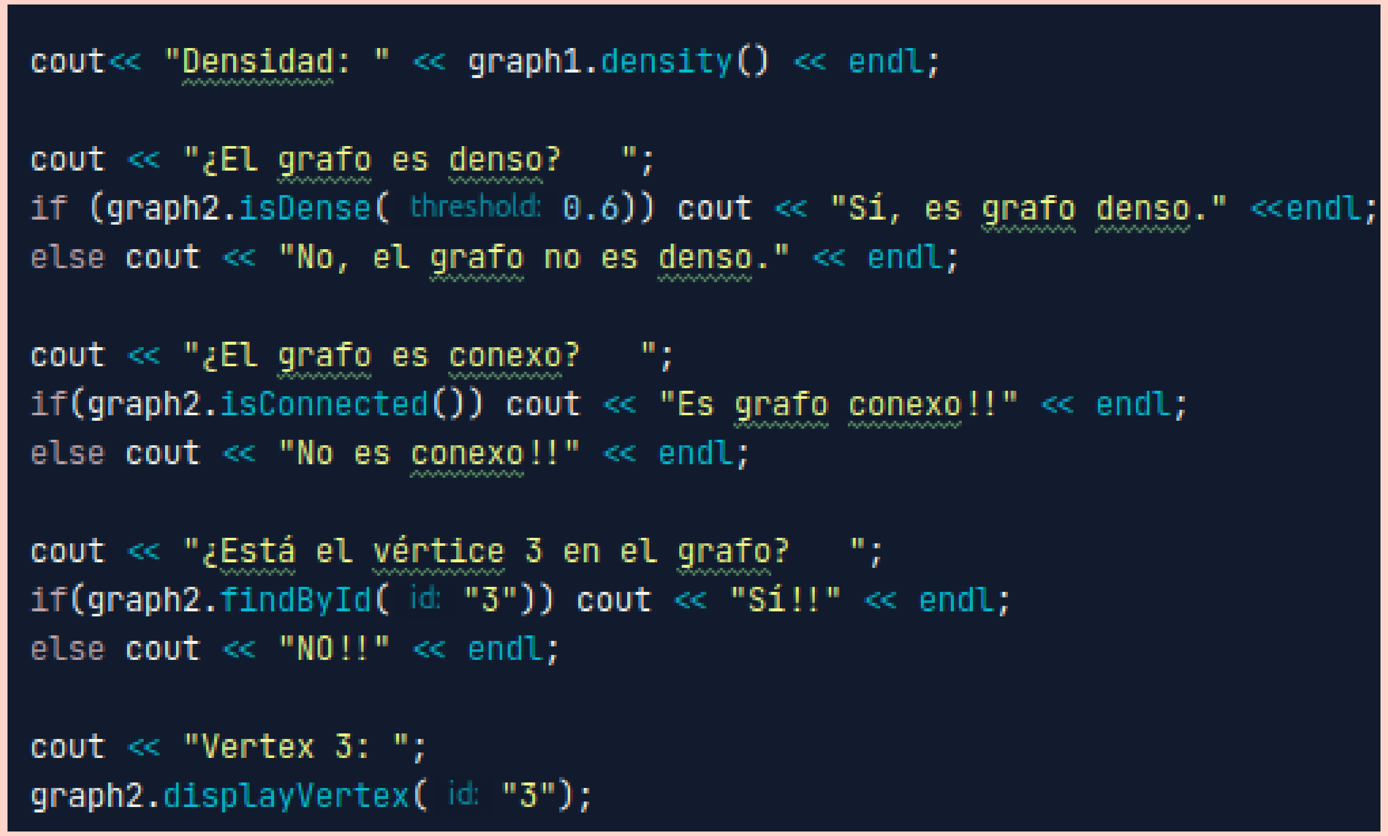


```
UnDirectedGraph<char, int> graph2;
graph2.insertVertex2( id: "1",  vertex: 'A');
graph2.insertVertex2( id: "2",  vertex: 'B');
graph2.insertVertex2( id: "3",  vertex: 'C');
graph2.insertVertex2( id: "4",  vertex: 'D');
graph2.insertVertex2( id: "5",  vertex: 'E');

graph2.createEdge( id1: "1",  id2: "2",  w: 10);
graph2.createEdge( id1: "1",  id2: "3",  w: 5);
graph2.createEdge( id1: "3",  id2: "4",  w: 15);
graph2.createEdge( id1: "2",  id2: "4",  w: 20);
graph2.createEdge( id1: "1",  id2: "4",  w: 12);
graph2.createEdge( id1: "2",  id2: "5",  w: 25);
graph2.createEdge( id1: "4",  id2: "5",  w: 30);

graph2.display();
```

Grafo No Dirigido:

