Teoría de grafos pt.1

¿Cómo funciona google maps?

¿Qué es un grafo?

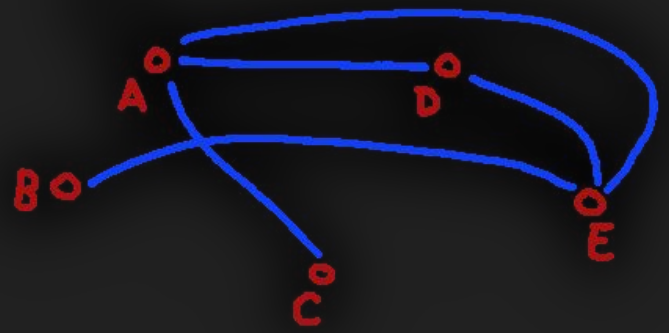
Definición

Un grafo consta de un conjunto de vértices (puntos) y un conjunto de aristas (líneas) que los conectan.

Estos se usan para modelar entidades y relaciones simétricas entre ellas.

Solemos decir que un grafo tiene N vértices y M aristas

Ejemplos: personas y amistades, ciudades y rutas, caminos en laberintos, etc.



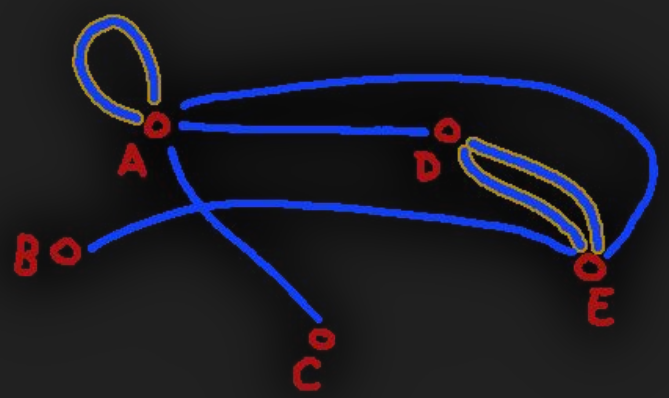
Grafos simples

Dos aristas son paralelas si tocan los mismos vértices.

Un bucle es una arista que va de un vértice a sí mismo.

Decimos que un grafo es simple si no tiene bucles ni aristas paralelas.

Normalmente trabajamos con grafos simples, así que lo típico es aclarar solo si no es simple.

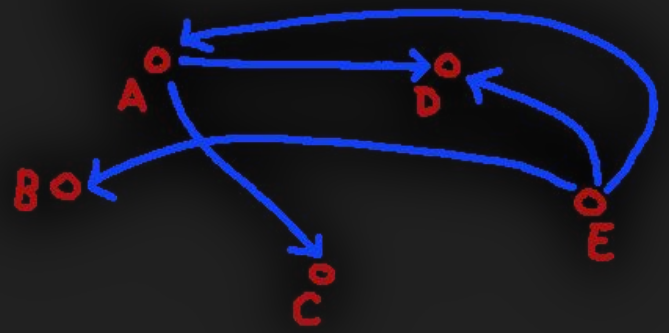


Grafos dirigidos

Si le asignamos una dirección a cada arista, terminamos con un grafo dirigido (o “digrafo”)

Estos sirven para modelar relaciones asimétricas.

Ejemplos: seguidores en redes sociales, derrotas en un torneo, calles de una sola mano, orden de las misiones en un videojuego, etc.

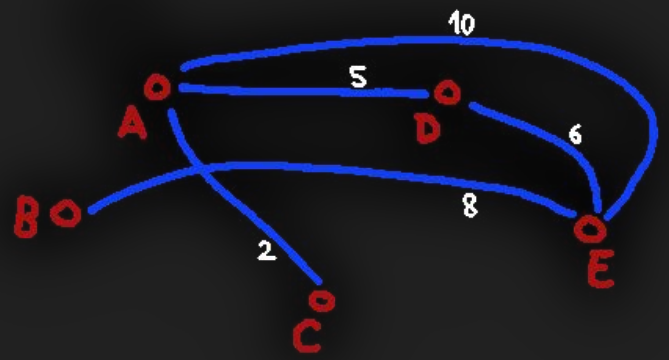


Grafos ponderados

Es común querer asignarle un “costo” (o “peso”) a cada arista. Al hacer esto, terminamos con un grafo ponderado.

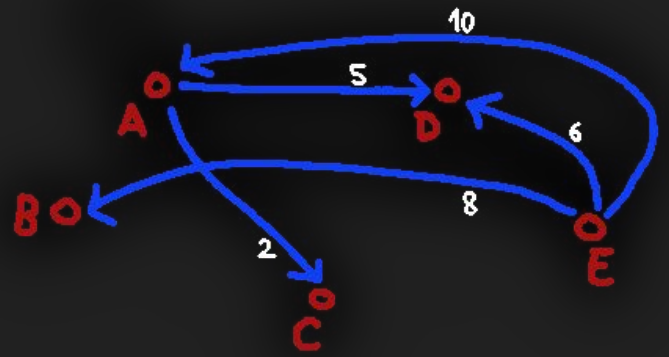
Estos sirven para representar relaciones con un valor numérico.

Por ejemplo: distancias entre lugares, deudas entre personas, costo de una jugada en un juego de mesa, etc.



Aclaración

Un grafo puede ser simultáneamente simple, dirigido, ponderado, etc.



Algunas definiciones más

grado de un vértice: cantidad de aristas que tocan ese vértice. En grafos dirigidos distinguimos entre el grado de entrada y de salida.

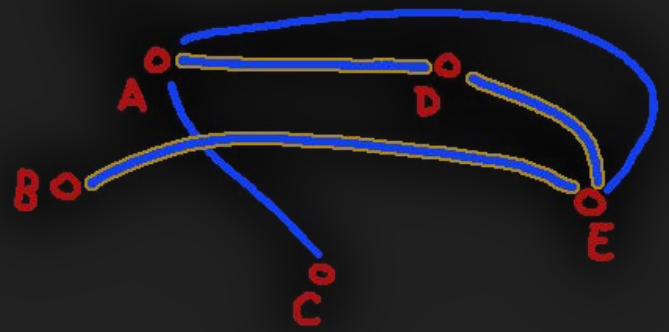
subgrafo: Decimos que “un grafo es subgrafo de otro” cuando el primero se puede obtener borrando algunos vértices y aristas del primero.

subgrafo recubridor: Decimos que “un grafo es subgrafo recubridor de otro” cuando el primero es subgrafo del segundo, y tiene todos los vértices. (en otras palabras, su puede obtener borrando aristas)

Caminos

camino: una secuencia de aristas conectadas que arranca en un vértice y termina en otro. Un camino es simple si no repite vértices.

