ALGORITMO DE DIJKSTRA

GRADO EN INGENIERÍA INFORMÁTICA

CURSO 2020-2021

Daniela Córdova Porta

Facultad de Informática

Universidad Complutense de Madrid

Madrid, 17 de octubre de 2020

Contents

1. Introducción 3
2. Algoritmo De Dijkstra 3
   a. Funciones 3
      i. Representación 3
      ii. Algoritmo 4
   b. Casos 6
      i. Grafo No Dirigido (D1) 6
      ii. Grafo Dirigido (D2) 8
3. Test Unitarios: 9
   a. Existencia de un nodo 9
   b. Existencia de un arco 9
   c. Existencia de que un nodo tiene conexión con otros 10
   d. Prueba diferentes errores al insertar y borrar nodos 10
   e. Reciba argumentos de entrada adecuados 11
   f. Prueba un caso (D1: Grafo no dirigirdo) 12

1. Introducción

Este documento explica cómo fue desarrollado el algoritmo, las funciones que posee y ejemplos a casos que se pueden pasar por entrada al proyecto en c++. También describe cómo y el por qué de la representación del grafo y los test unitarios que se crearon para el proyecto.

2. Algoritmo De Dijkstra

Está desarrollado en lenguaje C++ en 2 ficheros. Uno es “Source.cpp” que posee el main para la ejecución del proyecto y recibe la entrada por parte del usuario y “grafo.h” que tiene la definición de la clase grafo.

a. Funciones

i. Representación

```
struct arco;

	struct nodo
	{
		string v;
		unordered_map<string, arco> mapaArcos;
		nodo(string _v) : v(_v) {}
		nodo() : v("") {}
		bool operator ==(const nodo& b) const // Sobrecarga del operador ==
		{
			return this->v == b.v;
		}
	};

	struct arco {
		int peso;
		nodo *origen;
		nodo *destino;
		arco(nodo* _origen, nodo* _destino, int valor) :peso(valor), origen(_ori-
gen), destino(_destino) {}
	};


	int numNodos;
	bool dirigido;
	unordered_map<string, pair<nodo, bool>> mapaNodos;
```

Para representar el grafo, uso 2 estructuras y un mapa de Nodos. Se decidió de esta manera ya que necesitaba una estructura nodo que serán los vértices, luego una estructura arco para los arcos entre vértices y un mapa de nodos para guardar todos los nodos en la aplicación y poder encontrar el que desea ser el nodo inicial al realizar el algoritmo.

La estructura nodo tiene su identificados y un mapa de arcos que serán todos los arcos que tiene con otro vértice. La estructura arco guarda los nodos origen, destino y peso para poder obtener información durante el algoritmo.

ii. Algoritmo

Parte pública:

```
bool algoritmoD(string ini, string fin) {


		int sumaFinal = -1;
		list<string> listaNodos;
		pair< list<string>, list<list<string>> > r = { {},{} };
		listaNodos.push_back(nodoInicial(ini)->v);
		if (nodoExiste(ini) && nodoExiste(fin)&&nodoTieneConexion(ini) && nodoTiene-
Conexion(fin))
		{

			algoritmoDBase(nodoInicial(ini), 0, 1, listaNodos, r, sumaFinal,
fin);
			resultado.solucion = r.first;
			resultado.listaCaminos = r.second;
			return true;

		}
		else
			return false;
	}
```

Esta parte del algoritmo es la que es llamada en el fichero Source.cpp y verifica si existen los nodos de origen y destino, y si esos mismos nodos tienen arcos con algún nodo. Así se podrá llegar con un camino. Devuelve un booleano dependiendo de si se pudo realizar el algoritmo o no. Guarda la solución en sol resultado:

```
	struct sol {
		list<string> solucion;
		list<list<string>> listaCaminos;

	};

		sol resultado;
```

resultado.solucion será una lista de tipo string para luego enseñar en pantalla el orden de vértices que toma para llegar hasta el final y resultado.listaCaminos tiene una lista de tipo de listas de strings para guardar todas las soluciones que se fueron guardando.

