Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Inteligencia Artificial:

Práctica 01

Búsquedas no Informadas.

Cheuk Kelly Ng Pante
(alu0101364544@ull.edu.es)

Índice:

1. Introducción. 2

2. Desarrollo. 2

3. Tablas o gráficas de resultados. 4

4. Conclusión. 5

5. Referencias. 5

1. Introducción.

En esta práctica vamos a desarrollar un programa para que realice la búsqueda no informada para encontrar un camino entre dos nodos de un grafo. Se usará el algoritmo *Búsqueda en Anchura* o también llamado Breadth First Search.

La búsqueda en anchura es un algoritmo de búsqueda no informada utilizado para recorrer o buscar elementos en un grafo. Comienza en la raíz y se visitan todos los vecinos de este nodo. A continuación, para cada uno de los vecinos se exploran sus respectivos vecinos adyacentes, y así hasta que se recorra todo el grafo.

La búsqueda por anchura se usa para aquellos algoritmos en donde resulta crítico elegir el mejor camino posible en cada momento del recorrido.

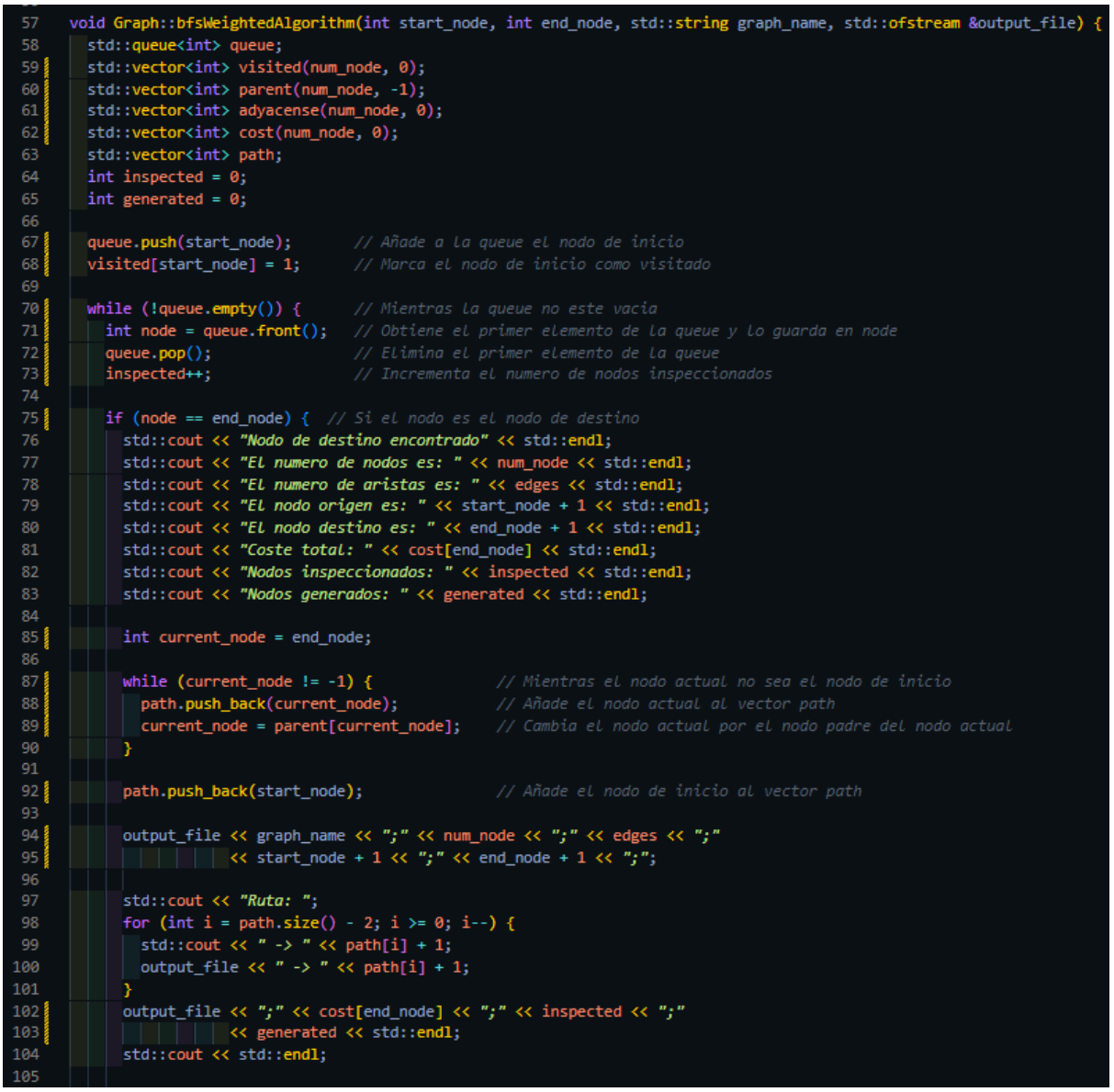
2. Desarrollo.