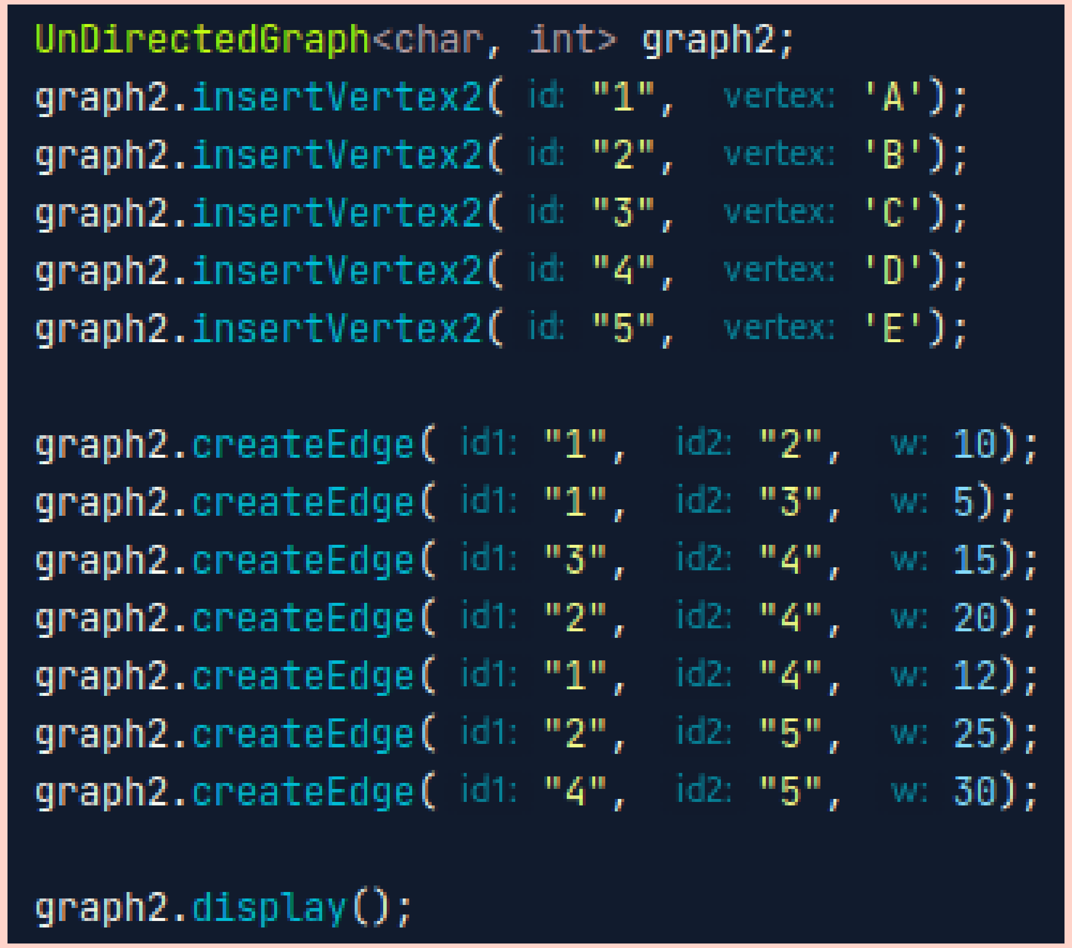


```
UnDirectedGraph<char, int> graph2;
graph2.insertVertex2( id: "1",  vertex: 'A');
graph2.insertVertex2( id: "2",  vertex: 'B');
graph2.insertVertex2( id: "3",  vertex: 'C');
graph2.insertVertex2( id: "4",  vertex: 'D');
graph2.insertVertex2( id: "5",  vertex: 'E');

graph2.createEdge( id1: "1",  id2: "2",  w: 10);
graph2.createEdge( id1: "1",  id2: "3",  w: 5);
graph2.createEdge( id1: "3",  id2: "4",  w: 15);
graph2.createEdge( id1: "2",  id2: "4",  w: 20);
graph2.createEdge( id1: "1",  id2: "4",  w: 12);
graph2.createEdge( id1: "2",  id2: "5",  w: 25);
graph2.createEdge( id1: "4",  id2: "5",  w: 30);

graph2.display();
```

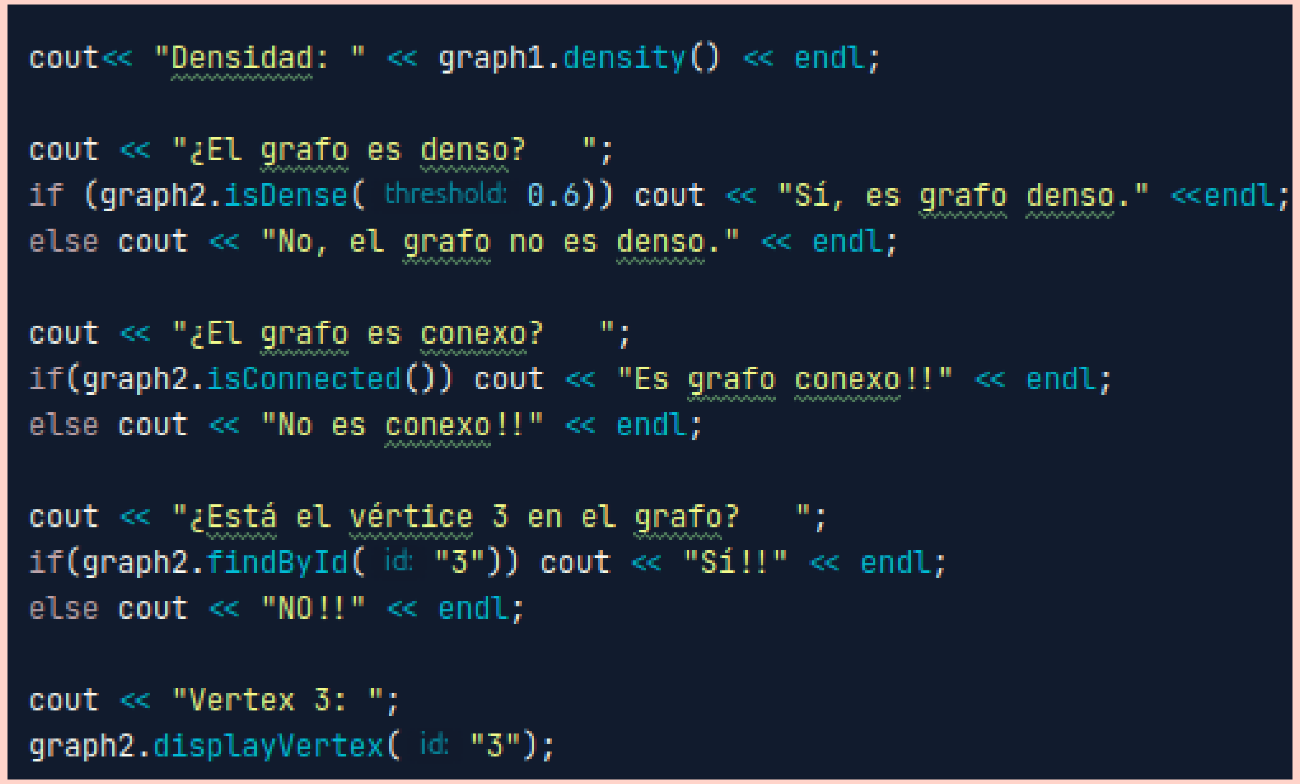


```
cout<< "Densidad: " << graph1.density() << endl;

cout << "¿El grafo es denso?   ";
if (graph2.isDense( threshold: 0.6)) cout << "Sí, es grafo denso." <<endl;
else cout << "No, el grafo no es denso." << endl;

cout << "¿El grafo es conexo?   ";
if(graph2.isConnected()) cout << "Es grafo conexo!!" << endl;
else cout << "No es conexo!!" << endl;

cout << "¿Está el vértice 3 en el grafo?   ";
if(graph2.findById( id: "3")) cout << "Sí!!" << endl;
else cout << "NO!!" << endl;

cout << "Vertex 3: ";
graph2.displayVertex( id: "3");
```