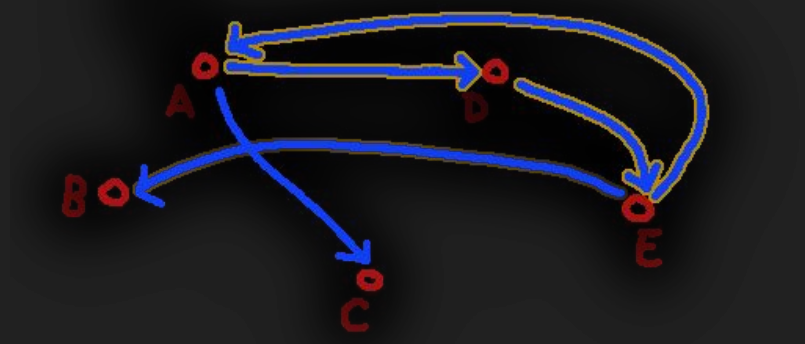
camino dirigido: camino que va “a favor” de las aristas en un grafo dirigido



Ciclos

ciclo: un camino que arranca y termina en el mismo vértice.

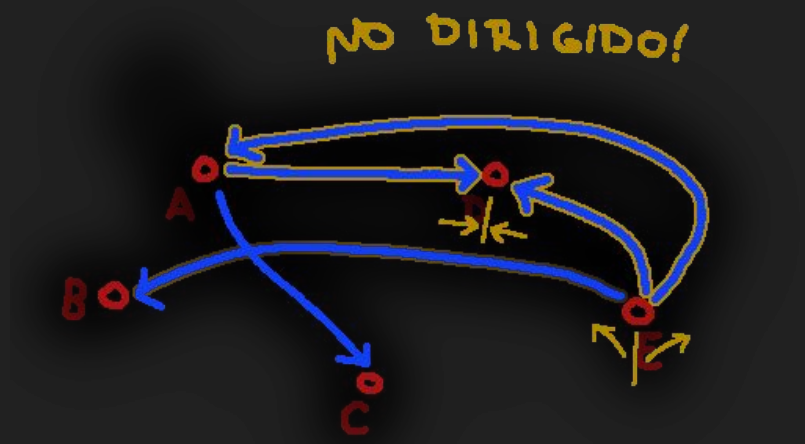
ciclo dirigido: idem pero con camino dirigido



Ciclos

ciclo: un camino que arranca y termina en el mismo vértice.

ciclo dirigido: idem pero con camino dirigido



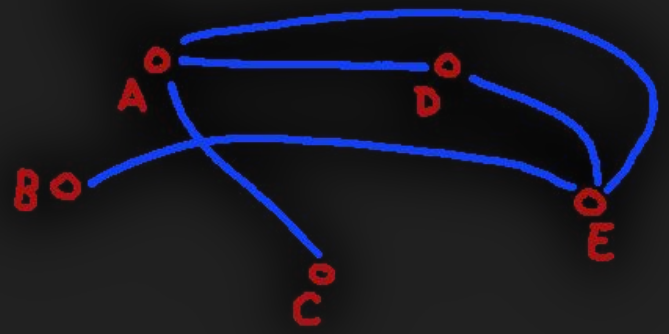
Componentes conexas

conectividad: decimos que u y v (dos vértices) están conectados si hay un camino de u a v

componente conexa: conjunto de vértices donde todos están conectados

alcanzabilidad: decimos que un u es alcanzable desde v si hay un camino dirigido de u a v

componente fuertemente conexa: conjunto de vertices donde todos son alcanzables desde todos



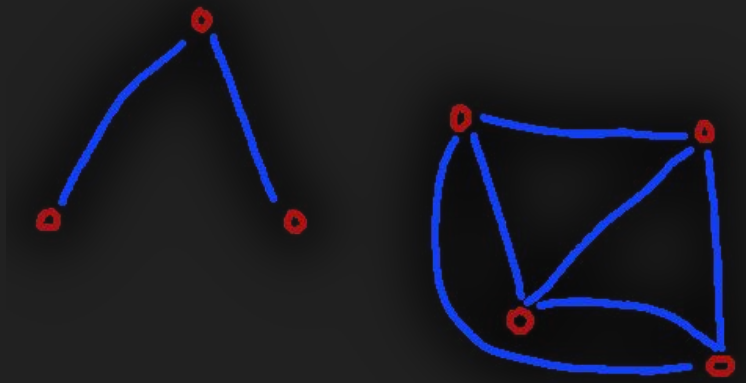
Componentes conexas

conectividad: decimos que u y v (dos vértices) están conectados si hay un camino de u a v

componente conexa: conjunto de vértices donde todos están conectados

alcanzabilidad: decimos que un u es alcanzable desde v si hay un camino dirigido de u a v

componente fuertemente conexa: conjunto de vertices donde todos son alcanzables desde todos



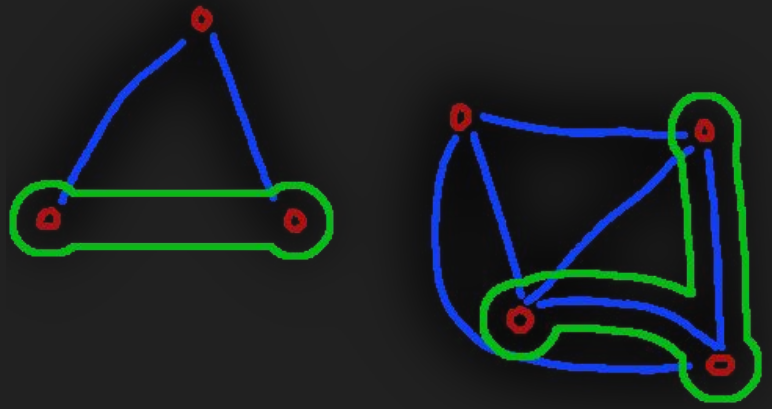
Componentes conexas

conectividad: decimos que dos vértices u y v están conectados si hay un camino de u a v

componente conexa: conjunto de vértices donde todos están conectados

alcanzabilidad: decimos que un u es alcanzable desde v si hay un camino dirigido de u a v

componente fuertemente conexa: conjunto de vértices donde todos son alcanzables desde todos



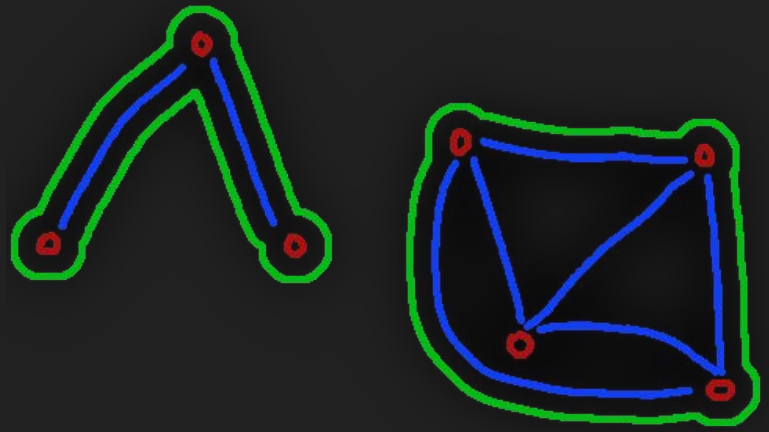
Componentes conexas

conectividad: decimos que dos vértices u y v están conectados si hay un camino de u a v

componente conexa: conjunto de vértices donde todos están conectados

alcanzabilidad: decimos que un u es alcanzable desde v si hay un camino dirigido de u a v

componente fuertemente conexa: conjunto de vértices donde todos son alcanzables desde todos



