APPUNTI ALGO GRAFI

BFS: visita in ampiezza

Pseudo-codice

```
BFS (G, s)
    for ogni vertice v ∈ G.V - {s}
        v.color = WHITE
        v.d = ∞
        v.π = NIL
    s.color = GRAY
    s.d = 0
    s.π = NIL
    Q = ∅
    ENQUEUE(Q, s)
    while Q ≠ ∅
        u = DEQUEUE(Q)
        for ogni vertice v ∈ G.Adj[u]
            if v.color == WHITE
                v.color = GRAY
                v.π = u
                v.d = u.d + 1
                ENQUEUE(Q, v)
        u.color = BLACK
```

Descrizione:

È un algoritmo di esplorazione di grafi che visita i nodi "a livelli", partendo da un nodo sorgente ed esplorando tutti i vicini a distanza crescente. Utilizza una coda (struttura FIFO) per gestire i nodi da visitare, garantendo che i nodi più vicini alla sorgente siano processati prima di quelli più lontani. È ideale per trovare cammini minimi nei grafi non pesati.

Tempo di esecuzione nel caso peggiore:

$O(|V|+|E|)$, dove V è il numero di nodi e E il numero di archi. Ogni nodo viene inserito nella coda una volta e ogni arco viene esaminato una volta.

Problemi risolvibili con BFS:

1. **cammino minimo in grafi non pesati**: BFS trova il percorso con il minor numero di archi tra due nodi in tempo lineare;
2. **verifica se un grafo è bipartito**: assegnando livelli durante l'esplorazione, si controlla l'assenza di archi tra nodi dello stesso livello (cicli dispari);
3. **ricerca di componenti connesse in grafi non orientati**: BFS visita tutti i nodi raggiungibili da una sorgente, identificando una componente connessa per esecuzione.

DFS: visita in profondità

Pseudocodice

```
DFS(G)
    for ogni vertice u ∈ G.V
        u.color = WHITE
        u.π = NIL
    time = 0
    for ogni vertice u ∈ G.V
        if u.color == WHITE
            DFS-VISIT(G, u)

DFS-VISIT(G, v)
    time = time + 1
    v.d = time
    for ogni vertice u ∈ G.Adj[v]
        if u.color == WHITE
            u.π = v
            DFS-VISIT(G, u)
    v.color = BLACK
    time = time +1
    v.f = time
```

Descrizione:

È un algoritmo di esplorazione di grafi che visita i nodi "in profondità", seguendo un percorso il più lontano possibile da un nodo iniziale prima di effettuare backtracking. Utilizza una struttura dati a stack (implicita nella ricorsione o esplicita nell'implementazione iterativa) per gestire i nodi da visitare. L'obiettivo è esplorare tutti i rami partendo da un nodo sorgente, marcando i nodi come visitati per evitare cicli.

Tempo di esecuzione:

$O(|V| + |E|)$, dove V è il numero di nodi (vertici) e E il numero di archi. Ogni nodo e arco viene processato una sola volta.

Problemi risolvibili con DFS:

1. **ordinamento topologico**: (per DAG), DFS genera un ordinamento lineare dei nodi in cui ogni nodo precede i suoi successori;
2. **rilevazione di cicli in un grafo diretto**: durante l'esplorazione, se si incontra un arco "all'indietro", il grafo contiene un ciclo.
3. **componenti fortemente connesse (SCC)**: algoritmi come Kosaraju o Tarjan utilizzando un DFS per identificare gruppi di nodi mutualmente raggiungibili.

Bellman-Ford

Pseudocodice

```
BELLMAN-FORD(G, w, s)
    INITIALIZE-SINGLE-SOURCE(G, s)
    for i = 1 to |G.V|-1
        for each edge (u, v) ∈ G.E
            RELAX(u, v, w)
    for each edge (u, v) ∈ G.E
        if v.d > u.d + w(u, v)
            return false
    return true
```

Descrizione

È un algoritmo per trovare i cammini minimi da un nodo sorgente a tutti gli altri nodi in un grafo con pesi qualsiasi (positivi, negativi o nulli). A differenza di Dijkstra, può gestire archi con peso negativo e rilevare cicli a peso negativo raggiungibili dalla sorgente. L'algoritmo si basa sul rilassamento iterativo di tutti gli archi per V - 1 iterazioni, garantendo che le distanze siano corrette se non ci sono cicli negativi.

