Grafos

Wilmer Emiro Castrillón Calderón

16 de septiembre de 2022

1. Teoría de grafos

Los grafos son una estructura de datos donde se pueden relacionar distintos objetos entre si, los grafos se pueden definir como un conjunto de nodos(objetos) unidos por aristas(relaciones), estos son estudiados por la matemática y las ciencias de la computación, esta rama se conoce como teoría de grafos, ademas tienen muchas aplicaciones, por ejemplo permiten modelar redes informáticas, sistemas de carreteras, redes sociales, etc.

Terminología general

Los grafos se pueden clasificar de distintas formas, las mas utilizadas son:

1. **Grafos ponderados y no ponderados** Si las aristas de un grafo tienen un peso, es decir si para atravesar una arista esta tiene un costo asociado, entonces el grafo se clasifica como ponderado, pero si ninguna arista tiene algún costo entonces el grafo se clasifica como no ponderado.

2. **Grafos dirigidos y no dirigidos:** Si un grafo posee por lo menos una arista dirigida entonces se clasifica como un grafo dirigido, una arista (A, B) es dirigida si esta permite el paso de A hacia B, pero no permite ir de B hacia A. Si todas las aristas son bidireccionales(no dirigidas) entonces el grafo se clasifica como no dirigido.

3. **Grafos cíclicos y acíclicos** Se considera un grafo como acíclico cuando este no tiene ciclos, es decir para cada pareja de nodos (A, B) si existe un camino para ir de A hacia B entonces no existe otro camino para ir de B hacia A. Si un grafo no es acíclico entonces este es cíclico.

4. **Grafos conexos y no conexos** Si para cada pareja de nodos (A, B) existe un camino para ir de A hacia B y también existe camino para ir de B hacia A entonces el grafo es conexo, en caso contrario es un grafo no conexo.

Representación.

Los grafos se pueden representar de múltiples maneras cada una permite realizar distintos algoritmos, cada forma de representar los grafos se puede ajustar según su tipo, un aspecto importante a considerar es que cada una arista bidireccional (A, B) se puede interpretar como dos aristas dirigidas (A, B) y (B, A). Existen principalmente de tres formas distintas de representarlos:

1. **Matriz de adyacencia:** Es una matriz M en la cual cada fila representa un nodo de inicio y cada columna un nodo destino(a cada nodo se le asignara un numero representando su posición en fila y columna), si existe una arista (A, B) entonces se marca la casilla $M_{A,B}$, en el caso de grafos ponderados se debe llenar $M_{A,B}$ con el costo de la arista (A, B), para los todos los pares de nodos que no tengan aristas entre ellos el costo es infinito. Si el grafo es no ponderado entonces en la matriz simplemente se marca si existe o no la arista (A, B). Por definición cada nodo tiene conexión con sí mismo con costo cero.

2. **Lista de adyacencia:** Consiste en guardar para cada nodo una lista con las conexiones que posee, es decir, para cada arista (A, B) se pondrá en la lista de conexiones de A el nodo B. Si el grafo es ponderado entonces se deberá guardar el nodo destino junto con su costo. Esta es la manera mas utilizada para representar grafos, pues el espacio de memoria que ocupa es menor que el utilizado por una matriz de adyacencia y ademas facilita recorrer el grafo de manera sencilla.

3. **Lista de aristas:** Consiste en una lista en la cual se guardara todas las aristas de un grafo, para cada una se guarda el nodo de inicio, nodo de destino y el costo en caso de tener. Esta es la menos utilizada de las tres, aunque facilita ordenar las aristas según su costo.

