Algoritmi

BFS

Esempio stampa per ogni nodo di ogni livello il valore del nodo e il livello

```
public static <V> void bfs(Graph<V> g, Graph.GraphNode<V> source){
    //reset
    for(Graph.GraphNode<V> n: g.getNodes()){
        n.timestamp=0;
        n.state=Graph.GraphNode.Status.UNEXPLORED;
    }
    //check
    if(source.state!=Graph.GraphNode.Status.UNEXPLORED){
        System.out.println("Livello "+source.value +": " + source.timestamp);
        return;
    }
    //creo Coda per inserire i nodi dentro
    Queue<Graph.GraphNode<V>> que= new ArrayDeque<Graph.GraphNode<V>
    //inserisco il source nella coda e ne aggiorno lo stato
    source.state=Graph.GraphNode.EXPLORED;
    que.add(source);
    //rimuovo dalla lista e inserisco man mano i nodi collegati al nodo rimosso
    while(!que.isEmpty()){
        Graph.GraphNode<V> nodo= que.remove();
        System.out.println("Livello " + nodo.value + ": "+ nodo.timestamp);
        //ciclo i vicini del nodo rimosso per aggiungerli nella coda
        for(Graph.GraphNode<V> n: nodo.outEdges){
            if(n.state==Graph.GraphNode.Status.UNEXPLORED){
```

```
            que.add(n);
            n.timestamp=nodo.timestamp+1;
            n.state=Graph.GraphNode.Status.EXPLORED;
          }
        }
    }
    //reset
    for (Graph.GraphNode e: g.getNodes(){
        e.timestamp=0;
        e.state=Graph.GraphNode.Status.UNEXPLORED;
     }
    }
}
```

BFS per distanza massima

```
public static <V> int max_dist(Graph<V> g, Graph.GraphNode<V> source){
    if(source.state!=Graph.GraphNode.Status.UNEXPLORED){
        return 0;
    }
    //reset
    for(Graph.GraphNode<V> e: g.getNodes()){
        e.timestamp=0;
        e.state= Graph.GraphNode.Status.UNEXPLORED;
    }
    //variabile per altezza
    int ret =0;
    //bsf
    Queue<Graph.GraphNode<V>> que= new ArrayDeque<Graph.GraphNode<V>
    source.state=Graph.GraphNode.Status.EXPLORED;
    que.add(source);
    while(!que.isEmpty()){
        Graph.GraphNode<V> node= que.remove();
        for (Graph.GraphNode<V> e: node.outEdges){
            if (e.state==Graph.GraphNode.Status.UNEXPLORED){
```

```
            e.state=Graph.GraphNode.Status.EXPLORED;
            e.timestamp=node.timestamp+1;
            que.add(e);
          }
       }
       if(node.timestamp>ret)
         ret= node.timestamp;
    }
    //reset
    for(Graph.GraphNode<V> e: g.getNodes()){
       e.timestamp=0;
       e.state= Graph.GraphNode.Status.UNEXPLORED;
    }
    return ret;

}
```

DFS

```
public static <V> void dfsVisit(Graph.GraphNode<V> n){
    for (Graph.GraphNode<V> e: n.outEdges){
       if(e.state==Graph.GraphNode.Status.UNEXPLORED){
          dfsVisit(e);
       }
    }
    n.state=Graph.GraphNode.Status.EXPLORED;
}
//esplora tutti i nodi se non connessi
public static <V> void dfs(Graph<V> g){
    for(Graph.GraphNode<V> e: g.getNodes()){
       if(e.state==Graph.GraphNode.Status.UNEXPLORED){
          dfsVisit(e);
       }
```

```
    }
}
```

DDFS per rilevare cicli su grafi orientati

Qui ho anche salvato in ordine topologico su una lista

```
private static <V> int DDFS(Graph.GraphNode<V> nd, LinkedList<GraphNode<V
    //funzione ritorna piu di 0 se rileva cicli

    if(nd.state== Graph.GraphNode.Status.EXPLORED){
        return 0;
    }
    //caso in cui trova ciclo
    if(nd.state== Graph.GraphNode.Status.EXPLORING){
        return 1;
    }
    int ret=0;
    nd.state=Graph.GraphNode.Status.EXPLORING;
    for ( Graph.GraphNode<V> e: nd.outEdges){
        ret+=DDFS(e,ts);
    }
    nd.state=Graph.GraphNode.Status.EXPLORED;
    ts.addFirst(nd);
    return ret;
}
```