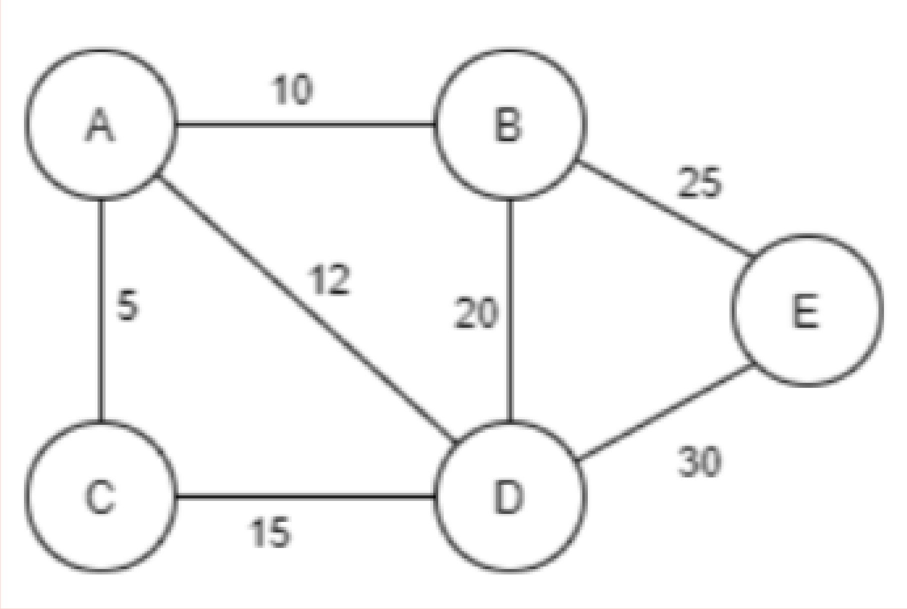
Grafo No Dirigido:

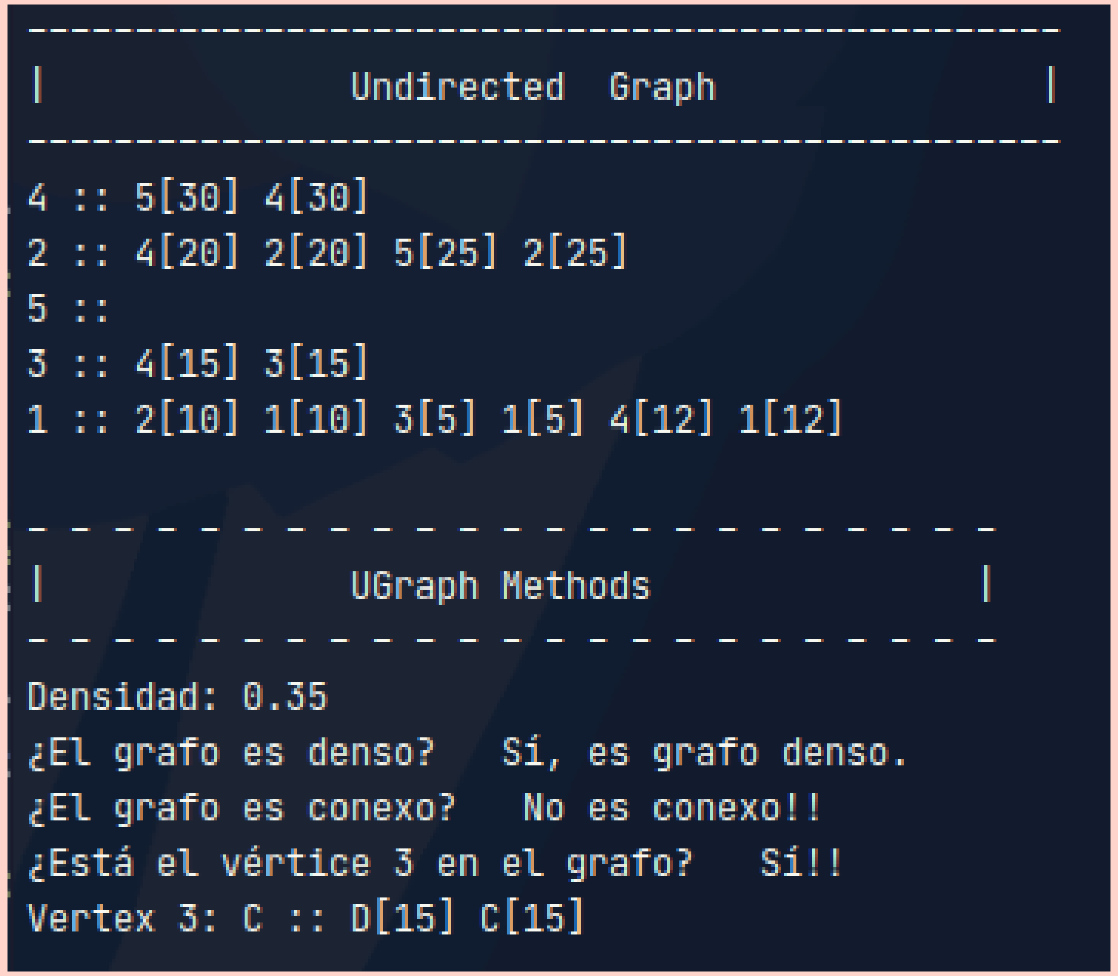


```
UnDirectedGraph<char, int> graph2;
graph2.insertVertex2( id: "1",  vertex: 'A');
graph2.insertVertex2( id: "2",  vertex: 'B');
graph2.insertVertex2( id: "3",  vertex: 'C');
graph2.insertVertex2( id: "4",  vertex: 'D');
graph2.insertVertex2( id: "5",  vertex: 'E');

graph2.createEdge( id1: "1",  id2: "2",  w: 10);
graph2.createEdge( id1: "1",  id2: "3",  w: 5);
graph2.createEdge( id1: "3",  id2: "4",  w: 15);
graph2.createEdge( id1: "2",  id2: "4",  w: 20);
graph2.createEdge( id1: "1",  id2: "4",  w: 12);
graph2.createEdge( id1: "2",  id2: "5",  w: 25);
graph2.createEdge( id1: "4",  id2: "5",  w: 30);

graph2.display();
```



```
cout<< "Densidad: " << graph1.density() << endl;

cout << "¿El grafo es denso?   ";
if (graph2.isDense( threshold: 0.6)) cout << "Sí, es grafo denso." <<endl;
else cout << "No, el grafo no es denso." << endl;

cout << "¿El grafo es conexo?   ";
if(graph2.isConnected()) cout << "Es grafo conexo!!" << endl;
else cout << "No es conexo!!" << endl;

cout << "¿Está el vértice 3 en el grafo?   ";
if(graph2.findById( id: "3")) cout << "Sí!!" << endl;
else cout << "NO!!" << endl;

cout << "Vertex 3: ";
graph2.displayVertex( id: "3");
```





```
-----------------------------------------------------
|                Undirected  Graph                  |
-----------------------------------------------------
4 :: 5[30] 4[30]
2 :: 4[20] 2[20] 5[25] 2[25]
5 ::
3 :: 4[15] 3[15]
1 :: 2[10] 1[10] 3[5] 1[5] 4[12] 1[12]


- - - - - - - - - - - - - - - - - - - - - - - - -
|                UGraph Methods                 |
- - - - - - - - - - - - - - - - - - - - - - - - -
Densidad: 0.35
¿El grafo es denso?   Sí, es grafo denso.
¿El grafo es conexo?   No es conexo!!
¿Está el vértice 3 en el grafo?   Sí!!
Vertex 3: C :: D[15] C[15]
```