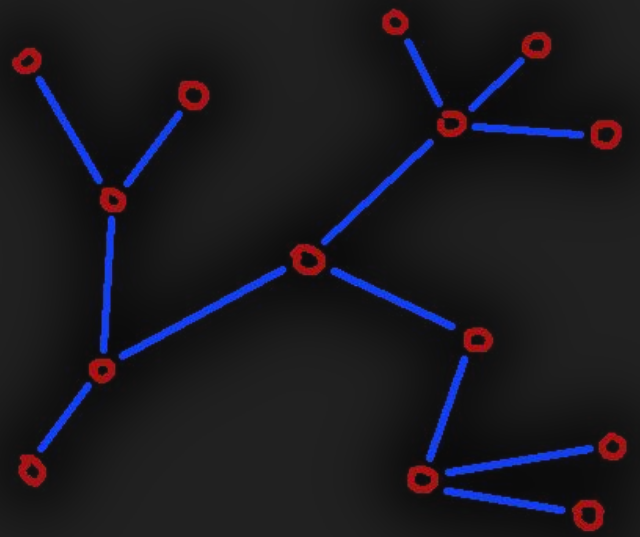
Árboles

Árbol

Un grafo conexo que no tiene ciclos.

Otras definiciones equivalentes:

- grafo conexo con M=N-1
- grafo donde existe exactamente un camino simple entre cada par de vértices
- grafo conexo que deja de serlo si se elimina una arista

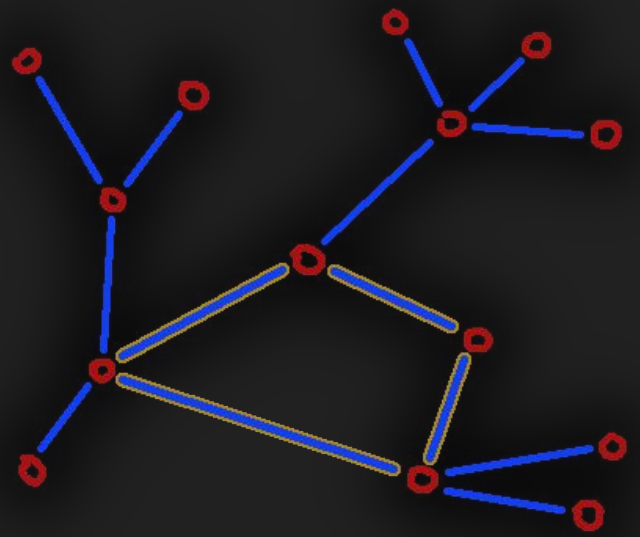


Árbol

Un grafo conexo que no tiene ciclos.

Otras definiciones equivalentes:

- grafo conexo con M=N-1
- grafo donde existe exactamente un camino simple entre cada par de vértices
- grafo conexo que deja de serlo si se elimina una arista

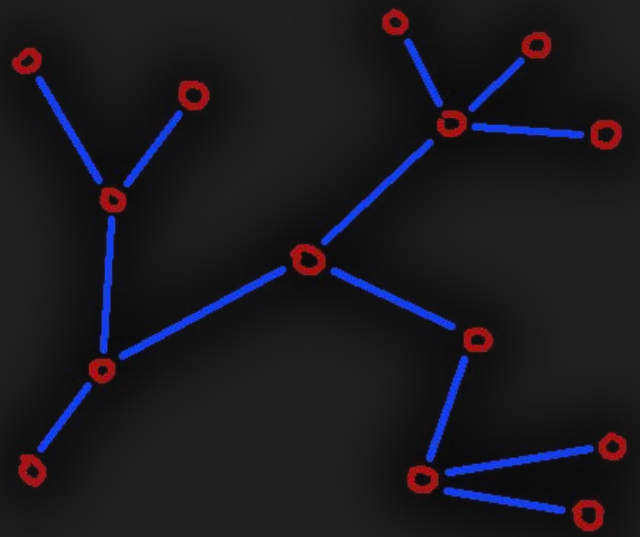


Árbol

Un grafo conexo que no tiene ciclos.

Otras definiciones equivalentes:

- grafo conexo con M=N-1
- grafo donde existe exactamente un camino simple entre cada par de vértices
- grafo conexo que deja de serlo si se elimina una arista

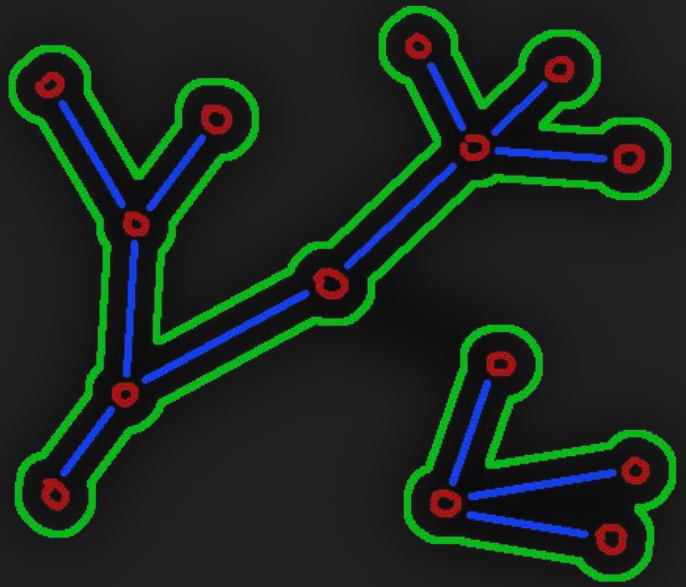


Árbol

Un grafo conexo que no tiene ciclos.

Otras definiciones equivalentes:

- grafo conexo con M=N-1
- grafo donde existe exactamente un camino simple entre cada par de vértices
- grafo conexo que deja de serlo si se elimina una arista

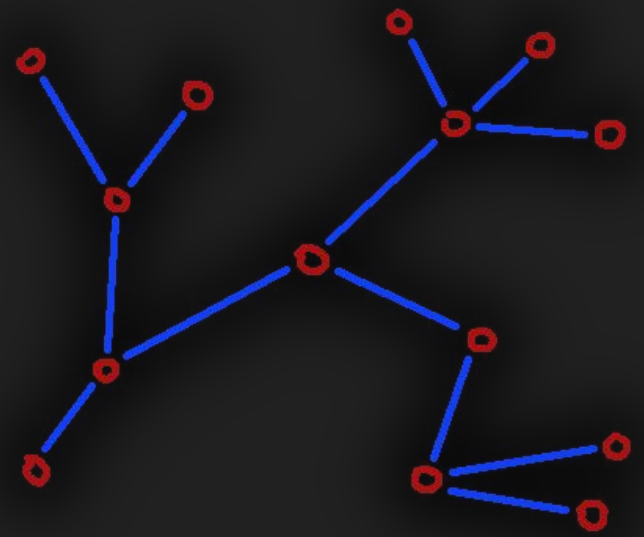


Árbol - Enraizado

Enraizar un árbol consiste en elegir un vértice como punto de partida o “raíz”, y asignar direcciones a las aristas, tal que “se alejen” de ese vértice.

Se puede usar para representar relaciones de jerarquía.

También puede ser útil enraizar un árbol para simplificar la implementación de un algoritmo.

