Grafi

Grafi bipartiti

- Un grafo non orientato G è bipartito se l'insieme dei nodi può essere partizionato in due sottoinsiemi disgiunti tali che nessun arco del grafo connette due nodi appartenenti allo stesso sottoinsieme.
- G è 2-colorabile se ogni nodo può essere colorato di bianco o di nero in modo che nodi connessi da archi siano colorati con colori distinti.
- Si dimostri che G è bipartito: (1) se e solo se è 2-colorabile; (2) se e solo se non contiene circuiti di lunghezza dispari.
- Scrivere un algoritmo che colori i nodi di un grafo bipartito di due colori, per comodità identificati da 0 e 1. I colori devono essere assegnati in modo che due nodi con lo stesso colore non abbiano archi in comune. Discuterne la complessità.

Stessa distanza

Si consideri il seguente problema: dato un grafo orientato G e due nodi s_1 e s_2 di G, trovare il numero di nodi di G raggiungibili sia da s_1 che da s_2, e che si trovano alla stessa distanza da s_1 e da s_2. Scrivere un algoritmo che risolva il problema e discuterne la complessità.
Ricordiamo che in un grafo orientato G, dati due nodi s e v, si dice che v è raggiungibile da s se esiste un cammino da s a v e, in questo caso, la distanza di v da s è la lunghezza del più breve cammino da s a v (misurato in numero di archi).

Distanza fra partizioni

Dato un grafo G e due sottoinsiemi V_1 e V_2 dei suoi vertici si definisce distanza tra V_1 e V_2 la distanza minima per andare da un nodo in V_1 ad un nodo in V_2. Nel caso V_1 e V_2 non siano disgiunti allora il valore è 0.
Descrivere un algoritmo mindist(Graph G, Set V_1, Set V_2) che restituisce la distanza minima (in numero di archi). Discutere complessità e correttezza, assumendo che l'implementazione degli insiemi sia tale che il costo di verificare l'appartenenza di un elemento all'insieme abbia costo $O(1)$.
Nota: è facile descrivere un algoritmo $O(nm)$; esistono tuttavia algoritmi di complessità $O(n^2)$ (con matrice di adiacenza) e $O(m+n)$. La valutazione dipenderà dall'efficienza dell'algoritmo trovato.

Connetti il grafo

- Progettare un algoritmo efficiente che dato un grafo non orientato, restituisca il numero minimo di archi da aggiungere per renderlo connesso.
- Progettare un algoritmo efficiente che dato un grafo non orientato, aggiunga il numero minimo di archi necessari a renderlo connesso.

Spoiler alert!

Grafi bipartiti

- Se G è bipartito, è 2-colorabile. Diamo colore 1 a tutti i nodi in una partizione, diamo colore 2 a tutti i nodi nell'altra. Non essendoci archi fra i nodi di una partizione, la colorazione è valida.
- Se G è 2-colorabile, non contiene cicli di lunghezza dispari. Supponiamo per assurdo che esista un ciclo $(v_1, v_2), (v_2, v_3) \ldots, (v_{k-1}, v_k), (v_k, v_1)$, con k dispari. Se il nodo v_1 ha colore 1, il nodo v_2 deve avere colore 2; il nodo v_3 deve avere colore 1, e così via fino al nodo v_k, che deve avere colore 1. Poichè v_1 è successore di v_k, v_1 deve avere colore 2, assurdo.

Grafi bipartiti

- Se non esistono cicli di lunghezza dispari, il grafo è bipartito. Dimostriamo questa affermazione costruttivamente. Si prenda un nodo x lo si assegna alla partizione S_1. Si prendono poi tutti i nodi adiacenti a nodi in S_1 e li si assegna alla partizione S_2. Si prendono tutti i nodi adiacenti a nodi in S_2 e li si assegna alla partizione S_1. Questo processo termina quando tutti i nodi appartengono ad una o all'altra partizione. Un nodo può essere assegnato più di una volta se e solo se fa parte di un ciclo. Ma affinché venga assegnato a due colori diversi, deve far parte di un ciclo di lunghezza dispari, e questo non è possibile.

Grafi bipartiti

Questa funzione ritorna un vettore di colori se il grafo è 2-colorabile, **nil** altrimenti.

---

integer[ ] color(GRAPH G)

```
integer[ ] colors ← new integer[1 ... G.n]
for u ∈ G.V() do
    colors[u] ← −1
foreach u ∈ G.V() do
    if colors[u] < 0 then
        if not colorRec(G, u, colors, 0) then
            return nil
return colors
```

Grafi bipartiti

Questa funzione ritorna **true** se il grafo è 2-colorabile, **false** altrimenti.

---

boolean colorRec(GRAPH G, NODE u, **integer** $color$)

```
colors[u] ← color
foreach v ∈ G.adj(u) do
    if colors[v] < 0 then
        colorRec(G, u, colors, 1 − color)
    else if colors[v] = color then
        return false
return true
```

Stessa distanza

integer sameDistance(GRAPH G, NODE s_1, NODE s_2)

integer $dist_1 \leftarrow$ **new integer**$[1 \ldots G.n]$
integer $dist_2 \leftarrow$ **new integer**$[1 \ldots G.n]$
erdos$(G, s_1, dist_1)$
erdos$(G, s_2, dist_2)$
integer $counter = 0$
foreach $u \in G.\mathsf{V}()$ **do**
    if $dist_1[u] = dist_2[u]$ **then**
        $counter \leftarrow counter + 1$
return $counter$

Distanza fra partizioni

mindist(Graph G, Set V_1, Set V_2)

```
Queue Q ← Queue()
integer[] dist ← new integer[1 ... G.n]
foreach u ∈ G.V() do
    if V_1.contains(u) then
        Q.enqueue(u)
        dist[u] ← 0
        if V_2.contains(u) then
            return 0
    else
        dist[u] ← ∞
```

Distanza fra partizioni

```
while not Q.isEmpty() do
    NODE u ← Q.dequeue()
    foreach v ∈ G.adj(u) do
        if dist[v] = ∞ then
            dist[v] ← dist[u] + 1
            if V2.contains(v) then
                return dist[v]
            Q.enqueue(v)
return +∞
```

Connetti il grafo - 1

```
integer numberToConnect(Graph G)
integer[] id ← new integer[1...G.n]
foreach u ∈ G.V() do id[u] ← 0

integer counter ← 0
foreach u ∈ G.V() do
    if id[u] = 0 then
        counter ← counter + 1
        ccdfs(G, counter, u, id)
return counter − 1

ccdfs(Graph G, integer counter, Node u, integer[] id)
    id[u] ← counter
    foreach v ∈ G.adj(u) do
        if id[v] = 0 then
            ccdfs(G, counter, v, id)
```

Connetti il grafo - 2

numberToConnect(GRAPH G)

```
integer[ ] id ← new integer[1 ... G.n]
integer[ ] rappresentante ← new integer[1 ... G.n]
foreach u ∈ G.V() do id[u] ← 0
integer counter ← 0
foreach u ∈ G.V() do
    if id[u] = 0 then
        counter ← counter + 1
        rappresentante[counter] ← u
        ccdfs(G, counter, u, id)
for i ← 2 to counter do
    G.insertEdge(rappresentante[i − 1], rappresentante[i])
```

ccdfs(GRAPH G, **integer** $counter$, NODE u, **integer**[ ] id)

```
id[u] ← counter
foreach v ∈ G.adj(u) do
    if id[v] = 0 then
        ccdfs(G, counter, v, id)
```