3.

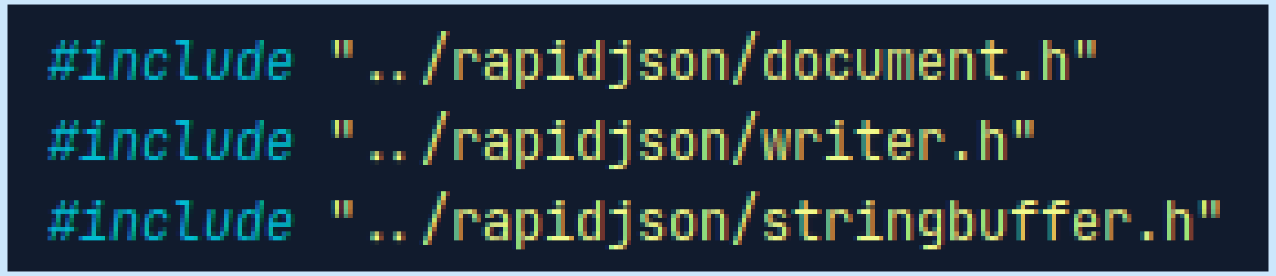
ESTRUCTURA Y MÉTODOS DEL GRAFO

PARSER: JSON File to Graph Structure

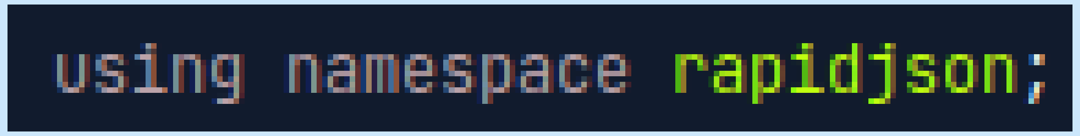
- Se utilizó la librería RapidJSON, un parser/generator de json para c++.
- Recuperado de: https://rapidjson.org/

RapidJSON logo

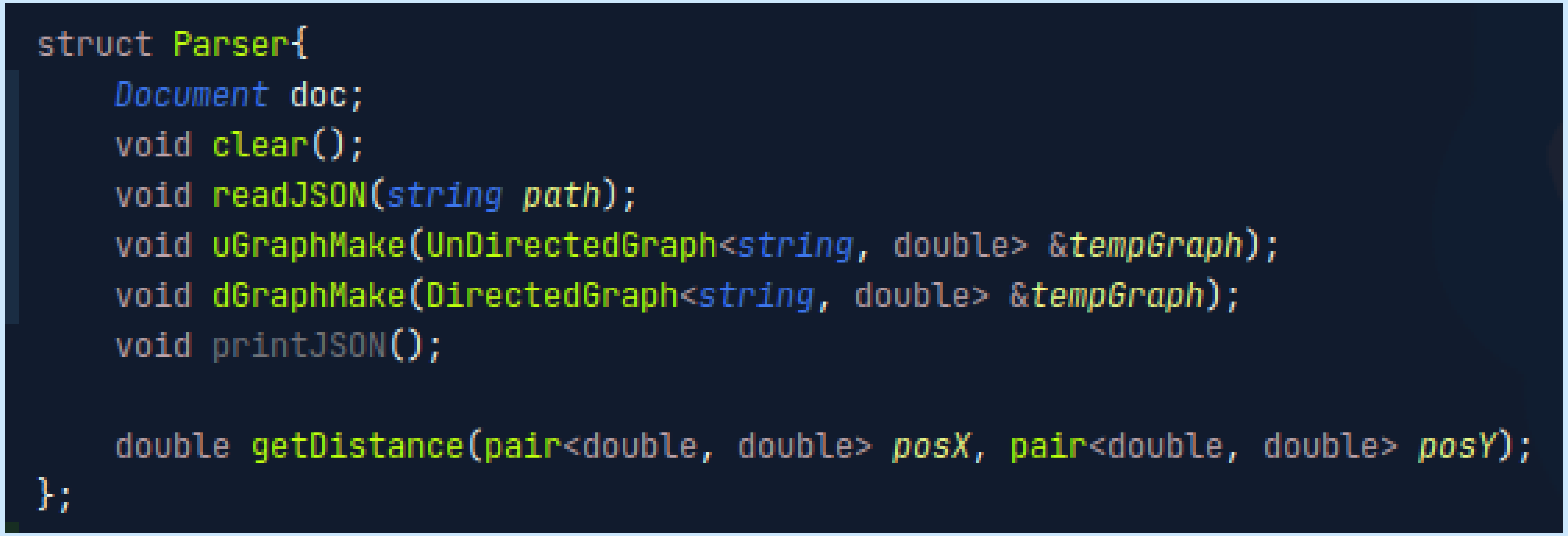
- GitHub: https://github.com/Tencent/rapidjson/tree/master/include/rapidjson



```cpp
#include "../rapidjson/document.h"
#include "../rapidjson/writer.h"
#include "../rapidjson/stringbuffer.h"
```