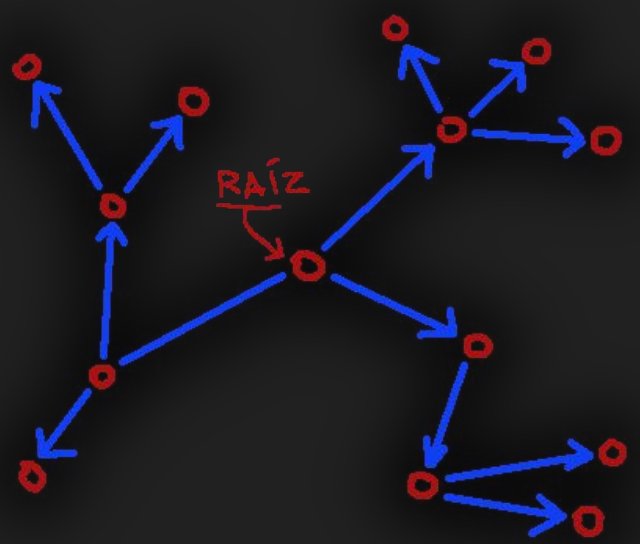


Árbol - Enraizado

Enraizar un árbol consiste en elegir un vértice como punto de partida o “raíz”, y asignar direcciones a las aristas, tal que “se alejen” de ese vértice.

Se puede usar para representar relaciones de jerarquía.

También puede ser útil enraizar un árbol para simplificar la implementación de un algoritmo.



¿Cómo codeo un grafo?

Representaciones de grafos

- Lista de adyacencia: por cada nodo, una lista con sus vecinos
- Matriz de adyacencia: matriz booleana de N por N, true en la posicion (i,j) indica que una hay una arista del vértice i al j
- Lista de aristas: un vector con todas las aristas

Por lo general usamos lista de adyacencia porque nos sirve para implementar algoritmos de “recorrido” eficientemente.

Representación: lista de adyacencia

Estructura de datos: O(N+M) en espacio

```
vector<int> grafo[MAXN];
```

Insertar arista (u->v): O(1)

```
grafo[u].push_back(v);
```

Recorrer vecinos de u: O(grado de u)

```
for (int v : grafo[u])
    visitar(v);
```

Borrar arista (u->v): O(grado de u) con vector, O(log N) con set

```
set<int> grafo[MAXN]; // no sale rápido con vector asique usamos set
grafo[u].erase(v);
```

Representación: matriz de adyacencia

Estructura de datos: O(N^2) en espacio

```
bool grafo[MAXN][MAXN];
```

Insertar arista (u->v): O(1)

```
grafo[u][v] = true;
```

Recorrer vecinos de u: O(N)

```
for (int v = 0; v < N; ++v)
    if (grafo[u][v]) visitar(v);
```

Borrar arista (u->v): O(1)

```
grafo[u][v] = false;
```

Representación: lista de aristas

Estructura de datos: O(M) en espacio

```
struct arista { int u, v; };
vector<arista> grafo;
```

Insertar arista (u->v): O(1)

```
grafo.push_back({u, v});
```

Recorrer vecinos de u: O(M)

```
for (int e : grafo)
    if (e.u == u) visitar(e.v);
```

Borrar arista (u->v): O(M) con vector, O(log N) con set

```
set<arista> grafo; // usamos set – hay que implementar el operador ‘<’ entre aristas
grafo.erase({u, v});
```

Representaciones de grafos ponderados

Para representar grafos ponderados, agregamos pesos.

Lista de adyacencia:

```
struct arista { int v, peso; }
vector<arista> grafo[MAXN];
```

Matriz de adyacencia:

```
int grafo[MAXN][MAXN]; // INF o -1 significa “no hay arista”
```

Lista de aristas:

```
struct arista { int u, v, peso; };
vector<arista> grafo;
```

Representaciones de grafos no dirigidos

Las representaciones que vimos todas son implícitamente para grafos dirigidos. Para representar grafos no dirigidos, simplemente insertamos aristas en ambos sentidos.

Lista de adyacencia:

```
grafo[u].push_back(v);
grafo[v].push_back(u);
```

Matriz de adyacencia:

```
grafo[u][v] = grafo[v][u] = true;
```

Lista de aristas:

```
grafo.push_back({u, v});
grafo.push_back({v, u});
```

Lectura de un grafo por consola

Por lo general nos dan N y M en un renglón y después una arista por renglón.

N M
u_1 v_1
u_2 v_2
…
u_M v_M

Podemos leerlo y guardarlo así:

```
int N, M; cin >> N >> M;
for (int i = 0; i < M; ++i) {
  int u, v; cin >> u >> v;
  insertar_arista(u, v);  // grafo[u].push_back(v) si usamos lista de adyacencia
}
```

Algoritmos

Recorridos

Nos permiten procesar los vértices y aristas en órdenes específicos, partiendo de un vértice: “el origen”. Los principales recorridos son:

- en profundidad (“DFS”): avanza hasta atorarse, después retrocede.
- en anchura (“BFS”): visita los que están a 1 paso, después a 2, etc.
- en distancia (“SPF” o “Dijkstra”): visita en orden de distancia según los pesos.

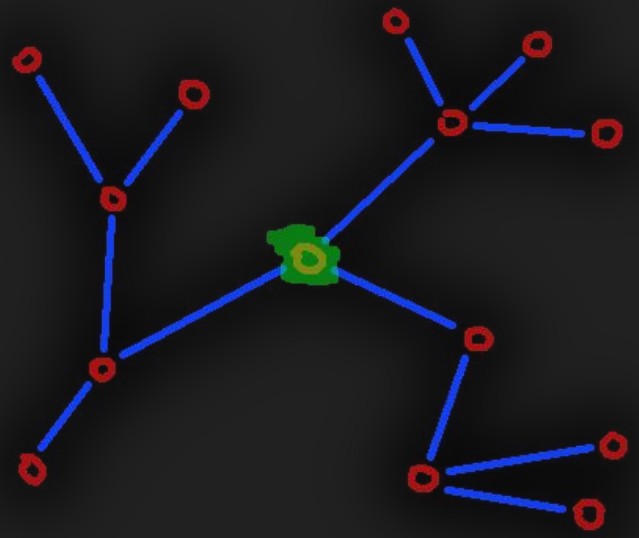
Si todas las aristas tienen el mismo peso, el recorrido en distancia es equivalente al recorrido en anchura.