A continuación se muestra un pequeño ejemplo para cada una de las tres formas:

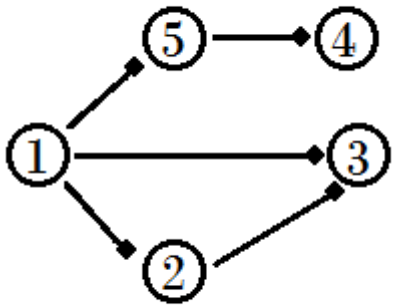

Figura 1

```cpp
bool grafo[10][10];

void construir_grafo(){
    memset(grafo, false, sizeof(grafo));
    grafo[1][5] = true;
    grafo[1][3] = true;
    grafo[1][2] = true;
    grafo[5][4] = true;
    grafo[2][3] = true;
}
```

Matriz de adyacencia

```cpp
vector<vector<int>> grafo(10);

void construir_grafo(){
    grafo[1].push_back(5);
    grafo[1].push_back(3);
    grafo[1].push_back(2);
    grafo[5].push_back(4);
    grafo[2].push_back(3);
}
```

Lista de adyacencia

```cpp
typedef pair<int, int> ii;
vector<ii> grafo;

void construir_grafo(){
    grafo.push_back(ii(1, 5));
    grafo.push_back(ii(1, 3));
    grafo.push_back(ii(1, 2));
    grafo.push_back(ii(5, 4));
    grafo.push_back(ii(2, 3));
}
```

Lista de aristas

2. DFS y BFS

Los grafos son estructuras no lineales, estos no tienen un nodo inicio o un orden especifico para recorrerlos, existen principalmente dos algoritmos que permiten recorrer un grafo DFS y BFS, estos no son algoritmos muy estrictos, es decir, se pueden modificar de múltiples maneras para realizar diferentes tareas, sin embargo la idea básica sigue siendo la misma, para la implementación de ambos se utiliza una lista de adyacencia.

DFS.

El DFS (deep first search) o búsqueda en profundidad, es un algoritmo que permite recorrer un grafo de forma recursiva, de manera general consiste en tomar un nodo, marcarlo como visitado y para cada arista hacer un llamado recursivo al nodo destino, solo si ese nodo no esta visitado, y repetir el proceso hasta que no queden mas nodos por visitar. De esta forma se van marcando los nodos mientras se visitan y en caso de llegar a un nodo sin aristas o un nodo cuyos vecinos se encuentran visitados entonces se devuelve al nodo anterior.

En la siguientes figuras se observa el orden en el cual los nodos son visitados empezando por el nodo 1, entonces comienza con el 1, después avanza al 5 y luego al 4, como no hay mas destinos se devuelve al 5, pero su único vecino ya esta visitado, se devuelve al 1, desde ahí pasa al 3, este no tiene vecinos y se devuelve otra vez al 1, y finalmente al 2, como su vecino ya esta visitado regresa al 1, y como este no tiene mas vecinos finaliza el algoritmo.

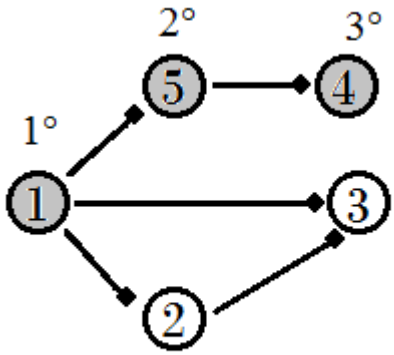

Figura 2

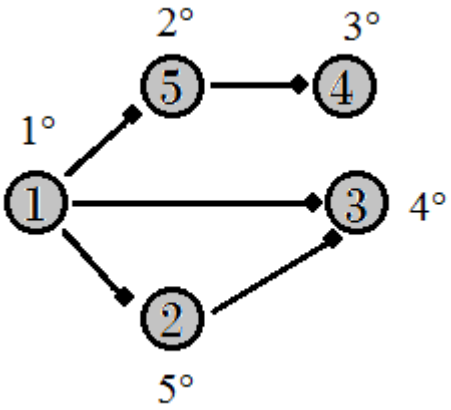

Figura 3

La implementación del DFS es corta y consiste básicamente en un algoritmo de backtraking, tiene una complejidad algorítmica O($V+E$) con V como el numero de nodos y E como el numero de aristas.

```
vector<vector<int>> grafo(10);
bool visitado[10];

void dfs(int nodo){
    visitado[nodo] = true;
    for(int i = 0; i < grafo[nodo].size(); i++){
        if(!visitado[grafo[nodo][i]]){
            dfs(grafo[nodo][i]);
        }
```

```
    }
}
```

BFS.

