UNIVERSIDAD NEBRIJA

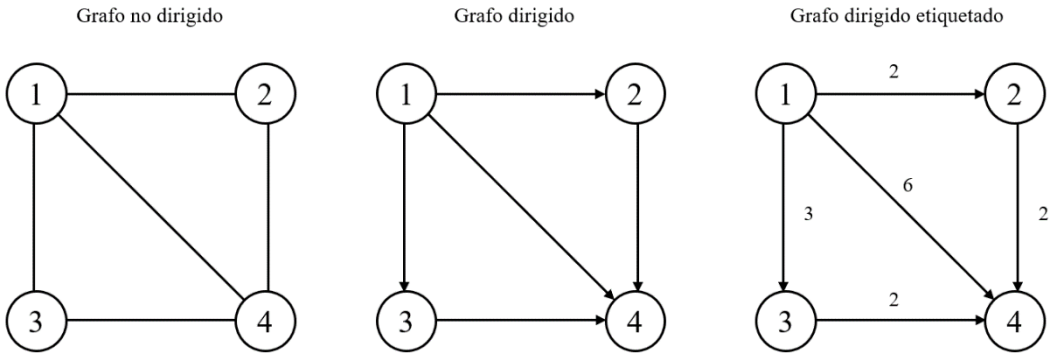
Escuela Politécnica Superior

Estructuras de datos y algoritmos

Práctica 5

Un grafo $G = (V, A)$ se define como un conjunto finito V de vértices y un conjunto A de aristas. Los vértices representan los nodos de un grafo y las aristas indican las uniones entre vértices. Si los vértices o las aristas tienen una etiqueta asociada, el grafo está etiquetado. Una etiqueta puede representar un nombre, un coste, una distancia o cualquier otro dato.

Un grafo no dirigido $G = (V, A)$ consta de un conjunto finito de vértices V y un conjunto de aristas A, donde cada arista es un par no ordenado de vértices. Un grafo dirigido $G = (V, A)$ consta de un conjunto finito de vértices V y un conjunto de aristas A, donde cada arista es un par ordenado de vértices.



Las aristas de un grafo dirigido son pares ordenados. La arista $v \rightarrow w$ se representa (v, w) y es distinta de (w, v).

La interfaz del TAD grafo.

```
inicializa(G, n) Inicializa el grafo G

  Precondición:  Ninguna
  Postcondición: Un grafo con n vértices

inserta(G, v, w, coste) Inserta la arista (v, w) con el coste indicado

  Precondición:  Ninguna
  Postcondición: El grafo incluye la arista (v, w)

imprime(G) Muestra la tabla de aristas del grafo

  Precondición:  Ninguna
  Postcondición: Ninguna
```

Implemente el TAD grafo utilizando el siguiente código.

```
class Grafo {

public:

   Grafo(int v);
   ~Grafo();

   int coste(int v, int w);
   void inserta(int v, int w, int coste);
   int totalVertices();

   static int* dijkstra(Grafo G);
   static Grafo prim(Grafo G);

   std::string imprime(std::string s);
   std::string profundidad(int inicio);

   static std::string imprimeVector(std::string s,
      int* vector, int n);

private:

   static const int INFINITO =
      std::numeric_limits<unsigned short int>::max();

   int vertices;
   int **aristas;

   static void costeMinimoArista(Grafo G, Conjunto U,
      Conjunto W, int &u, int &w);

   int verticeNoVisitado(bool *visitado, int vertices);
   std::string profundidad(int v, bool *&visitado);

};
```

El recorrido en profundidad.

```
int Grafo::verticeNoVisitado(bool *visitado, int vertices) {

   // devuelve el siguiente vértice no visitado o -1 si ya
   // se han visitado todos los vértices del grafo

}

std::string Grafo::profundidad(int v, bool *&visitado) {
   std::stringstream ss;

   visitado[v] = true;

   ss << v + 1 << " ";

   // selecciona la arista i que lleva a un vértice no visitado

   for (int i = 0; i < vertices; i++)
      if (aristas[v][i] != Grafo::INFINITO && !visitado[i])
         ss << profundidad(i, visitado);

   return ss.str();
}

std::string Grafo::profundidad(int inicio) {
   std::stringstream ss;

   bool *visitado = new bool[vertices];

   for (int i = 0; i < vertices; i++)
      visitado[i] = false;

   ss << "\nRecorrido en profundidad \n";

   int v = inicio - 1;

   do {
      ss << "\n  " << profundidad(v, visitado);

      // comprueba si existe un vértice v del grafo que no ha
      // sido visitado

      v = verticeNoVisitado(visitado, vertices);
   } while (v != -1);

   ss << "\n";

   return ss.str();
}
```