Recorridos

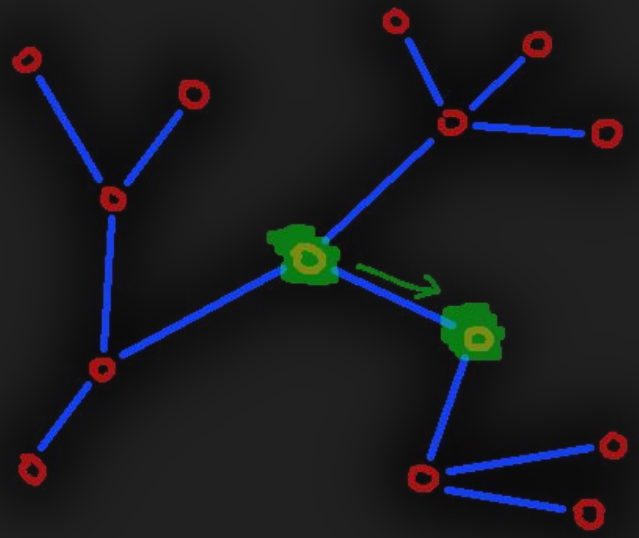
Los tres tipos tipos de recorridos se pueden expresar con este pseudocódigo. El tipo de bolsa que usemos determina cuál recorrido se hace.

```
recorrer(src) {
  b = crear_bolsa()
  insertar_bolsa(b, src)
  mientras b no está vacía {
    u = extraer_bolsa(b)
    por cada v, vecino de u que no fue visitado {
      insertar_bolsa(b, v)
    }
  }
}
```

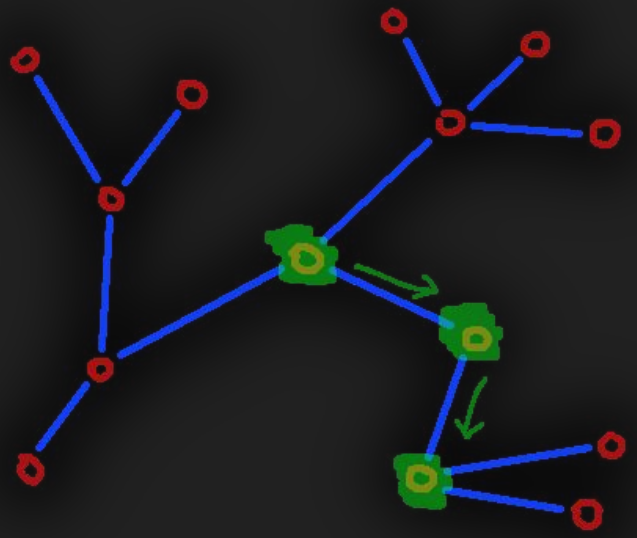
DFS (en grafo no ponderado)



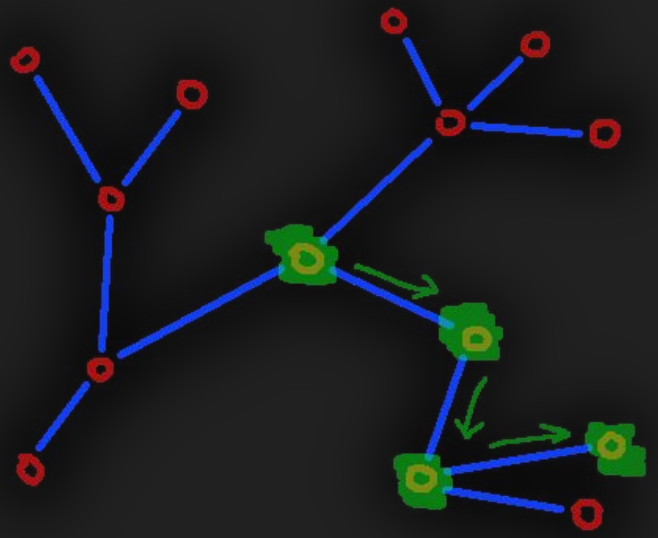
DFS (en grafo no ponderado)



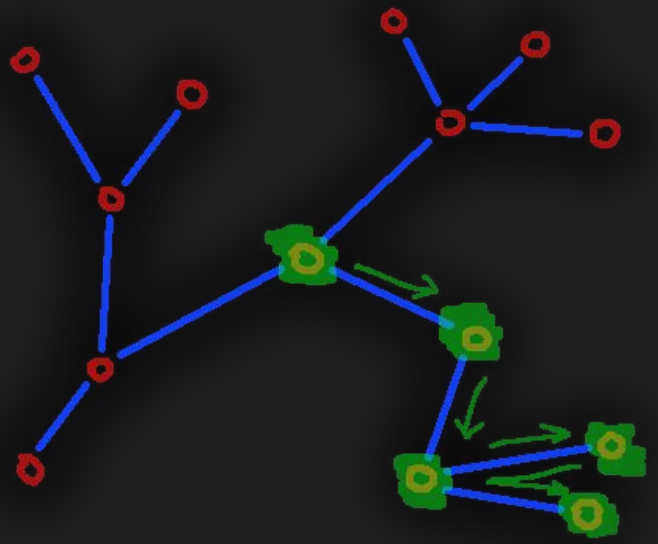
DFS (en grafo no ponderado)



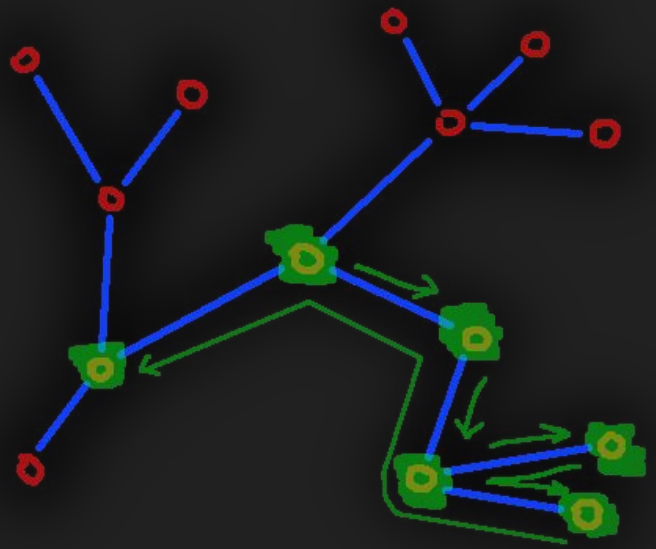
DFS (en grafo no ponderado)



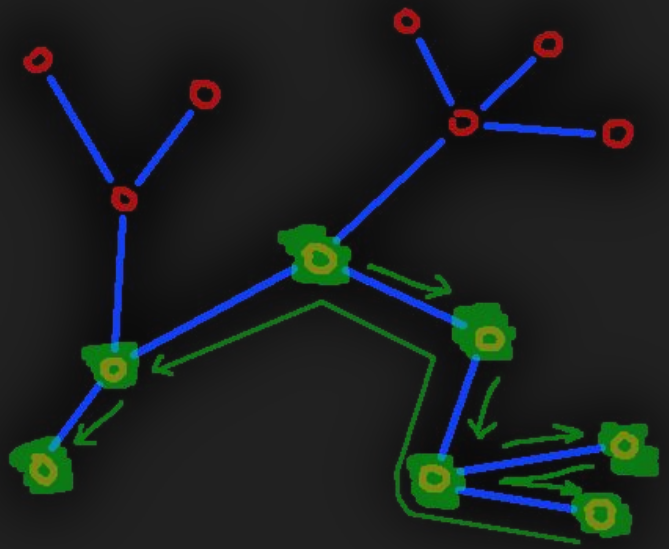
DFS (en grafo no ponderado)