El BFS(Breadth First Search) o búsqueda en anchura es un algoritmo que permite recorrer un grafo de forma iterativa, la idea consiste en tomar un nodo inicial y recorrer todos los nodos que están a un "salto"de distancia(es decir, sus vecinos), después los nodos que están a dos "saltos"de distancia y así sucesivamente hasta recorrer todos los nodos.

La implementación consiste de utilizar una cola(queue) inicialmente con un nodo, este nodo se debe marcar como visitado, después se empieza a iterar los elementos en la cola, en cada iteración se desencola el siguiente nodo, después se agrega todos sus nodos vecinos no visitados a la cola y se marcan como visitados, el algoritmo termina cuando la cola queda vacía. En la siguiente figura se observa el orden en el cual se visitan los nodos en el grafo de ejemplo anterior.

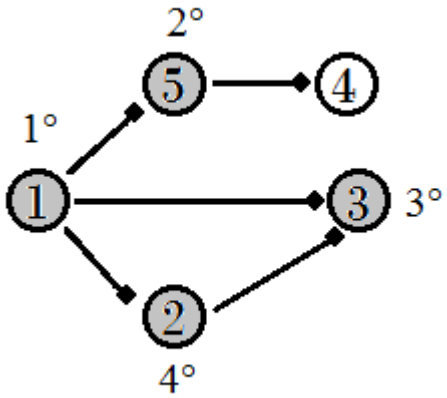

Figura 4

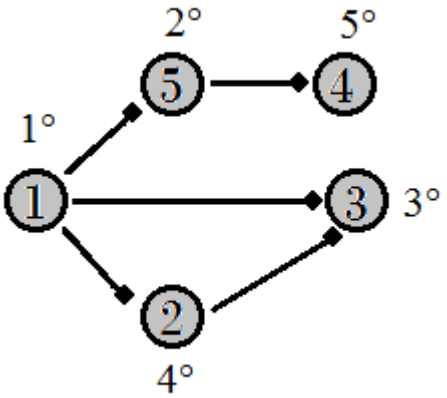

Figura 5

La implementación del BFS es sencilla, tiene una complejidad algorítmica O($V+E$) al igual que el DFS, ademas al ser iterativo evita los problemas de stack overflow que puede presentar el DFS al trabajar con grafos muy extensos.

```
vector<vector<int>> grafo(10);
bool visitado[10];

void bfs(int nodo){
    queue<int> cola;
    cola.push(nodo);  visitado[nodo] = true;

    while(cola.size()){
        nodo = cola.front();  cola.pop();

        for(int i = 0; i < grafo[nodo].size(); i++){
            if(!visitado[grafo[nodo][i]]){
                cola.push(grafo[nodo][i]);
                visitado[grafo[nodo][i]] = true;
            }
```

```
16          }
17      }
18  }
```

Observaciones.

Ambos algoritmos son igual de buenos para recorrer grafos de forma general, dependiendo del problema a resolver uno de los dos algoritmos puede resultar mejor que el otro, esto se debe a las distintas formas de recorrer los grafos, el DFS al ser un algoritmo recursivo permite recorrer los nodos de una forma ordenada, siguiendo un camino en el cual cada nodo es vecino del anterior, en cambio el BFS permite recorrer los grafos por niveles, comenzando con los nodos mas cercanos al nodo inicial, de esta forma se puede encontrar mas fácil distancias entre los distintos nodos. Al utilizar DFS se debe tener cuidado con la pila de memoria(call stack) la cual se puede llenar rápidamente con grafo extensos, produciendo errores de ejecución.

3. Grafos bipartitos

Un grafo bipartito es un grafo el cual se puede dividir sus vértices en dos conjuntos X, Y en los cuales todos los nodos de un mismo conjunto no están conectados entre si, es decir no existe ninguna arista la cual conecte dos nodos de un mismo conjunto, los conjuntos X, Y pueden estar vacíos.