Parte privada:

```
void algoritmoDBase(nodo* actual, int suma, int num, list<string>& listaNodos, pair<
list<string>, list<list<string>> >& r, int& sumaFinal, string fin) {

		mapaNodos[actual->v].second = true;
		for (auto it = actual->mapaArcos.begin(); it != actual->mapaArcos.end();
++it) {
			{
				//nodo* aux = it->destino;
				nodo* aux = it->second.destino;

				if (!mapaNodos.at(aux->v).second)
				{
```

```
                        if (!mapaNodos.at(aux->v).second)
                                listaNodos.push_back(aux->v);

                        mapaNodos[aux->v].second = true;

                        suma = suma + it->second.peso;
                        if (num == this->numNodos || fin == aux->v)
                        {

                                list<string> f;

                                if (fin == aux->v) {
                                        for (auto const& sol : listaNodos)
                                                f.push_back(sol);
                                        r.second.push_back(f);

                                        if (sumaFinal == -1 || sumaFinal > suma)
                                        {
                                                sumaFinal = suma;
                                                r.first.clear();
                                                for (auto const& sol : listaNodos)
                                                        r.first.push_back(sol);

                                        }
                                }

                        }
                        else
                                if ((int)aux->mapaArcos.size() > 0 && (sumaFinal
> suma || sumaFinal == -1))

                                        algoritmoDBase(aux, suma, num + 1, lista-
Nodos, r, sumaFinal, fin);
                        if (mapaNodos.at(aux->v).second)
                        {
                                mapaNodos[aux->v].second = false;
                                listaNodos.pop_back();
                                suma = suma - it->second.peso;
                        }
                }


        }}}
```

Este algoritmo está resuelto basándose en la estrategia Backtracking o Vuelta atrás. Esta es una técnica basada en hacer una búsqueda sistemática a través de todas las configuraciones posibles dentro de un espacio de búsqueda. Cómo nuestro espacio son todos los caminos posibles desde un origen a destino, se decidió resolverlo de esta manera. Por otra parte, al querer obtener todas las soluciones, no se hace poda del árbol de soluciones.

El algoritmo inicialmente recibe el nodo de origen y usando el mapa de Arcos recorre con un for todos los arcos que tiene este nodo. Luego usando el mapa de nodos, revisa si este nodo ya fue recorrido previamente. Si no es así, entra en el if. Aquí suma los pesos de los arcos y analiza si llegó al último nodo que puede recorrer o al nodo deseado. Si es así, guarda la solución; sino, vuelve a llamar la función (recursión) y establece como el nodo que ahora se analizará para ver todos sus arcos es el nodo que no se ha recorrido y era el destino del nodo actual:

```
nodo* aux = it->second.destino
```

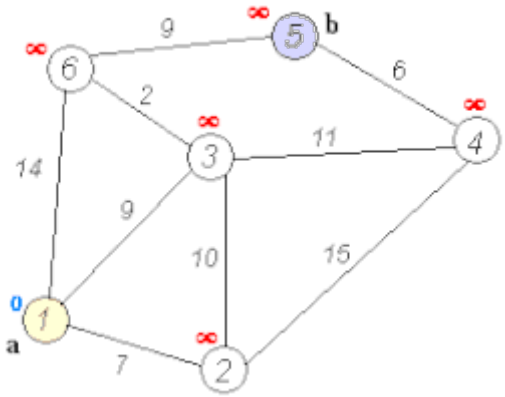
```
algoritmoDBase(aux, suma, num + 1, listaNodos, r, sumaFinal, fin);
```

Esto lo irá haciendo con cada vértice hasta llegar a la solución final. Al terminar la recursión y devolverse; se deben "limpiar" los valores. Esto implica, que la suma de pesos ahora hay que devolverla antes de cuándo se escogió el siguiente vértice, actualizar le mapa de nodos y la lista de nodos que es la que tiene la solución del camino actual:

```
mapaNodos[aux->v].second = false;
listaNodos.pop_back();
suma = suma - it->second.peso;
```

b. Casos

i. Grafo No Dirigido (D1)



Para obtener este grafo, copiar y pegar todo el Fichero: D1.txt en la entrada del proyecto en ejecución.