El camino más corto: Dijkstra.

```
int* Grafo::dijkstra(Grafo G) {

   // V = {1, 2, 3, … , N}
   // S = {1}
   // D = array de costes mínimos desde el vértice 1
   //
   // W = V – S
   //
   // for vértices 2, 3, 4,.. N del grafo
   //    busca el vértice w con coste mínimo en D
   //    añade w al conjunto S
   //
   //    W = V – S
   //
   //    for vértices u en el conjunto W
   //       calcula D[u] = min(D[u], D[w] + coste arista(w, u))
   //    end for
   // end for
   //
   // devuelve el array D

}
```

El árbol de recubrimiento de coste mínimo: Prim.

```
Grafo Grafo::prim(Grafo G) {

   // V = {1, 2, 3, … , N}
   // U = {1}
   //
   // W = V – U
   //
   // T = Grafo(N)
   //
   // while (U != V)
   //    añade la arista de coste mínimo (u, w) que une un vértice
   //    u elemento de U con un vértice W elemento de W
   //
   //    añade el vértice w al conjunto U
   //    elimina el vértice w del conjunto W
   // end while
   //
   // devuelve el árbol T

}
```

El programa de prueba del TAD grafo.

```
int main() {

   Grafo g1 = Grafo(5);

   g1.inserta(1, 2, 10);
   g1.inserta(1, 4, 30);
   g1.inserta(1, 5, 100);
   g1.inserta(2, 3, 50);
   g1.inserta(3, 5, 10);
   g1.inserta(4, 3, 20);
   g1.inserta(4, 5, 60);

   std::cout << g1.imprime("Grafo");

   // recorrido en profundidad

   Grafo g2 = Grafo(7);

   g2.inserta(1, 2, 10);
   g2.inserta(2, 3, 10);
   g2.inserta(2, 4, 10);
   g2.inserta(3, 1, 10);
   g2.inserta(4, 1, 10);
   g2.inserta(5, 6, 10);
   g2.inserta(5, 7, 10);
   g2.inserta(6, 2, 10);
   g2.inserta(7, 4, 10);
   g2.inserta(7, 6, 10);

   std::cout << g2.imprime("Grafo");

   std::cout << g2.profundidad(1);

   // el camino mas corto: Dijkstra

   Grafo d = Grafo(5);

   d.inserta(1, 2, 10);
   d.inserta(1, 4, 30);
   d.inserta(1, 5, 100);
   d.inserta(2, 3, 50);
   d.inserta(3, 5, 10);
   d.inserta(4, 3, 20);
   d.inserta(4, 5, 60);
```

```
    std::cout << d.imprime("Grafo para Dijkstra");

    int *C = Grafo::dijkstra(d);

    std::cout << Grafo::imprimeVector("Costes Dijkstra",
      C, d.totalVertices());

    // arbol de recubrimiento de coste minimo: Prim

    Grafo p = Grafo(6);

    p.inserta(1, 2, 6);
    p.inserta(2, 1, 6);
    p.inserta(1, 3, 1);
    p.inserta(3, 1, 1);
    p.inserta(1, 4, 5);
    p.inserta(4, 1, 5);
    p.inserta(2, 3, 5);
    p.inserta(3, 2, 5);
    p.inserta(2, 5, 3);
    p.inserta(5, 2, 3);
    p.inserta(3, 4, 5);
    p.inserta(4, 3, 5);
    p.inserta(3, 5, 6);
    p.inserta(5, 3, 6);
    p.inserta(3, 6, 4);
    p.inserta(6, 3, 4);
    p.inserta(4, 6, 2);
    p.inserta(6, 4, 2);
    p.inserta(5, 6, 6);
    p.inserta(6, 5, 6);

    std::cout << p.imprime("Grafo para Prim");

    Grafo r = Grafo::prim(p);

    std::cout << r.imprime("Arbol de recubrimiento mínimo de P");

}
```