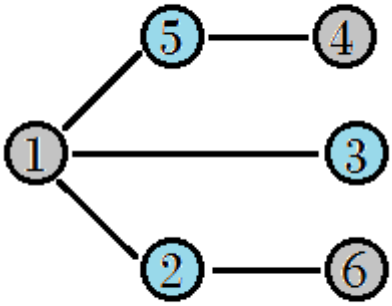

Figura 6: Grafo bipartito

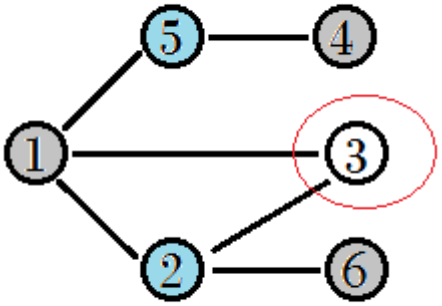

Figura 7: Grafo no bipartito

Para verificar si un grafo es bipartito se puede utilizar un algoritmo sencillo de DFS o BFS, la implementación consiste en tomar un nodo inicial y marcarlo como nodo perteneciente al conjunto X después se visitan y marcan sus vecinos como nodos pertenecientes al conjunto Y, este proceso se repite para cada nodo visitando y marcando sus vecinos como nodos del conjunto opuesto, en caso de encontrar en algún momento a dos nodos vecinos del mismo conjunto entonces significa que el grafo no es bipartito, pero si no se encuentran contradicciones significa que si es bipartito.

```
1  typedef pair<int, int> ii;
2  vector<vector<int>> grafo;
3  bool color[200], used[200];
4
```

```cpp
bool BipartiteCheck(int nodo){
    queue<ii> cola;
    cola.push(ii(nodo, 0));
    memset(used, false, sizeof(used));
    color[nodo] = 0;  used[nodo] = true;
    int auxnodo;

    while(cola.size()){
        ii x = cola.front();  cola.pop();
        nodo = x.first;
        bool newcolor = 1 - x.second;

        for(int i = 0; i < grafo[nodo].size(); ++i){
            auxnodo = grafo[nodo][i];
            if(used[auxnodo] && newcolor != color[auxnodo])
                return false;
            if(used[auxnodo]) continue;

            cola.push(ii(auxnodo, newcolor));
            color[auxnodo] = newcolor;
            used[auxnodo] = true;
        }
    }
    return true;
}
```

4. Topological sort

El ordenamiento topológico o toposort de un grafo G acíclico y dirigido es un ordenamiento lineal de los vértices, de tal forma que para cada arista de u a v se cumple que el nodo u aparece antes que el nodo v, este ordenamiento suele ser utilizado para representar tareas que dependen de otras, es decir si una tarea x depende de otra tarea y, es decir existe una arista $y \rightarrow x$, entonces x no puede iniciar hasta que y este terminada, entonces en el ordenamiento topológico primero debe aparecer y y después x, este ordenamiento puede dar un posible orden para realizar tareas. En la figura 8 se puede observar un sencillo ejemplo.

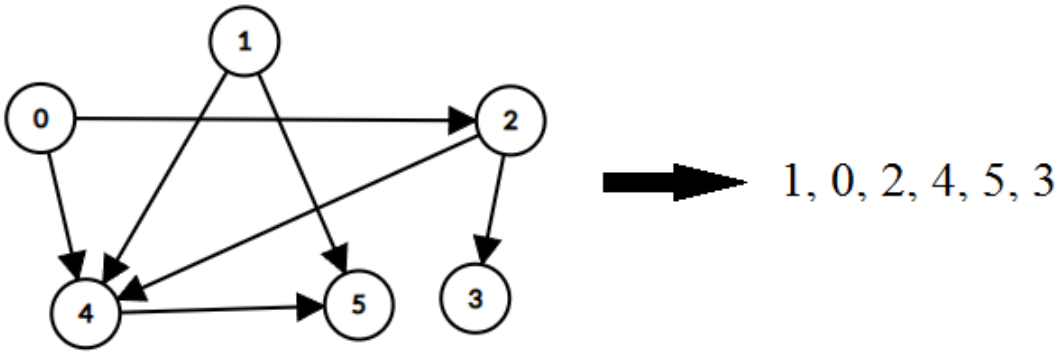

Figura 8

4.1. Topological sort con DFS

Una manera sencilla de encontrar el toposort es utilizando DFS, la solución se construye agregando los nodos en una lista de enteros, sin embargo el objetivo sera agregar primero los nodos destino antes que los nodos de origen, es decir, construir un toposort invertido de tal forma que final solo se debe invertir la lista para obtener el resultado.