Como se puede observar:

1. Camino óptimo es:

a d c b f e h

2. Caminos posibles:

Camino 1

a b c d e f g h

Camino 2

a b c d e f h

Camino 3

a b c d e h

Camino 4

a b c e f g h

Camino 5

a b c e f h

Camino 6

a b c e h

Camino 7

a b c f h

Camino 8

a b f g h

Camino 9

a b f e h

Camino 10

a b f h

Camino 11

a b g h

Camino 12

a c b f g h

Camino 13

a c b f e h

Camino 14

a c b f h

Camino 15

a c b g h

Camino 16

a c e f h

Camino 17

a c e h

Camino 18

a c f h

Camino 19

a d c b f g h

Camino 20

a d c b f e h

Camino 21

a d c b f h

Camino 22

a d c b g h

Camino 23

a d c e f h

Camino 24

a d c e h

Camino 25

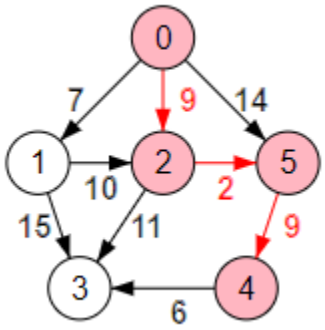
a d c f h

Camino 26

a d e h

Al ser un grafo no dirigido, recorre por los dos sentidos el arco.

ii. Grafo Dirigido (D2)



Para obtener este grafo, copiar y pegar todo el Fichero: D2.txt en la entrada del proyecto en ejecución.

Como se puede observar:

1. Camino óptimo es:

0 2 5 4

2. Caminos posibles:

Camino 1

0 1 2 5 4

Camino 2

0 2 5 4

Camino 3

0 5 4

Al ser un grafo dirigido, sólo recorre los vértices siguiendo las flechas vista en la imagen.

3. Test Unitarios:

Se encuentra en el proyecto AlgortmDjs-Test1 que forma parte de la solución del Project1. En este apartado de explicaran los test unitarios creados en el test.cpp

a. Existencia de un nodo

```
//Determina si encuentra un nodo. Probando para grafo no dirigido
TEST_F(GrafoTest, SiNodoExiste) {

	grafo g;
	string inicio = "a";
	string fin = "b";
	ASSERT_FALSE(g.nodoExiste(inicio));

	g = grafo(inicio, 0);
	ASSERT_TRUE(g.nodoExiste(inicio));

	g.insertarNodo(fin);
	g.insertarArco(inicio, fin, 10);

	ASSERT_TRUE(g.nodoExiste(fin));

	g.borrarNodo(inicio);
	ASSERT_FALSE(g.nodoExiste(inicio));

	g.borrarNodo(fin);
	ASSERT_FALSE(g.nodoExiste(fin));

}
```

b. Existencia de un arco

```
//Determina si encuentra un arco. Probando para no dirigido
TEST_F(GrafoTest, arcos) {

	grafo g;
	string inicio = "a";
	string fin = "b";


	g = grafo(inicio, 0);
	g.insertarNodo(fin);
	g.insertarArco(inicio, fin, 10);
```

```
        ASSERT_TRUE(g.arcoExiste(inicio, fin));
        ASSERT_TRUE(g.arcoExiste(fin, inicio));

        ASSERT_TRUE(g.nodoTieneConexion(inicio));

        g.borrarArco(inicio, fin);
        ASSERT_FALSE(g.arcoExiste(inicio, fin));

        ASSERT_ANY_THROW(g.borrarArco(fin, inicio));  //Debe dar excepción porque ese arco
ya se borró

        ASSERT_FALSE(g.nodoTieneConexion(inicio));

}
```

c. Existencia de que un nodo tiene conexión con otros

```
//Determina si encuentra una conexion para nodos. Probando para no dirigido
TEST_F(GrafoTest, nodoTieneConexion) {

        grafo g;
        string inicio = "a";
        g = grafo(inicio, 0);
        ASSERT_FALSE(g.nodoTieneConexion(inicio));

        g.insertarNodo("b");
        g.insertarArco("a", "b", 16);
        ASSERT_TRUE(g.nodoTieneConexion(inicio));

        g.borrarArco("a", "b");
        ASSERT_FALSE(g.nodoTieneConexion(inicio));
}
```

