Grafos

Un grafo es un conjunto de **nodos** conectados por **arcos**. Son muy similares a los arboles pero pueden tener ciclos. Se definen a partir de una lista de nodos V y una lista de arcos E. El numero de arcos en un grafo es $|E|$ y es un valor entre 0 y $|V|^2$.

> Numero de arcos $|E|$ en un grafo
>
> $$0 \leq |E| \leq |V|^2$$

Cuando en un grafo $|E|$ es del orden de $|V|$, se dice que el mismo es **disperso**. Cuando es del orden de $|V|^2$, se dice que es **denso**. Basicamente esto quiere decir que si la cantidad de arcos que tiene el grafo es comparable con la cantidad de nodos, es denso. De lo contrario, seria disperso.

Un grafo es **dirigido** si hay arcos que solamente pueden ser recorridos en un sentido. De lo contrario, es un grafo **no-dirigido**.

Un grafo es **ciclico** si contiene ciclos.

un **DAG** es un Directed Acyclyc Graph, es decir, un grafo dirigido que no presenta ciclos. Son muy utiles pare representar dependencias de tareas y de software. Un ejemplo de DAG es el plan de carrera.

Un grafo es **conexo** cuando existe un camino entre dos nodos cuales quiera. Un grafo puede no ser conexo pero tener componentes conexas, que son subgrafos conexos.

Se dice que un grafo es **pesado** cuando sus arcos tienen pesos o costos asociados.

Implementacion en C++

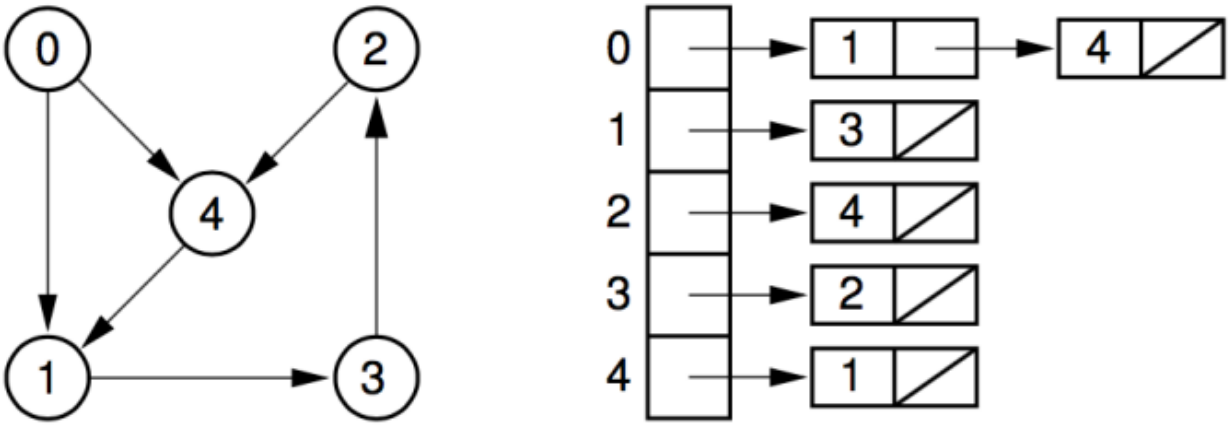
Se puede representar un grafo como una lista de nodos en la que cada nodo enumera sus vecinos. En un grafo con lista de adyacencia, la lista de vecinos enumera los vecinos directos.

```
template <class T>
class GraphNode
{
public:
        T value;
```

```
        bool mark;
        list<int> neighbours;  // Node indices
};

template <class T>
class Graph
{
public:
        vector<GraphNode<T>> nodes;
};
```

La lista de nodos en C++ suele ser un `vector`. Es lo que resulta mas eficiente para esta aplicacion.



Asi se ve la lista de adyacencia para un grafo dirigido.

La lista de adyacencia, recordando mate discreta, se puede representar como una matriz de adyacencia. Por lo general conviene la lista, pero hay casos en los que la matriz es mas eficiente. Para grafos no dirigidos se puede trabajar con solamente media matriz (un triangulo) porque la misma va a ser simetrica.

La lista de adyacencia y los conjuntos de arcos ocupan en memoria $O(|V| + |E|)$, la matriz de adyacencia es $O(|V|^2)$.