Para realizar esto se necesita agregar el nodo v a la lista justo antes de terminar el llamado del DFS, debido a que en ese punto ya se recorrieron y agregaron todos los nodos que van después de v, se recomienda hacer el proceso manualmente para que se pueda entender mejor. La complejidad final seguirá siendo la misma de un DFS normal.

```
vector<vector<int>> grafo(MAX);
bool visitado[MAX];

void dfs(int nodo, vector<int> &toposort){
    visitado[nodo] = true;
    for(int i = 0; i < grafo[nodo].size(); i++){
        if(!visitado[grafo[nodo][i]])
            dfs(grafo[nodo][i], toposort);
    }
    toposort.push_back(nodo);
}

vector<int> toposort_dfs(int n){
    vector<int> toposort;
    memset(visitado, false, sizeof(visitado));

    for(int i = 0; i < n; ++i) {
        if(!visitado[i]) dfs(i, toposort);
    }
    reverse(toposort.begin(), toposort.end());
    return toposort;
}
```

5. Minimum spanning tree

También conocido como árbol recubridor mínimo o abreviado en inglés MST, es un problema clásico en la teoría de grafos, consiste en lo siguiente, Sea $G = (V, E)$ un grafo conexo, ponderado y no dirigido, se denomina árbol recubridor a cualquier subgrafo T tal que T sea un árbol e incluya todos los vértices de G, entonces el árbol recubridor mínimo es un subgrafo $T*$ tal que la suma de los pesos de las aristas de $T*$ es menor o igual al de todos los demás árboles recubridores T, esto significa que pueden haber varios posibles árboles recubridores mínimos. Existen varios algoritmos que pueden resolver este problema, dos de ellos son el algoritmo de Kruskal y el algoritmo de Prim.

5.1. Algoritmo de Kruskal

Es un algoritmo inventado por el matemático Joseph Kruskal, permite hallar el MST utilizando un enfoque greddy y funciona en complejidad $O(E * log(V))$ con E como el número

de aristas y V el número de nodos, consiste en representar el grafo como una lista de aristas, ordenar la lista según el peso de las aristas e ir seleccionando aristas para formar un árbol, entonces para cada arista se verifica que no forme un ciclo con las demás aristas seleccionadas, de esta forma se priorizan las aristas de menor peso y el árbol formado al final es un MST.

la implementación consiste en representar el grafo como una lista de aristas, posteriormente ordenarlas de forma ascendente según el peso de la arista, después se empieza a construir un árbol, se van recorriendo las aristas y para cada una se verifica si al agregarla al árbol se forma un ciclo, de ser así la arista se descarta, en caso contrario la arista se agrega al árbol, para realizar esa verificación se utiliza un disjoint set, el cual resulta mucho mejor computacionalmente que realizar un DFS, cada vez que se agrega una arista al árbol se actualiza el disjoint set uniendo los dos nodos de la arista en un mismo conjunto, entonces si en algún momento los dos nodos de una arista se encuentran en un mismo conjunto se puede concluir que esos dos nodos ya están dentro de una misma componente conexa y por lo tanto, al agregar esa arista se estaría formando un ciclo, de esta forma se valida cada arista rápidamente. El algoritmo termina cuando se han agregado $n-1$ aristas al árbol(la cantidad necesaria para que el árbol conecte todos los N nodos) o cuando se acaben las aristas, lo que significa que el árbol no es conexo.

```
#define mpiii(peso, inicio, destino) iii(peso, ii(inicio, destino))
typedef pair<int, int> ii;
typedef pair<int, ii> iii;

int kruskal(vector<iii>& grafo, int n) {
  sort(grafo.begin(), grafo.end());
  union_find uf(n+1);
  int peso = 0, cont = 0;

  for(int i = 0; i < grafo.size(); i++) {
    ii arista = grafo[i].second;
    if(!uf.mismo_grupo(arista.first, arista.second)) {
      uf.unir(arista.first, arista.second);
      peso += grafo[i].first;
      if(++cont == n - 1)
        return peso;
    }
  }
  return -1;
}
```