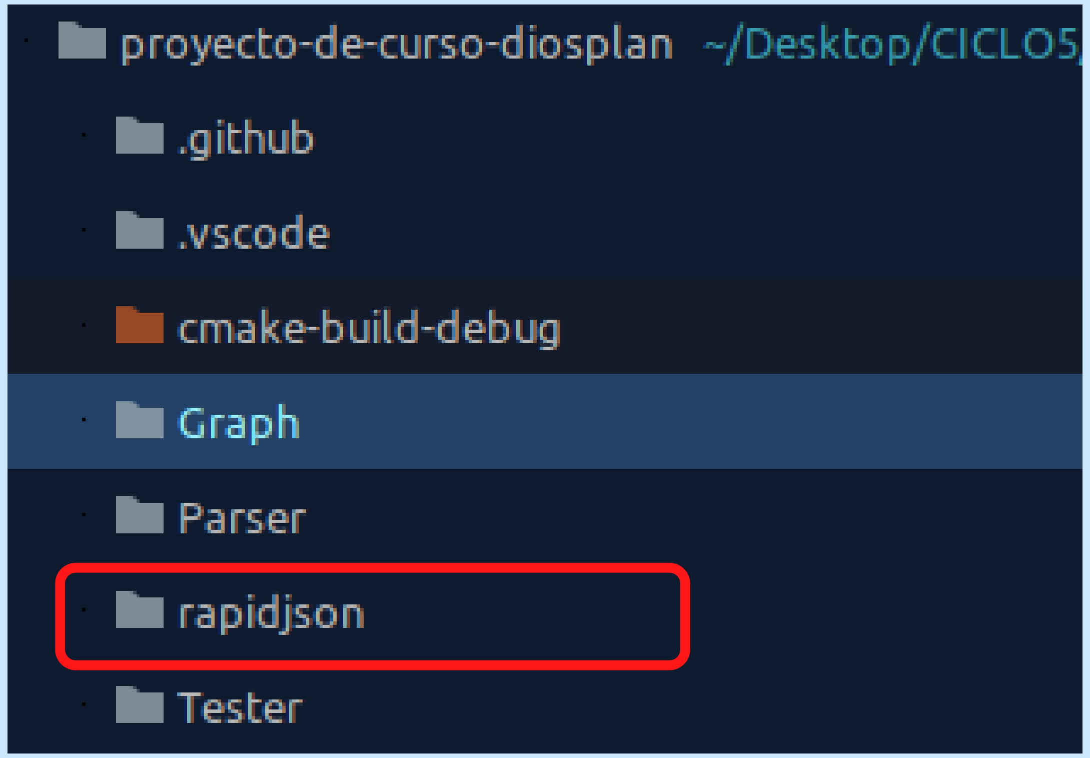


```cpp
using namespace rapidjson;
```

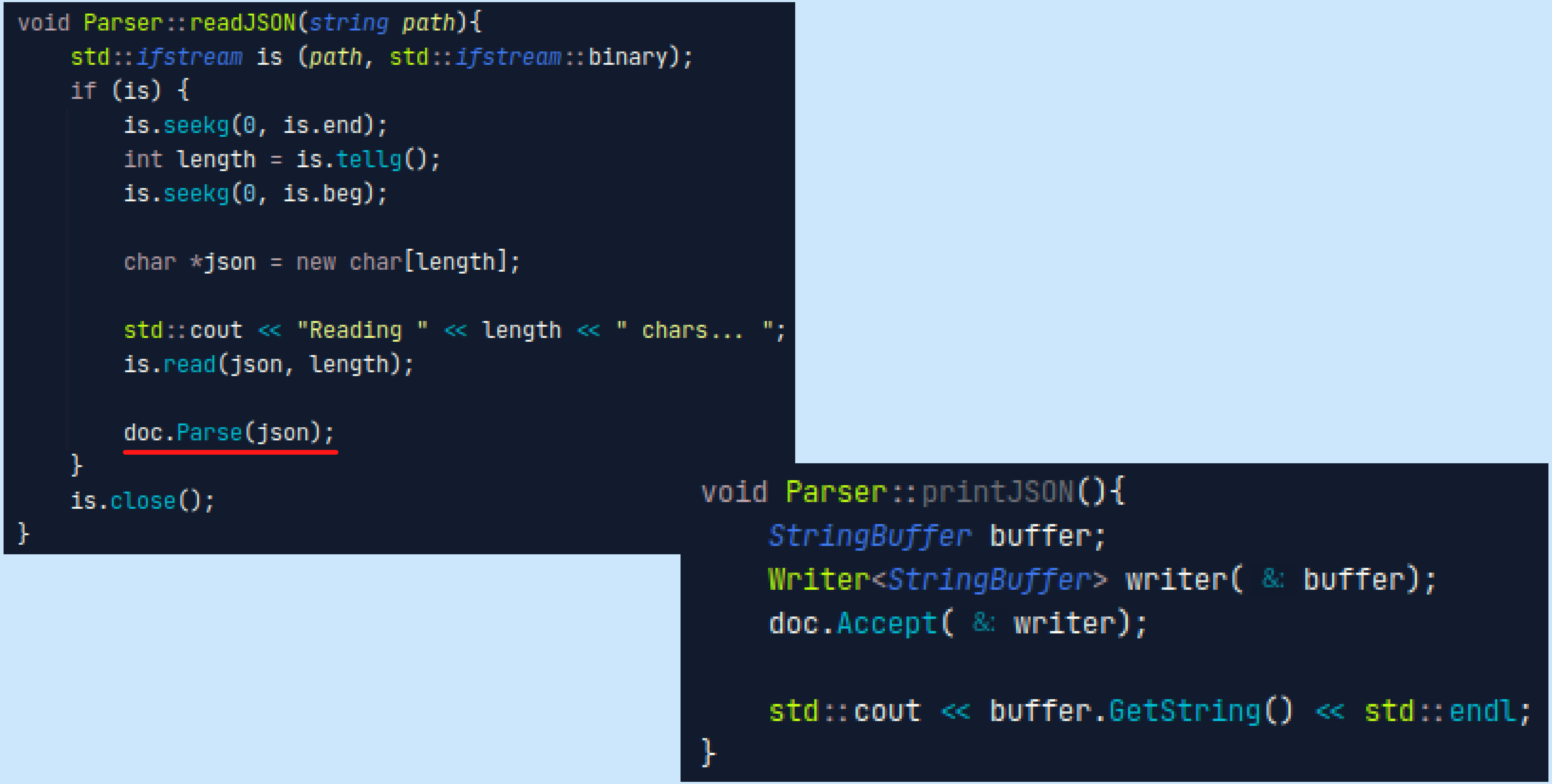


```cpp
struct Parser{
    Document doc;
    void clear();
    void readJSON(string path);
    void uGraphMake(UnDirectedGraph<string, double> &tempGraph);
    void dGraphMake(DirectedGraph<string, double> &tempGraph);
    void printJSON();

    double getDistance(pair<double, double> posX, pair<double, double> posY);
};
```