El desarrollo de está práctica ha sido realizado en el lenguaje de programación C++. Este programa está compuesto por un fichero main que lo llamamos "`BusquedaNoInformada.cc`" y luego dos ficheros que son la clase "`Graph`", `graph.cc` y `graph.h`.

Como observamos en la figura 1.1 y la figura 1.2 tenemos el desarrollo del algoritmo de búsqueda por anchura. Lo primero que hace es inicializar los componentes del algoritmo. Inicializa una "queue" para ir metiendo en la cola los nodos. Luego, lo demás son vectores, uno para los nodos visitados, otro para los nodos padres, las adyacencias, los costes y por último un vector para almacenar el camino del nodo inicial al final.

A continuación, se añade en la "queue" el nodo origen y marcamos este como visitado. Luego, comprueba que la "queue" no esté vacía, si está vacía termina el algoritmo. En el caso de que no esté vacío lo que hace es obtener el primer elemento o nodo que hay en la "queue" y una vez obtenido lo elimina de la cola.

Ahora, si el nodo obtenido en la cola es el nodo destino va a generar el "path" o el camino que hay desde el nodo origen hasta el nodo destino. Y, si no es el nodo destino lo que hace es recorrer todos los nodos y si el nodo es adyacente al nodo actual y no ha sido visitado este se añade a la "queue", lo marca como visitado y como nodo padre.



```cpp
void Graph::bfsWeightedAlgorithm(int start_node, int end_node, std::string graph_name, std::ofstream &output_file) {
  std::queue<int> queue;
  std::vector<int> visited(num_node, 0);
  std::vector<int> parent(num_node, -1);
  std::vector<int> adyacense(num_node, 0);
  std::vector<int> cost(num_node, 0);
  std::vector<int> path;
  int inspected = 0;
  int generated = 0;

  queue.push(start_node);       // Añade a la queue el nodo de inicio
  visited[start_node] = 1;      // Marca el nodo de inicio como visitado

  while (!queue.empty()) {      // Mientras la queue no este vacia
    int node = queue.front();   // Obtiene el primer elemento de la queue y lo guarda en node
    queue.pop();                // Elimina el primer elemento de la queue
    inspected++;                // Incrementa el numero de nodos inspeccionados

    if (node == end_node) {  // Si el nodo es el nodo de destino
      std::cout << "Nodo de destino encontrado" << std::endl;
      std::cout << "El numero de nodos es: " << num_node << std::endl;
      std::cout << "El numero de aristas es: " << edges << std::endl;
      std::cout << "El nodo origen es: " << start_node + 1 << std::endl;
      std::cout << "El nodo destino es: " << end_node + 1 << std::endl;
      std::cout << "Coste total: " << cost[end_node] << std::endl;
      std::cout << "Nodos inspeccionados: " << inspected << std::endl;
      std::cout << "Nodos generados: " << generated << std::endl;

      int current_node = end_node;

      while (current_node != -1) {             // Mientras el nodo actual no sea el nodo de inicio
        path.push_back(current_node);          // Añade el nodo actual al vector path
        current_node = parent[current_node];   // Cambia el nodo actual por el nodo padre del nodo actual
      }

      path.push_back(start_node);              // Añade el nodo de inicio al vector path

      output_file << graph_name << ";" << num_node << ";" << edges << ";"
                  << start_node + 1 << ";" << end_node + 1 << ";";

      std::cout << "Ruta: ";
      for (int i = path.size() - 2; i >= 0; i--) {
        std::cout << " -> " << path[i] + 1;
        output_file << " -> " << path[i] + 1;
      }
      output_file << ";" << cost[end_node] << ";" << inspected << ";"
                  << generated << std::endl;
      std::cout << std::endl;

```

Figura 1.1: Desarrollo del algoritmo BFS

```
106         exit(0);
107       } else {                                  // Si el nodo no es el nodo de destino
108         for (int i = 0; i < num_node; i++) {    // Recorre todos los nodos
109           if (graph[node][i] != NO_EDGE && !visited[i]) {   // Si el nodo es adyacente al nodo actual y no ha sido visitado
110             queue.push(i);                      // Añade el nodo a la queue
111             visited[i] = 1;                     // Marca el nodo como visitado
112             parent[i] = node;                   // Marca el nodo actual como padre del nodo
113             adyacense[i] = graph[node][i];      // Marca el nodo actual como adyacente del nodo
114             cost[i] = cost[node] + adyacense[i];// Guarda el coste total del nodo actual
115             generated++;                        // Incrementa el numero de nodos generados
116           }
117         }
118       }
119     }
120
121     std::cout << "Nodo de destino no encontrado" << std::endl;
122     std::cout << "Nodos inspected: " << inspected << std::endl;
123     std::cout << "Nodos generated: " << generated << std::endl;
124     exit(0);
125   }
```

