DFS simple:

```
public ListaEnlazadaGenerica<T> dfs(Grafo<T> grafo){
	ListaEnlazadaGenerica<T> lista = new ListaEnlazadaGenerica<T>();
	boolean[] marca = new boolean[grafo.listaDeVertices().tamanio()+1];
	for (int i=0; i<grafo.listaDeVertices().tamanio(); i++) {
		if (!marca[i]) {
			dfs(lista, grafo, i, marca);
		}
	}
	return lista;
}

public void dfs(ListaEnlazadaGenerica<T> lista, Grafo<T> grafo, int i, boolean[]
marca) {
	marca[i] = true;
	Vertice<T> v = grafo.listaDeVertices().elemento(i);
	lista.agregarEn(v.dato(), lista.tamanio());
	ListaGenerica<Arista<T>> adys = grafo.listaDeAdyacentes(v);
	adys.comenzar();
	while(!adys.fin()) {
		int j = adys.proximo().verticeDestino().getPosicion();
		if (!marca[j]) {
			this.dfs(lista, grafo, j, marca);
		}
	}
}
```

BFS Simple:

```
public ListaEnlazadaGenerica<T> bfs(Grafo<T> grafo) {
	ListaEnlazadaGenerica<T> lista = new ListaEnlazadaGenerica<T>();
	boolean[] marca = new boolean[grafo.listaDeVertices().tamanio()+1];
	for (int i=0; i<grafo.listaDeVertices().tamanio(); i++) {
		if (!marca[i]) {
			bfs(i,grafo,lista, marca);
		}
	}
	return lista;
}

public void bfs(int i, Grafo<T> grafo, ListaEnlazadaGenerica<T> lista, boolean[]
marca) {
	ListaGenerica<Arista<T>> ady = null;
	ColaGenerica<Vertice<T>> q = new ColaGenerica<Vertice<T>>();
	q.encolar(grafo.listaDeVertices().elemento(i));
	marca[i] = true;
	while(!q.esVacia()) {
		Vertice<T> v = q.desencolar();
		lista.agregarEn(v.dato(), lista.tamanio());
		ady = grafo.listaDeAdyacentes(v);
		ady.comenzar();
		while(!ady.fin()) {
			Arista<T> arista = ady.proximo();
			int j = arista.verticeDestino().getPosicion();
			if (!marca[j]) {
				Vertice<T> w = arista.verticeDestino();
				marca[j] = true;
				q.encolar(w);
			}
		}
	}
}
```

Kosaraju

Encuentra las componentes fuertemente conexas de un grafo dirigido:

1. Apila los vértices de G con DFS Post-Orden.
2. Invierte los arcos (G^R).
3. Aplica DFS a G^R en el orden de desapile del DFS de G.
4. Cada arbol de expansión resultante del DFS a G^r es una componente fuertemente conexa.

Ordenación Topológica

La Ordenación Topológica aplica a los Grafos Acíclicos Dirigidos.

El algoritmo puede implementarse usando un arreglo $O(|V|^2)$, o una pila/cola $O(|V| + |A|)$, o utilizando DFS y marcando los vértices en post-orden y apilandolos $O(|V| + |A|)$.

Caminos mínimos

Grafos	BFS $O(\|V\| + \|E\|)$	Dijkstra $O(\|E\| \log \|V\|)$	Dijkstra modificado $O(\|V\| * \|E\|)$	Dijkstra Optimizado $O(\|V\| + \|E\|)$
No pesado	Óptimo	Correcto	Malo	Incorrecto con ciclos
Pesado	Incorrecto	Óptimo	Malo	Incorrecto con ciclos
Pesos negativos	Incorrecto	Incorrecto	Óptimo	Incorrecto con ciclos
Pesados Acíclicos	Incorrecto	Correcto	Malo	Óptimo

BFS – Grafos no pesados

Información mantenida por cada vértice:

- D_V: Distancia mínima desde el origen *s*. Inicialmente infinita.
- P_V: Vértice por donde cruzó al llegar.

Pasos:

- Encolar el origen. Establecer D_S en 0.
- Mientras la cola no esté vacia:
  - Desencolar un vertice *u*.
  - Recorrer los adyacentes *w*.
  - Si D_W es infinito, cambiarlo a $D_U + 1$, y establecer P_W como *u*.

Dijkstra – Grafos pesados

Información mantenida por cada vértice:

- D_V: Distancia mínima desde el origen *s*. Inicialmente infinita.
- P_V: Vértice por donde cruzó al llegar.

- Conocido: dato booleano que indica si ya se procesó.