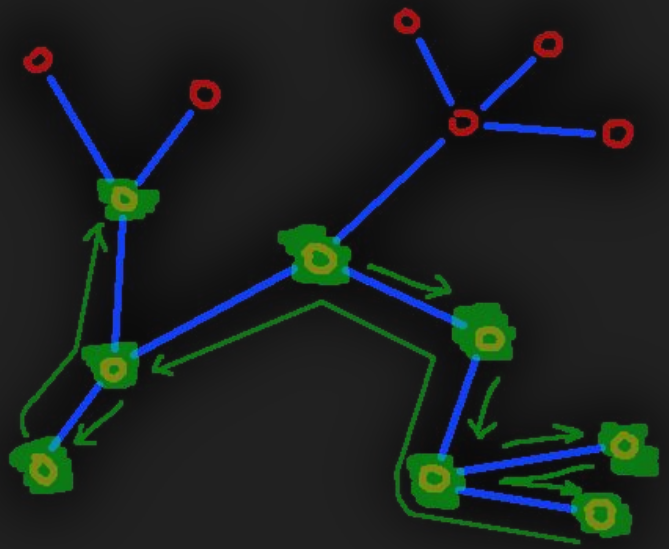
DFS (en grafo no ponderado)



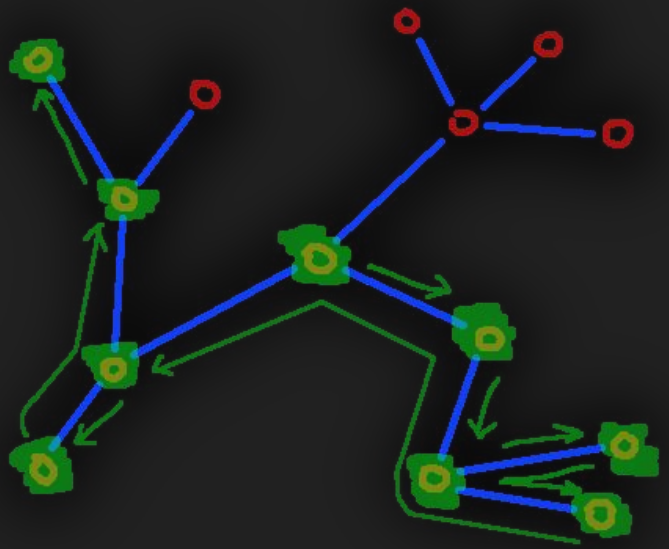
DFS (en grafo no ponderado)



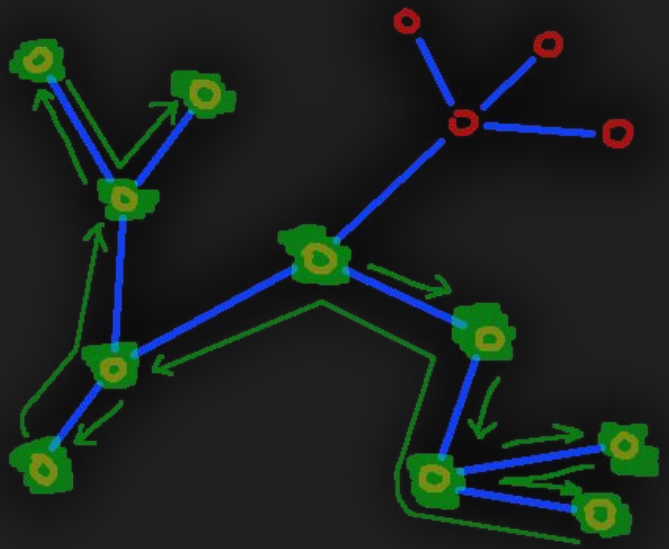
DFS (en grafo no ponderado)



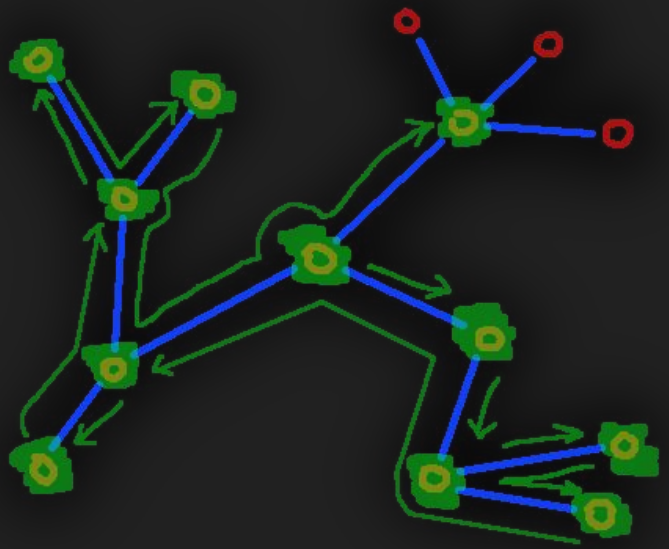
DFS (en grafo no ponderado)



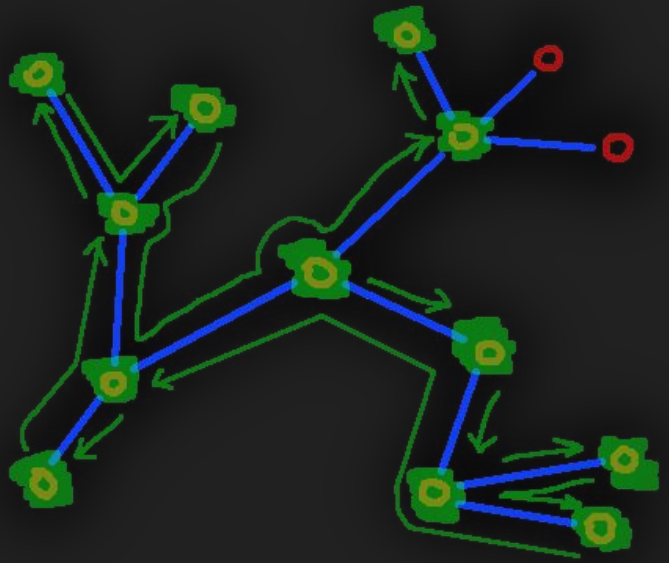
DFS (en grafo no ponderado)



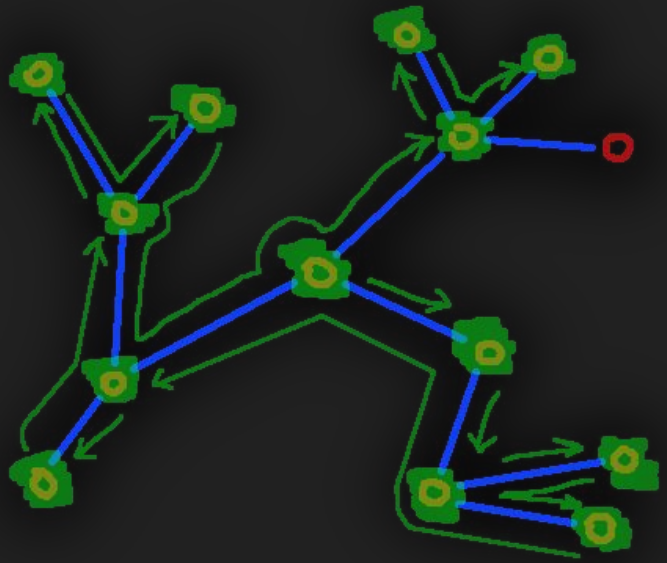
DFS (en grafo no ponderado)