Figura 1.2: Desarrollo del algoritmo BFS

3. Tablas y grafo generado.

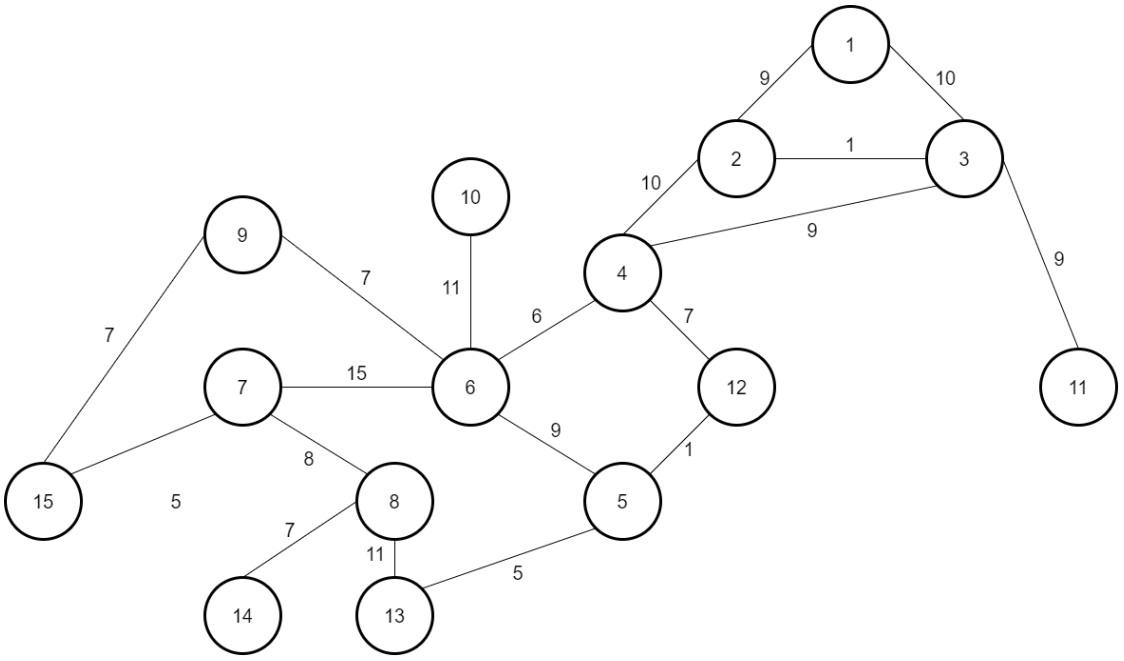


Figura 1.3: Grafo 1

Instancia	n	m	Vo	Vd	Camino	Distancia	Nodos generados	Nodos inspeccionados
Grafo1.txt	15	19	1	2	->1->2	9	2	2
Grafo1.txt	15	19	1	3	->1->3	10	3	3
Grafo1.txt	15	19	1	4	->1->2->4	19	4	4
Grafo1.txt	15	19	1	5	->1->2->4->6->5	31	8	10
Grafo1.txt	15	19	1	6	->1->2->4->6	22	6	6
Grafo1.txt	15	19	1	7	->1->2->4->6->7	37	9	11
Grafo1.txt	15	19	1	8	->1->2->4->6->7->8	45	13	13
Grafo1.txt	15	19	1	9	->1->2->4->6->9	29	10	13
Grafo1.txt	15	19	1	10	->1->2->4->6->10	33	11	13
Grafo1.txt	15	19	1	11	->1->3->11	19	5	6
Grafo1.txt	15	19	1	12	->1->2->4->12	26	7	10
Grafo1.txt	15	19	1	13	->1->2->4->6->5->13	36	12	13
Grafo1.txt	15	19	1	14	->1->2->4->6->7->8->14	52	15	14
Grafo1.txt	15	19	1	15	->1->2->4->6->7->15	42	14	14

Figura 1.4: Algunos resultados del Grafo 1

Instancia	n	m	Vo	Vd	Camino	Distancia	Nodos generados	Nodos inspeccionados
Grafo3.txt	20	25	1	1	->1	0	1	0
Grafo3.txt	20	25	1	2	->1->2	9	2	2
Grafo3.txt	20	25	1	3	->1->2->3	10	4	8
Grafo3.txt	20	25	1	4	->1->2->4	19	5	8
Grafo3.txt	20	25	1	5	->1->5	15	3	5
Grafo3.txt	20	25	1	6	->1->5->6	24	7	12
Grafo3.txt	20	25	1	7	->1->2->4->8->7	35	16	18
Grafo3.txt	20	25	1	8	->1->2->4->8	27	10	14
Grafo3.txt	20	25	1	9	->1->5->6->9	31	14	16
Grafo3.txt	20	25	1	10	->1->5->10	21	8	13

Figura 1.5: Algunos resultados del Grafo 3

4. Conclusión.

En conclusión, el desarrollo de está práctica ha tenido una complejidad grande para mí ya que el algoritmo BFS me costó implementarlo con la forma de introducir el grafo a través de un fichero con el formato que se pedía.

5. Referencias.

1. https://www.encora.com/es/blog/dfs-vs-bfs
2. https://www.programiz.com/dsa/graph-bfs#cpp-code
3. https://www.geeksforgeeks.org/shortest-path-weighted-graph-weight-edge-1-2/?ref=gcse