Tempo di esecuzione:

$T(n) = O(|V| \cdot |E|)$

Dijkstra

Pseudocodice:

```
DIJKSTRA(G, w, s)
    INITIALIZE-SINGLE-SOURCE(G, s)
    S = ∅
    Q = G.V
    while Q ≠ ∅
        u = EXTRACT-MIN(Q)
        S = S U {u}
        for ogni vertice v ∈ G.Adj[u]
            RELAX(u, v, w)
```

Descrizione:

Risolve il problema dei cammini minimi da sorgente unica in un grafo pesato G = (V, E) nel caso in cui tutti i pesi degli archi non siano negativi. L'algoritmo mantiene un insieme S di vertici i cui pesi finali dei cammini minimi dalla sorgente s sono stati già determinati. L'algoritmo seleziona ripetutamente il vertice u ∈ V-S con la stima minima del cammino minimo, aggiunge u ad S e rilassa tutti gli archi che escono da u.

Tempo di esecuzione:

- **con heap binario**: $T(n) = O((|V| + |E|) \cdot log|V|)$
- **con un heap di Fibonacci**: $T(n) = O(|E| + |V|log|V|)$

Kruskal

Pseudocodice:

```
MST-KRUSKAL(G, w)
    A = ∅
    for ogni vertice v ∈ G.V
        MAKE-SET(v)
    ordina gli archi di G.E in senso non decrescente rispetto al peso w
    for ogni arco (u, v) ∈ G.E preso in ordine di peso non decrescente
        if FIND-SET(u) ≠ FIND-SET(v)
            A = A ∪ {(u, v)}
            UNION(u, v)
    return A
```

Descrizione:

È un metodo greedy per trovare un Minimum Spanning Tree (MST) in un grafo non orientato e pesato. L'MST è un sottoinsieme di archi che connette tutti i nodi del grafo con il peso totale minimo, senza formare cicli.

1. **ordina tutti gli archi** del grafo in ordine crescente di peso
2. **aggiunge iterativamente** gli archi all'MST, partendo da quello con peso minore
3. **evita i cicli** utilizzando una struttura dati UNION-FIND (o insiemi disgiunti) per verificare se due nodi appartengono già alla stessa componente connessa
   L'algoritmo termina quando sono stati selezionati V-1 archi (dove V è il numero di nodi), poichè un albero ha esattamente V - 1 archi.

Tempo di esecuzione:

$T(n) = O(|E| \cdot log|V|)$

Prim

Pseudocodice:

```
MST - Prim(G, w, r)
    for each u ∈ G.V
        u.key = ∞
        u.π = NIL
    r.key = 0
    Q = G.V
    while Q ≠ ∅
        u = EXTRACT-MIN(Q)
        for each v ∈ G.Adj[u]
            if v ∈ Q and w(u, v) < v.key
                v.π = u
                v.key = w(u, v)
```

Descrizione:

È un metodo greedy per trovare un Minimum Spanning Tree (MST) in un grafo non orientato e pesato. A differenza di Kruskal, che lavora sugli archi, Prim costruisce l'MST incrementando un albero a partire da un nodo iniziale, aggiungendo iterativamente l'arco di peso minimo che collega l'albero corrente a un nodo non ancora incluso.

1. **seleziona un nodo iniziale** come radice dell'MST
2. **mantiene una struttura a coda a priorità** per tracciare i nodi non ancora inclusi nell'MST, con priorità pari al peso minimo degli archi che li collegano all'albero corrente
3. **aggiunge iterativamente il nodo con il peso minimo** all'MST, aggiornando i pesi dei suoi vicini
4. **ripete** finchè tutti i nodi non sono inclusi nell'MST

Tempo di esecuzione:

- **con heap binario**: $T(n) = O(|E|log|V|)$
- **con heap di Fibonacci**: $T(n) = O(|E| + |V|log|V|)$