Find strongly components

```
    public static <V> void transposedDFS(Graph.GraphNode<V> nd,
    LinkedList<Graph.GraphNode<V>> ts{
        if(nd.state==Graph.GraphNode.Status.EXPLORED)
            return;
        if(nd.state==Graph.GraphNode.Status.EXPLORING)
```

```
            return;
        nd.state=Graph.GraphNode.Status.EXPLORING;
        for (Graph.GraphNode<V> n: nd.inEdges){
            transposedDFS(n,ts);
        }
        nd.state = Graph.GraphNode.Status.EXPLORED;
        ts.addLast(nd);

    }
```

```
public static <V> void strongConnectedComponents(Graph<V> g){
    LinkedList<Graph.GraphNode<V>> stack= new LinkedList<Graph.GraphNode<
    for( Graph.GraphNode<V> n: g.getNodes()){
        if (n.state == Graph.GraphNode.Status.UNEXPLORED) {
        DDFS(n, stack);
    }
    }
    g.resetStatus();

    while (!stack.isEmpty()){
        Graph.GraphNode<V> v= stack.removeLast();
        if(v.state==Graph.GraphNode.Status.UNEXPLORED){
            LinkedList<Graph.GraphNode<V>> ret = new LinkedList<Graph.GraphNo
            transposedDFS(v,ret);
            System.out.println("Nodi Fortemente connessi per : "+v.value+":");
            for (Graph.GraphNode<V> cur: ret){
                System.out.print(cur.value+" ");
            }
            System.out.println("");
        }
    }
}
```

Ordinamento Albero

```
public LinkedList ordina(BST t) {
   if(t==null) return null;
   LinkedList<Integer> list= new LinkedList<Integer>();
   aiut(t.root(),list);
   return list;
}
private static void aiut(Node t, LinkedList<Integer> list){
   if(t==null) return;
   aiut(t.right, list);
   list.add(t.key);
   aiut(t.left,list);
}
//se fosse ordinamento crescente
//aiut(t.left,list)
//list.add(t.key)
//aiut(t.right,list)
```

Altezza albero

Albero bilanciato

```
 public int calculateHeight(Node node) {
      if (node == null) {
         return 0;  // Un albero vuoto ha altezza 0
      } else {
         // Calcola l'altezza del sottoalbero sinistro e destro
         int leftHeight = calculateHeight(node.left);
         int rightHeight = calculateHeight(node.right);

         // L'altezza dell'albero corrente è il massimo tra i due sottoalberi + 1
         return Math.max(leftHeight, rightHeight) + 1;
      }
   }
```

```
public boolean isBalanced(Node node) {
    return checkHeight(node) != -1;
}

// Metodo helper per calcolare l'altezza e verificare se è bilanciato
private int checkHeight(Node node) {
    if (node == null) {
        return 0;  // Un albero vuoto ha altezza 0
    }

    // Calcola l'altezza del sottoalbero sinistro
    int leftHeight = checkHeight(node.left);
    if (leftHeight == -1) return -1; // Se il sottoalbero sinistro non è bilanciato

    // Calcola l'altezza del sottoalbero destro
    int rightHeight = checkHeight(node.right);
    if (rightHeight == -1) return -1; // Se il sottoalbero destro non è bilanciato

    // Se la differenza tra le altezze è maggiore di 1, l'albero non è bilanciato
    if (Math.abs(leftHeight - rightHeight) > 1) {
        return -1; // L'albero non è bilanciato
    }

    // Restituisce l'altezza del nodo corrente
    return Math.max(leftHeight, rightHeight) + 1;
}
```

Dijkstra

```
public static <V> void sssp(Graph<V> g, Node<V> source){
    //strutture sono una MinHeap come priority queue e una hash map che mappa
    //con l'ultimo peso raggiunto
    MinHeap<Node<V>> pqueue= new MinHeap<Node<V>>();
    HashMap<Node<V>, HeapEntry<Node<V>>>dist= new HashMap<Node<V>,
```

```
    HashMap<Node<V>, Node<V>> predecessor = new HashMap<>();
    final int INFINITY=100000;
    //inizializzo il source a 0 e quelli vicini a infinity
    for(Node<V> u: g.getNodes(){
        HeapEntry<Node<V>> hu=pqueue.insert(u==source? 0: INFINITY, u);
        dist.put(u,hu);
        predecessor.put(u, null);
    }
    while(!pqueue.isEmpty()){
        HeapEntry<Node<V>> hp= pqueue.removeMin();
        Node<V> u= hp.getValue();

        for(Edge<V> e: g.getEdges(u)){
          int newDist = dist.get(u).getKey() + e.getWeight();
            Node<V> v= e.getTarget();
            if(newDist <dist.get(v).getKey()){
                pqueue.replaceKey(dist.get(v),dist.get(u).getKey()+e.getWeight());
                predecessor.replace(v,u);
            }
        }
    }
    System.out.print("Distanze dal nodo " + source + "[");
    for(Node<V> u: g.getNodes()){
        System.out.print( u + ": " + dist.get(u).getKey());
    }
    System.out.print("]");
    System.out.println("");
}

public static <V> void asp (Graph<V> g){
    for( Node<V> n: g){
        sssp(g,n);
        return;
    }
}
```

```
public static<V> void sssp(Graph<V> g, Node<V> source){
MinHeap<Nodo<V>> pqueue= new MinHeap<Nodo<V>>
HashMap<Nodo<V>,HeapEntry<Nodo<V>> dist
HashMap<Nodo<V>,Nodo<V>> predecessor;
//inizializzo
for(Node<V> v: g.getNodes()){
HeapEntry hp=pqueue.insert(v==source? 0, INFINITY, v);
dist.put(u,hp);
predecessor(u,null);
}
while(!pqueue.isEmpty(){
HeapEntry hp= pqueue.removeMin();
Node<V> n= hp.getValue();
for(Edge<V> e: g.getEdges(n)){
Node<V> v= e.target();
int newDist= dist.get(n).getKey()+e.getWeight();
if(newDisy<dist.get(v).getKey())
pqueue.replace(dist.get(v),newDist);
predecessor.replace(v,u);
}
}
}
public void asp(Graph g){
for nodi n in g:
sssp(g,n
```