LA LIBRERÍA SE IMPORTÓ DIRECTAMENTE:

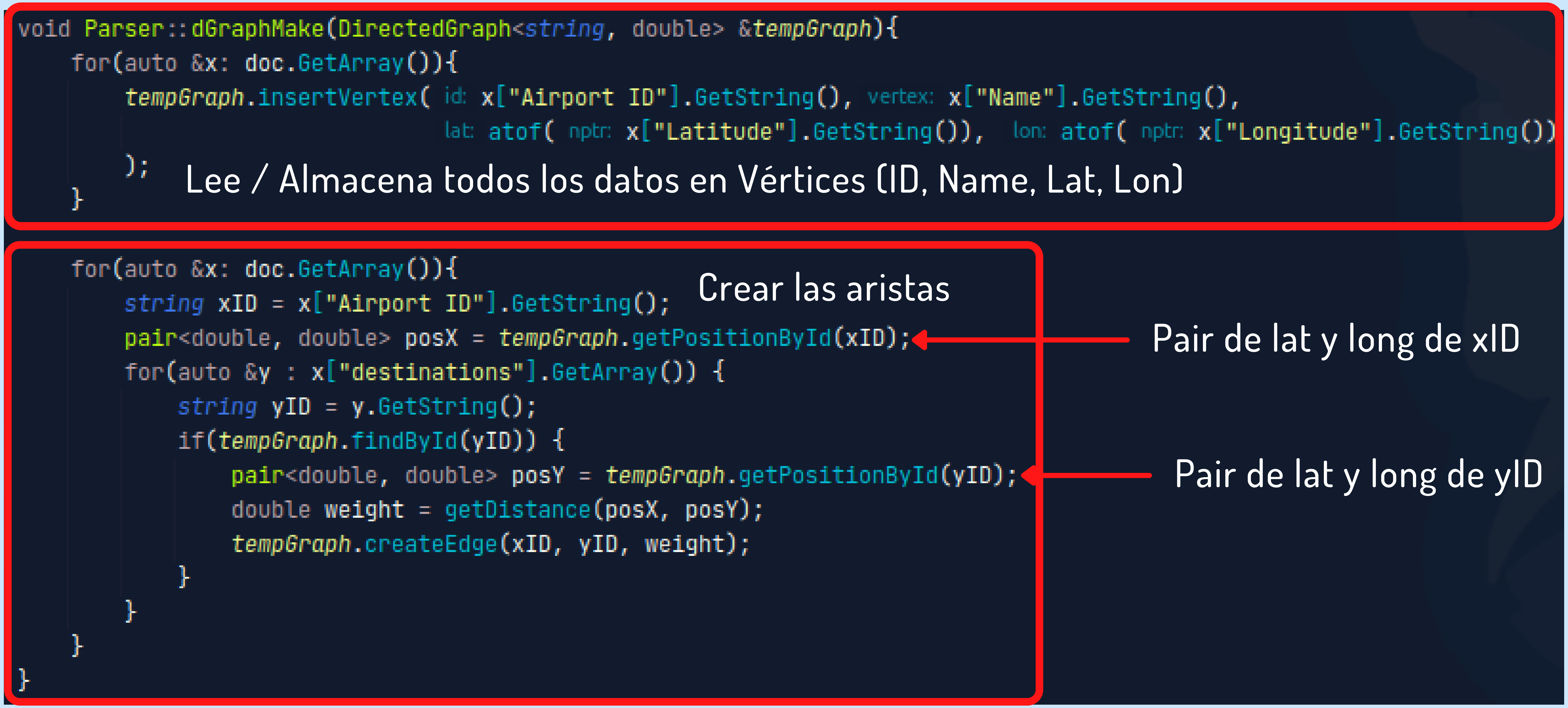


```
proyecto-de-curso-diosplan  ~/Desktop/CICLO5
    .github
    .vscode
    cmake-build-debug
    Graph
    Parser
    rapidjson
    Tester
```



```cpp
void Parser::readJSON(string path){
    std::ifstream is (path, std::ifstream::binary);
    if (is) {
        is.seekg(0, is.end);
        int length = is.tellg();
        is.seekg(0, is.beg);

        char *json = new char[length];

        std::cout << "Reading " << length << " chars... ";
        is.read(json, length);

        doc.Parse(json);
    }
    is.close();
}
```

```cpp
void Parser::printJSON(){
    StringBuffer buffer;
    Writer<StringBuffer> writer( &: buffer);
    doc.Accept( &: writer);

    std::cout << buffer.GetString() << std::endl;
}
```



```
void Parser::dGraphMake(DirectedGraph<string, double> &tempGraph){
    for(auto &x: doc.GetArray()){
        tempGraph.insertVertex( id: x["Airport ID"].GetString(), vertex: x["Name"].GetString(),
                                lat: atof( nptr: x["Latitude"].GetString()),  lon: atof( nptr: x["Longitude"].GetString())
        );
    }
```