Pasos:

- Dado un vértice de origen *s*, elegir el vértice *v* que esté a menor distancia de *s*, dentro de los vértices no procesados.
- Marcar *v* como procesado.
- Actualizar la distancia de *w* adyacente a *v*:
    - Si $D_W > D_V + c(v,w) \rightarrow D_W$ pasa a ser $D_V + c(v,w)$, y P_W pasa a ser *v*.

Tiempo de ejecución - Dijkstra

O(|V|²)

```
Dijkstra(G, s){
    para (cada vértice v Є V) {                           // |V| iteraciones
        Dv = ∞; Pv = 0;
        Ds = 0;
    }
    para (cada vértice v Є V) {                           // |V| iteraciones
        u = vérticeDesconocidoMenorDist;                  // |V| iteraciones
        Marcar u como conocido;
        para (cada vértice w Є V adyacente a u) {         // |E| iteraciones
            si (w no está conocido && Dw > Du + c(u,w) ) {
                Dw = Du + c(u,w);
                Pw = u;
            }
        }
    }
}
```

Debido a las |V| iteraciones dentro de las |V| iteraciones, el tiempo de ejecución es $|V|^2 + |E| \leq O(|V|^2)$.

O(|E| log |V|)

Si almacenamos las distancias en una heap, vérticeDesconocidoMenorDist pasa a ser log|V|, lo que nos da |V| log |V|, y como se debe reorganizar la heap si se cumple la condición para los adyacentes, nos da |E| log |V|.

El costo total del algoritmo es (|V| log |V| + |E| log |V|) ≤ O(|E| log |V|).

Para evitar reorganizar la Heap, se inserta el vertice *w* y su nuevo valor D_w cada vez que este modifica. El tamaño de la heap puede entonces crecer hasta |E|. Y dado que $|E| \leq |V|^2$, log |E| ≤ 2 log |V|. El costo total no varía.

Dijkstra Modificado – Grafos pesados con pesos negativos

Pasos:

- Encolar el vértice origen *s*.
- Procesar la cola:
  - Desencolar un vertice.
  - Actualizar la distancia de los adyacentes siguiendo el mismo criterio de Dijkstra.
    - Si $D_W > D_V + c(v,w) \rightarrow D_W$ pasa a ser $D_V + c(v,w)$, y P_W pasa a ser *v*.
  - Si *w* no está en la cola, encolarlo.

El costo total del algoritmos es O(|V| * |E|).

Dijkstra Optimizado – Grafos pesados acíclicos

Estrategia: Orden topológico.

- La selección de cada vértice se realiza siguiendo el orden topológico.
- Esto funciona debido a que al seleccionar un vertice *v*, no se va a encontrar una distancia menor porque ya se procesaron todos los caminos que llegan hasta él.

El costo total del algoritmo es O(|V| + |E|).

Algoritmo de Floy – Distancia mínima entre todos los pares de vértices.

Estrategia:

- Llevar dos matrices D y P, ambas de |V| x |V|.

El costo total del algoritmo es $O(|V|^3)$.

Toma cada vértice como intermedio para calcular los caminos: Calcula la distancia entre los vértices *i* y *j*, pasando por un vertice *k*.

Árbol de expansión mínima

Dado un grafo no dirigido y conexo, el arbol de expansión mínima es un arbol formado por las aristas de G que conectan todos los vértices con un costo total mínimo.

Algoritmo de Prim

Construye el árbol haciéndolo crecer por etapas.

En las siguientes etapas:

1. Se selecciona la arista (*u*,*v*) de mínimo costro que cumpla: *u* existe en el árbol y *v* no existe.
2. Se agrega al árbol la arista seleccionada en a) (es decir, ahora el vértice *v* existe en el arbol).

3. Se repite a) y b) hasta que se hayan tomado todos los vértices del grafo.

Tiempo de Ejecución

Se hacen las mismas consideraciones que para el algoritmo de Dijkstra.

- Si se implementa con una tabla secuencial, el costro es $O(|V|^2)$.
- Si se implementa con heap, el costo es O(|E| log |V|).

Algoritmo de Kruskal

Selecciona las aristas en orden creciente según su perso y las acepta en caso de que no originen un ciclo.

El invariante que usa indica que en cada punto del proceso, dos vertices pertenecen al mismo conjunto si y solo sí están conectados.

Si dos vertices están el mismo conjunto, la arista ente ellos será rechazada porque al aceptarla forma un ciclo.

- Inicialmente cada vértice pertenece a su propio conjunto.
- Al aceptar una arista se realiza la Union de estos dos conjuntos.
- Las aristas se organizan en una heap, para ir recuperando la de mínimo costo en cada paso.

El tamaño de la heap es |E| y extraer cada arista lleva O(log |E|). El tiempo de ejecución total es O(|E| log |E|).

Dado que |E| ≤ |V|2, log |E| ≤ 2 log |V|, por lo tanto el costo toal del algorismo es O(|E| log |V|).