}

Teoria

Heap

minimale: nodo genitore più piccolo dei figli
massimale : nodo genitore più grande dei figli

Weighted things

PRIM
Algoritmo di Prim:
1 Partenza da un nodo iniziale:

Si sceglie un nodo di partenza arbitrario (in genere il nodo 0, o qualunque altro n
Si imposta il suo costo (peso) a 0 e quello di tutti gli altri nodi a infinito,
indicando che inizialmente non sono connessi.
Esaminare i nodi vicini:

2 Si osservano le connessioni (gli archi) tra il nodo attuale e i nodi vicini,
scegliendo quello con il peso minore tra i vicini non ancora visitati.
Aggiungere il nodo collegato a peso minore:

3 Si aggiunge il nodo connesso a costo minore all'albero di copertura
(o alla lista dei nodi già visti).
Ripetere il processo:

4 Si controllano tutte le connessioni dai nodi nella lista dei nodi già visti e si
sceglie, tra tutte, quella con il peso minore verso un nodo non ancora visitato.
Si aggiunge questo nodo alla lista e si continua, ripetendo il
processo fino a quando tutti i nodi del grafo non sono stati inclusi nell'albero di
copertura.

KRUSAL
1) Individua l'arco minore
Il primo passo dell'algoritmo è ordinare tutti gli archi del grafo in base al loro pes
dal più piccolo al più grande.
Si parte quindi dall'arco con il peso minore, che verrà

considerato per primo nell'albero di copertura minimo.
2) Aggiungi i nodi dell'arco nello spanning tree
Dopo aver individuato l'arco più leggero, si verifica se i due nodi collegati dall'arco appartengono già all'albero (spanning tree).
Se i nodi non fanno già parte dello stesso insieme (cioè non sono collegati tra di loro nel MST formato fino a quel momento), si aggiungono entrambi i nodi all'MST.
Se i due nodi sono già collegati (nello stesso insieme), l'arco viene scartato per evitare di formare un ciclo.
3) Cerca il prossimo arco minore e aggiungi i due nodi
(aggiungi l'arco solo se almeno uno dei due nodi è da visitare)
Si ripete il processo per l'arco successivo in ordine crescente di peso.
Si verifica nuovamente se i nodi dell'arco sono già collegati all'albero di copertura minimo:
Se uno o entrambi i nodi non sono ancora inclusi nell'albero, si aggiunge l'arco a
Se entrambi i nodi sono già connessi, l'arco viene scartato per evitare la formazi di cicli.
L'obiettivo è sempre quello di aggiungere un arco che colleghi nodi nuovi all'albe senza chiudere anelli o cicli.
4) Continua finché non ci sono più archi validi
L'algoritmo continua ad esaminare gli archi in ordine crescente di peso.
Si ripete il processo finché non si sono inclusi V - 1 archi nell'MST, dove V è il numero di nodi del grafo. A quel punto, l'albero di copertura minimo è completo.
Quando tutti i nodi del grafo sono stati inclusi e non ci sono più archi che possan essere aggiunti senza formare cicli, l'algoritmo termina.

Attraversamenti

In-order-traversal (simmetrico)
Attraversa il sottoalbero sinistro.
Visita il nodo corrente.
Attraversa il sottoalbero destro.

Post-order-traversal
Attraversa il sottoalbero sinistro.
Attraversa il sottoalbero destro.
Visita il nodo corrente.

Pre-order-traversal
Visita il nodo corrente.
Attraversa il sottoalbero sinistro.
Attraversa il sottoalbero destro.

Merge

```
function merge_sort(arr):
    if length(arr) <= 1:  # Se l'array ha un solo elemento, è già ordinato
        return arr

    mid = length(arr) // 2  # Trova il punto centrale per dividere l'array
    left_half = merge_sort(arr[0:mid])  # Ordina la prima metà ricorsivamente
    right_half = merge_sort(arr[mid:])  # Ordina la seconda metà ricorsivamente

    return merge(left_half, right_half)  # Combina le due metà ordinate


function merge(left, right):
    result = []  # Array temporaneo per memorizzare il risultato
    i = 0  # Indice per l'array 'left'
    j = 0  # Indice per l'array 'right'

    while i < length(left) and j < length(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
```

```
# Se ci sono elementi rimasti in 'left' o 'right', aggiungili a 'result'
result += left[i:]
result += right[j:]

return result
```