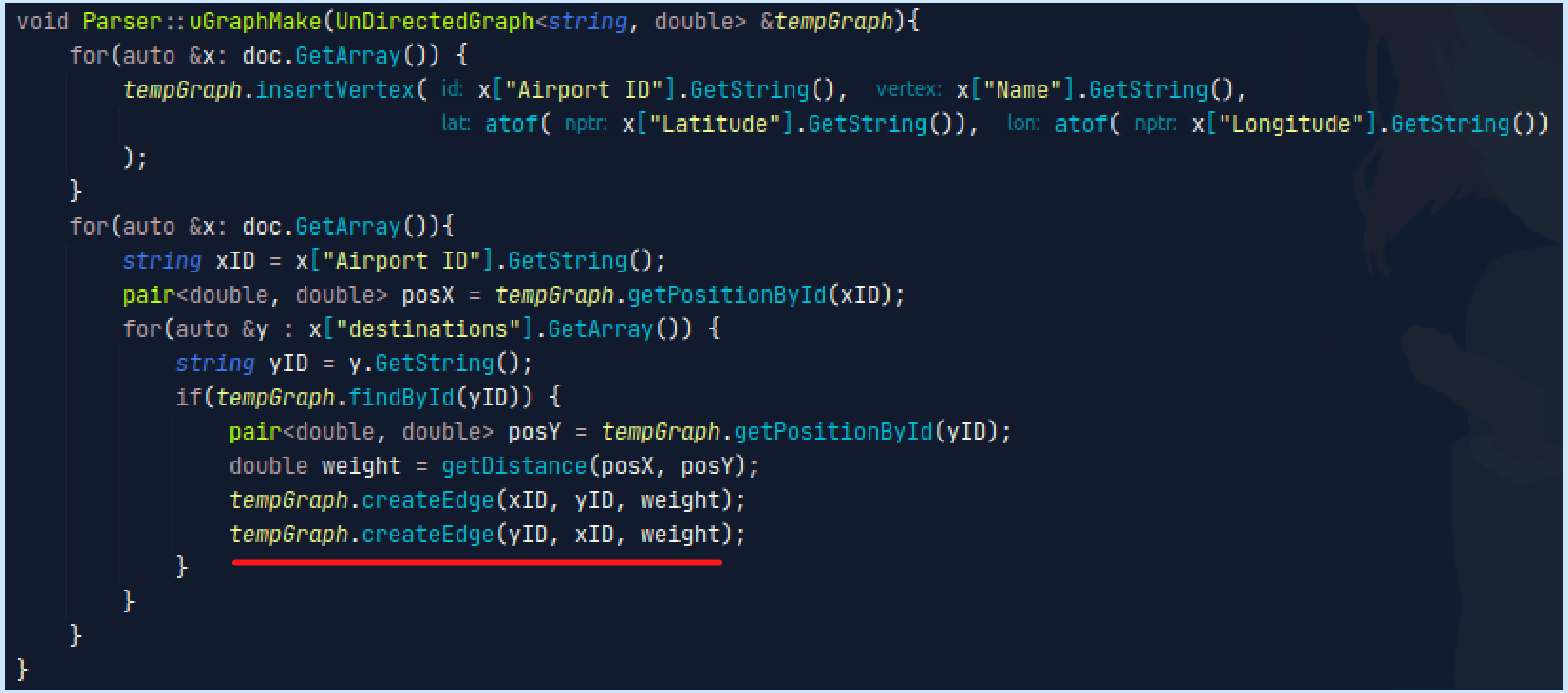
Lee / Almacena todos los datos en Vértices (ID, Name, Lat, Lon)

```
    for(auto &x: doc.GetArray()){
        string xID = x["Airport ID"].GetString();
        pair<double, double> posX = tempGraph.getPositionById(xID);
        for(auto &y : x["destinations"].GetArray()) {
            string yID = y.GetString();
            if(tempGraph.findById(yID)) {
                pair<double, double> posY = tempGraph.getPositionById(yID);
                double weight = getDistance(posX, posY);
                tempGraph.createEdge(xID, yID, weight);
            }
        }
    }
}
```

Crear las aristas

Pair de lat y long de xID

Pair de lat y long de yID

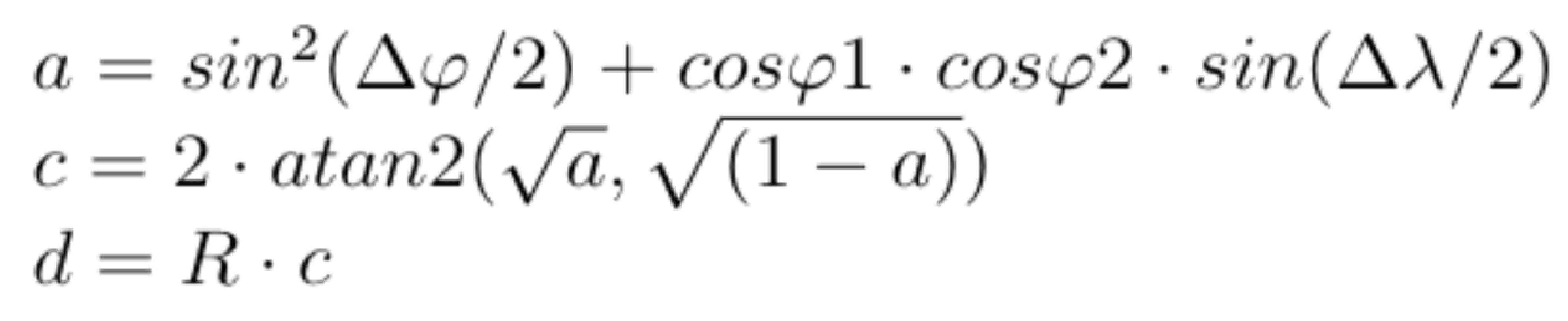


```
void Parser::uGraphMake(UnDirectedGraph<string, double> &tempGraph){
    for(auto &x: doc.GetArray()) {
        tempGraph.insertVertex( id: x["Airport ID"].GetString(),  vertex: x["Name"].GetString(),
                                lat: atof( nptr: x["Latitude"].GetString()),  lon: atof( nptr: x["Longitude"].GetString())
        );
    }
    for(auto &x: doc.GetArray()){
        string xID = x["Airport ID"].GetString();
        pair<double, double> posX = tempGraph.getPositionById(xID);
        for(auto &y : x["destinations"].GetArray()) {
            string yID = y.GetString();
            if(tempGraph.findById(yID)) {
                pair<double, double> posY = tempGraph.getPositionById(yID);
                double weight = getDistance(posX, posY);
                tempGraph.createEdge(xID, yID, weight);
                tempGraph.createEdge(yID, xID, weight);
            }
        }
    }
}
```