d. Prueba diferentes errores al insertar y borrar nodos

```
//Errores al insertar y borrar Nodos. Probando para no dirigido
TEST_F(GrafoTest, nodosExepciones) {

        grafo g;

        g = grafo("a", 0);

        g.insertarNodo("b");
        g.insertarNodo("c");
        g.insertarNodo("d");
        g.insertarNodo("e");
        g.insertarNodo("f");
        g.insertarNodo("g");
        g.insertarNodo("h");

        g.insertarArco("a", "b", 16);
        g.insertarArco("b", "c", 2);
```

```
        g.insertarArco("a", "c", 10);
        g.insertarArco("a", "d", 5);


        ASSERT_ANY_THROW(g.insertarNodo("c"));

        g.borrarNodo("c");
        ASSERT_FALSE(g.nodoExiste("c"));

        g.insertarNodo("c");
        ASSERT_FALSE(g.arcoExiste("b", "c"));

}
```

e. Reciba argumentos de entrada adecuados

Este es el apartado recomendado por el profesor:
Solo hay dos vértices y el destino es un tercer vértice inexistente

```
//Probando que el algoritmo acepte como bien los datos de entrada. Probando para no diri-
gido.
TEST_F(GrafoTest, algoritmoDErrores) {

        grafo g;


        //Creo grafo
        g = grafo("a", 0);

        g.insertarNodo("b");
        g.insertarNodo("c");
        g.insertarNodo("d");
        g.insertarNodo("e");
        g.insertarNodo("f");
        g.insertarNodo("g");
        g.insertarNodo("h");

        g.insertarArco("a", "b", 16);
        g.insertarArco("b", "c", 2);
        g.insertarArco("a", "c", 10);
        g.insertarArco("a", "d", 5);
        g.insertarArco("c", "d", 4);
        g.insertarArco("b", "g", 6);
        g.insertarArco("b", "f", 4);
        g.insertarArco("c", "f", 12);
        g.insertarArco("c", "e", 10);
        g.insertarArco("d", "e", 15);
        g.insertarArco("g", "h", 7);
        g.insertarArco("f", "g", 8);
        g.insertarArco("f", "h", 16);
        g.insertarArco("f", "e", 3);
        g.insertarArco("e", "h", 5);

        string nodoI = "a";
        string nodoD = "z"; //nodo que no existe
        string fin;
```

```
	ASSERT_FALSE(g.algoritmoD(nodoI, nodoD)); //devulve falso, no se puede

}
```

f. Prueba un caso (D1: Grafo no dirigirdo)

```
//Probando un caso. Probando para no dirigido (FICHERO DE TESTO D1.TXT)

TEST_F(GrafoTest, algoritmoDjCaso1) {

	grafo g;


	//Creo grafo
	g = grafo("a", 0);

	g.insertarNodo("b");
	g.insertarNodo("c");
	g.insertarNodo("d");
	g.insertarNodo("e");
	g.insertarNodo("f");
	g.insertarNodo("g");
	g.insertarNodo("h");

	g.insertarArco("a", "b", 16);
	g.insertarArco("b", "c", 2);
	g.insertarArco("a", "c", 10);
	g.insertarArco("a", "d", 5);
	g.insertarArco("c", "d", 4);
	g.insertarArco("b", "g", 6);
	g.insertarArco("b", "f", 4);
	g.insertarArco("c", "f", 12);
	g.insertarArco("c", "e", 10);
	g.insertarArco("d", "e", 15);
	g.insertarArco("g", "h", 7);
	g.insertarArco("f", "g", 8);
	g.insertarArco("f", "h", 16);
	g.insertarArco("f", "e", 3);
	g.insertarArco("e", "h", 5);

	string nodoI = "a";
	string nodoD = "h";
	string fin;



	ASSERT_TRUE(g.algoritmoD(nodoI, nodoD));

	list<string> l = g.caminoOptimo();
	for (auto const& elem : l) {
		fin += elem;
	}
```

```
        ASSERT_EQ(fin, "adcbfeh");

}
```