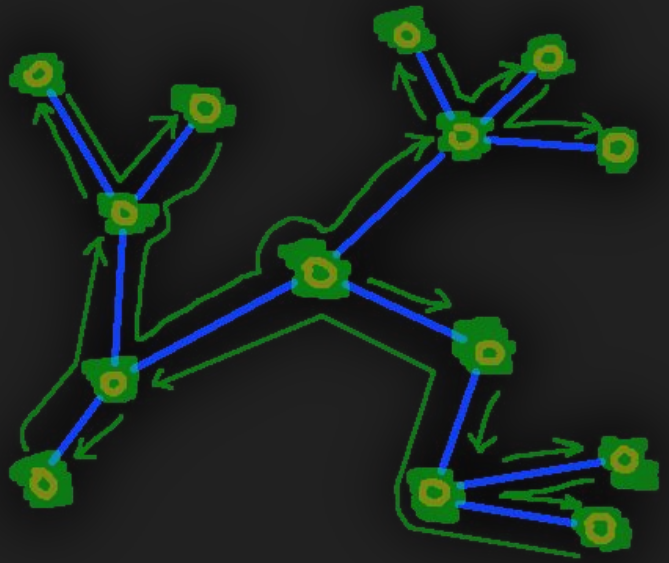
DFS (en grafo no ponderado)



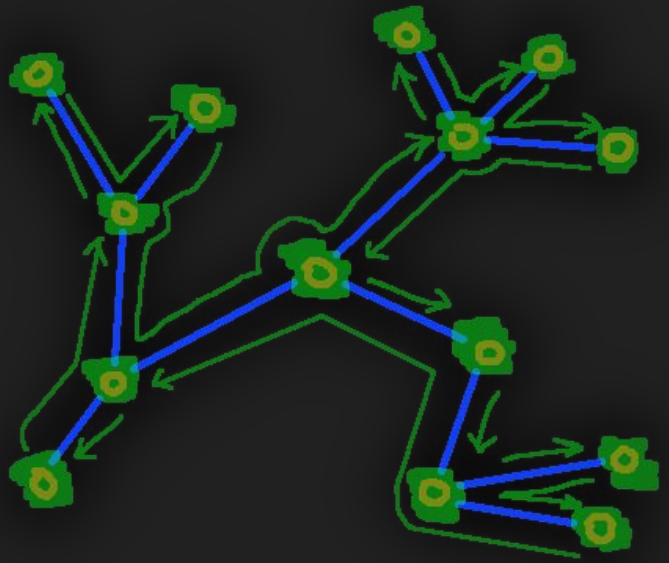
DFS (en grafo no ponderado)



DFS (en grafo no ponderado)



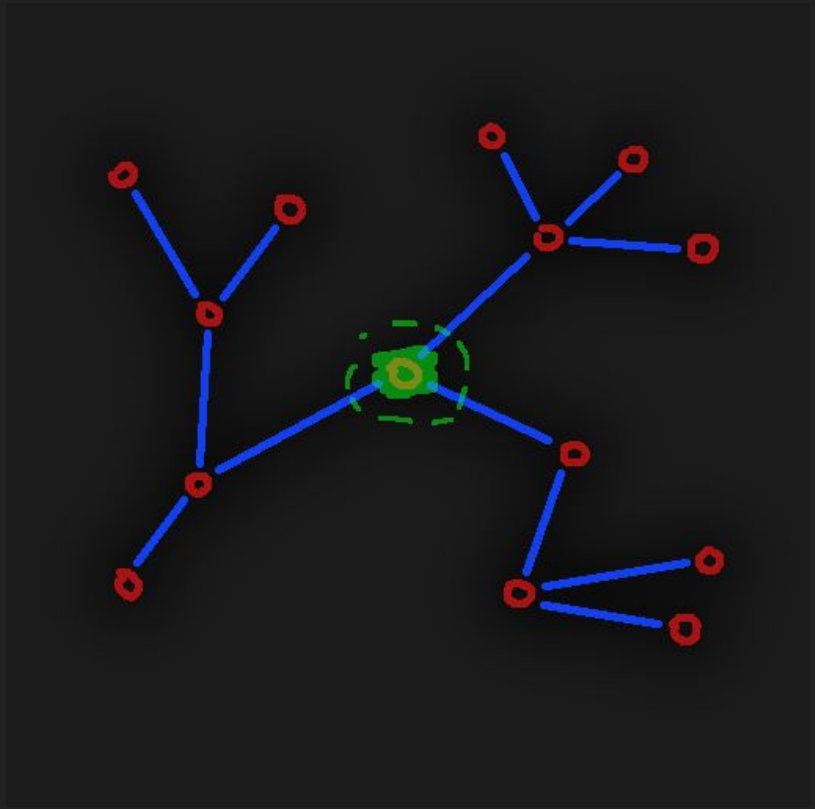
DFS (en grafo no ponderado)



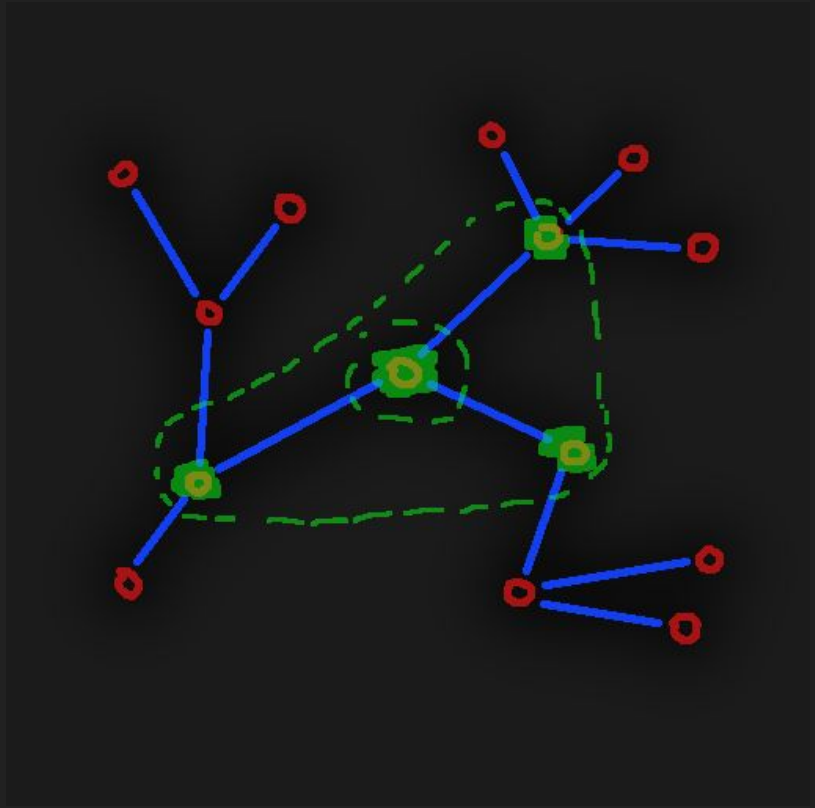
DFS (en grafo no ponderado)

```
bool visitado[MAXN];
// recorrido en profundidad
void dfs(int s) {
    fill(visitado, visitado+N, false);
    stack<int> b;
    visitado[s] = true; b.push(s);
    while (!b.empty()) {
        int u = b.top(); b.pop();
        for (int v : grafo[u]) {
            if (visitado[v]) continue;
            visitado[v] = true; b.push(v);
        }
    }
}
```