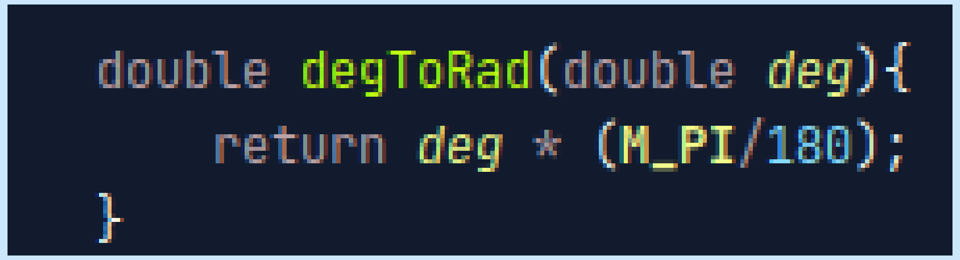
Mismo procedimiento que con el Directed (dGraphMake)

HEURÍSTICA: Haversine Formula

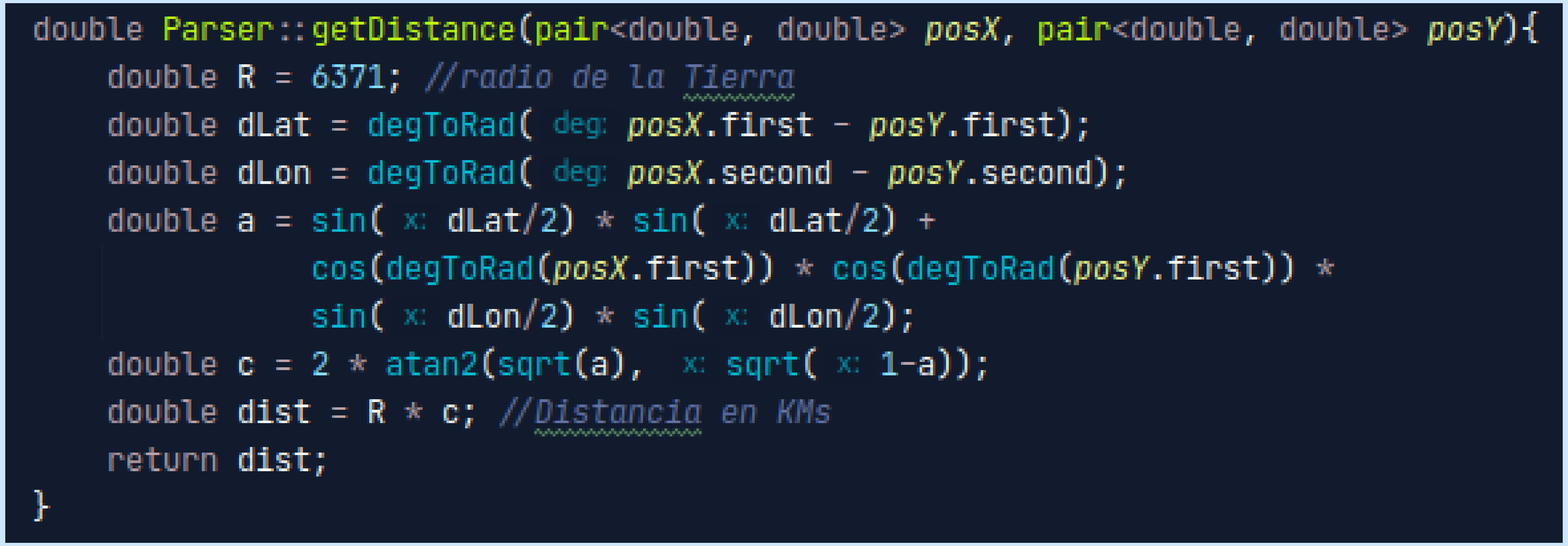
Fórmula:



$$a = sin^2(\Delta\varphi/2) + cos\varphi1 \cdot cos\varphi2 \cdot sin(\Delta\lambda/2)$$
$$c = 2 \cdot atan2(\sqrt{a}, \sqrt{(1-a)})$$
$$d = R \cdot c$$



```
double degToRad(double deg){
    return deg * (M_PI/180);
}
```



```
double Parser::getDistance(pair<double, double> posX, pair<double, double> posY){
    double R = 6371; //radio de la Tierra
    double dLat = degToRad( deg: posX.first - posY.first);
    double dLon = degToRad( deg: posX.second - posY.second);
    double a = sin( x: dLat/2) * sin( x: dLat/2) +
               cos(degToRad(posX.first)) * cos(degToRad(posY.first)) *
               sin( x: dLon/2) * sin( x: dLon/2);
    double c = 2 * atan2(sqrt(a),  x: sqrt( x: 1-a));
    double dist = R * c; //Distancia en KMs
    return dist;
}
```

4.

ALGORITMOS

VAMOS A LA REPO DDD:

5.

ANEXOS

- Haversine Formula:
  https://en.wikipedia.org/wiki/Haversine_formula
  https://www.movable-type.co.uk/scripts/latlong.html

- Repositorio del Proyecto:
  https://github.com/utec-cs-aed-2020-2/graph-project-graph-iteros

GRACIAS :'<