Otra manera de ver los grafos es como un conjunto de arcos. Esta representacion puede parecer mas facil de entender pero no suele ser eficiente en calculo.

```
template <class T>
class GraphNode3
{ `
public:
```

```
        T node;
        bool mark;
};

class GraphArc
{
public:
        int src;
        int dest;
};

template <class T>
class Graph3
{
public:
        vector<GraphNode3<T>> nodes;
        vector<GraphArc> arcs;
};
```

Si necesitamos recorrer un grafo disperso, se prefiere usar listas de adyacencia (tener la lista es como tener el camino pre-armado porque la lista me enumera los vecinos). La matriz de adyacencia (sobre todo en casos de grafos muy grandes) esta llena de elementos que son false y aportan muy poca informacion para la memoria que ocupan.

En un grafo muy denso y grande, la matriz puede resultar mas eficiente en memoria porque va a tener muchos true y menos punteros a elementos de una lista.

El caso mas raro es el del conjunto de arcos. Este casi nunca resulta ser el mas eficiente.

Recorrido de Grafos

El primer paso para recorrer grafos es tomar la lista de nodos y poner sus marcas en false. Luego se busca iterativamente un nodo que no se haya visitado y desde ahi recorrer cada componente conexa con DFS o BFS.

```
void traverseGraph(Graph &graph)
{
        // Clear marks
        for (auto &node : graph.nodes)
                node.mark = false;

        // Start traversal
        for (auto &node : graph.nodes)
        {
                if (!node.mark)
```

```
                {
                        traverseDFS(graph, &node);
                        // Or:
                        // traverseBFS(graph, &node);
                }
        }
}
```

Recorrido DFS

```
void traverseDFS(Graph &graph, GraphNode, *node)
{
        node->mark = true;
        visitPreorder(node);

        for (int i : node->neighbors)
        {
                if (!node.mark)
                        traverseDFS(graph, graph.nodes+i);
        }
        visitPostorder(node);
}
```

La novedad de esto con respecot al DFS que cubrimos para arboles no es mas que el manejo de las marcas. El resultado de un recorrido DFS es un arbol (se llaman arboles DFS) que no presentara ciclos.

Recorrido BFS

Este es casi identico al BFS de arboles con algunas salvedades imlpementadas para que funcione con grafos.

```
void traverseBFS(Graph &graph, GraphNode *root)
{
        queue<GraphNode *> q;
        q.push(root);

        while (!q.empty())
        {
                TreeNode *node = q.pop();
                node->mark = true;
                visit(node);

                for (int i : node->neighbors)
```

```
                {
                        if (!node.mark)
                                q.push(graph.nodes + i);
                }
        }
}
```

Algoritmos Interesantes

Floodfill

Este es un algoritmo que usa BFS o DFS para rellenar bitmaps. En la pagina de Wikipedia sobre Floodfill hay buenas animaciones que muestran como funciona. El grafo aca esta definido a partir de una matriz. Cada pixel es un nodo y los nodos vecinos son los pixels adyacentes. Se puede usar el valor de un pixel a modo de marca.

Shortest Path

Cuando un grafo no es pesado, se puede usar un BFS para determinar el camino mas corto a otro nodo. La distancia acumulativa recorrida desde el nodo de origen se puede sacar del `while` de BFS. Este algoritmo permite hallar el camino mas corto entre dos p untos evitando obstaculos.

Dijkstra

Este algoritmo permite determinar el camino mas corto entre dos nodos para grados pesados. Es el mismo que vimos en mate discreta.

A*

Es una version de Disjktra que usa una heuristica. Funciona mucho mejor que Dijkstra.

Otros algoritmos para hallar el camino mas corto son

- Bellman-Ford (para pesos negativos)
- Floyd-Warshall (calcula todos los caminos)
- Johnson's (es una mejora a F-W)
- Viterbi (para pesos probabilisticos)

Ordenamiento Topologico

El ordenamiento topologico permite ver de manera clara las dependencias entre una secuencia de tareas. Se puede resolver con un recorrido DFS pos-orden. Por ejemplo, si

tengo que instalar paquetes, me permite saber en que orden los tengo que instalar para que no se rompa nada.

PageRank

Esto es lo que usa Google. Asigna un puntaje o peso a cada pagina web modelando el grafo de links o enlaces entre sitios. Si, por ejemplo, muchas paginas tienen links entrnantes a una pagina, aumentara su peso ya que es un resultado mas deseado.

Navmesh

Es una extension de grafos. Es una estructura de datos basada en grafos que permite la navegacion de robots autonomos.