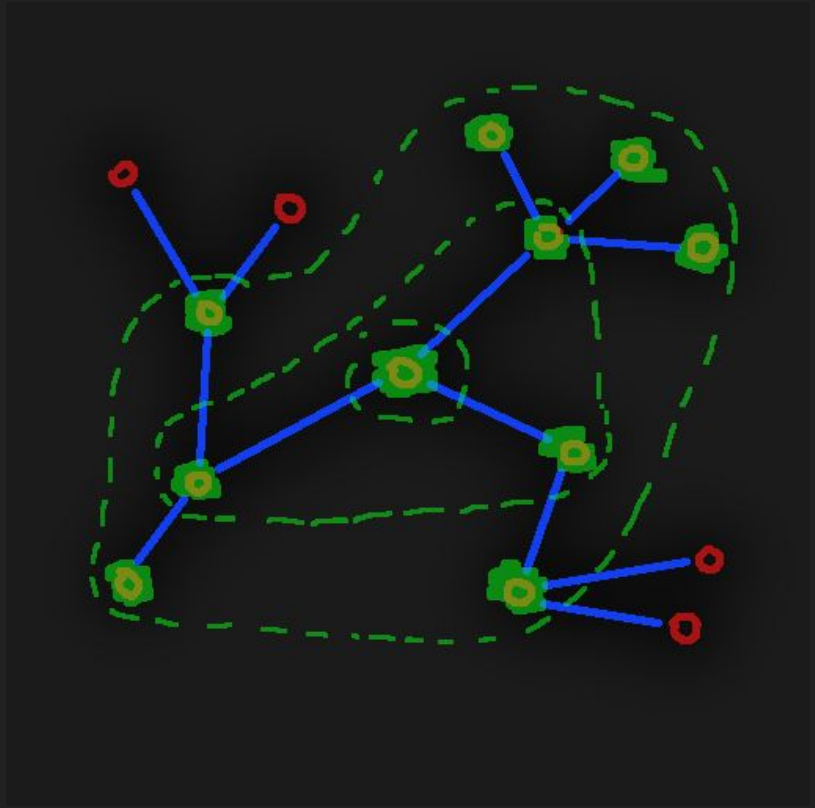
BFS



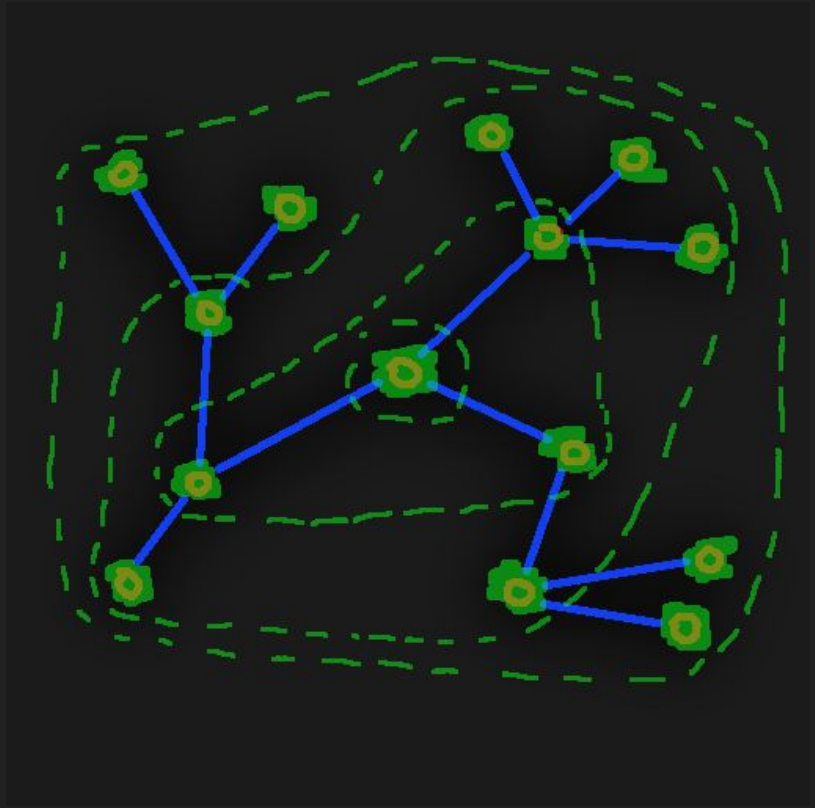
BFS



BFS



BFS



BFS (en grafo no ponderado)

```
bool visitado[MAXN];
// recorrido en anchura
void bfs(int s) {
    fill(visitado, visitado+N, false);
    queue<int> b;
    visitado[s] = true; b.push(s);
    while (!b.empty()) {
        int u = b.top(); b.pop();
        for (int v : grafo[u]) {
            if (visitado[v]) continue;
            visitado[v] = true; b.push(v);
        }
    }
}
```

SPF / Dijkstra (en grafo ponderado)

```
int D[MAXN];
struct camino { int u, d; };
bool operator< (camino a, camino b) { return a.d > b.d; }
// recorrido en distancia
void dijkstra(int s) {
    fill(D, D+N, INF);
    priority_queue<camino> b;
    D[s] = 0; b.push({s, 0});
    while (!b.empty()) {
        camino cu = b.top(); b.pop();
        int u = cu.u, du = cu.d;
        if (du > D[u]) continue;
        for (arista ve : grafo[u]) {
            int v = ve.u, dv = du + ve.d;
            if (dv >= D[v]) continue;
            D[v] = dv; b.push({v, dv});
        }
    }
}
```

SPF / Dijkstra (en grafo ponderado)

```
int D[MAXN];
struct camino { int u, d; };
bool operator< (camino a, camino b) { return a.d > b.d; }
// recorrido en distancia
void dijkstra(int s) {
    fill(D, D+N, INF);
    priority_queue<camino> b;
    D[s] = 0; b.push({s, 0});
    while (!b.empty()) {
        camino cu = b.top(); b.pop();
        int u = cu.u, du = cu.d;
        if (du > D[u]) continue;
        for (arista ve : grafo[u]) {
            int v = ve.u, dv = du + ve.d;
            if (dv >= D[v]) continue;
            D[v] = dv; b.push({v, dv});
        }
    }
}
```

```
recorrer(src) {
    b = crear_bolsa()
    insertar_bolsa(b, src)
    mientras b no está vacía {
        u = extraer_bolsa(b)
        por cada v, vecino no visitado de u {
            insertar_bolsa(b, v)
        }
    }
}
```

Componentes conexas

Los algoritmos de recorrida visitan todos los vértices de la componente conexa del origen. Esto da una forma sencilla de contarlas.

```
bool visitado[MAXN];
int componente[MAXN], contador = 0;
…
void dfs(int s) {
    // dfs modificado que no rellena “visitado” con false
    // tambien asigna componente[u] = contador; dentro del while
}
…
int main() {
  …
  for (int u = 0; u < N; ++u) {
    if (!visitado[u]) {
      dfs(u);
      contador++;
    }
  }
}
```

Componentes fuertemente conexas

Es bastante más complicado :-) … googlear “algoritmo de kosaraju” o “algoritmo de tarjan de componentes fuertemente conexas”. La explicación completa es más larga, pero ambos hacen un DFS modificado.