Grafi – Stessa distanza

- 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;
  - la distanza di v da s è la lunghezza del più breve cammino da s a v (misurato in numero di archi), oppure $+\infty$ se v non è raggiungibile da s
- Scrivere un algoritmo che prenda in input un grafo orientato $G = (V, E)$ e due nodi $s_1, s_2 \in V$ e che restituisca il numero di nodi in V tali che:
  - siano raggiungibili sia da s_1 che da s_2, e
  - si trovino alla stessa distanza da s_1 e da s_2.
- Discutere la complessità dell'algoritmo proposto.

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.
- $G = (V, E)$ è 2-colorabile se è possibile trovare una 2-colorazione di esso, ovvero un assegnamento $c[u] \in C$ per ogni nodo $u \in V$, dove C è un insieme di "colori" di dimensione 2, tale che:

$$(u, v) \in E \Rightarrow c(u) \neq c(v)$$

- Si dimostri che G è bipartito:
  - se e solo se è 2-colorabile
  - se e solo se non contiene circuiti di lunghezza dispari
- Scrivere un algoritmo che prenda in input un grafo bipartito G e restituisca una 2-colorazione di G sull'insieme di colori $C = \{0, 1\}$, espressa come un vettore $c[1 \ldots n]$. Discuterne la complessità.

Grafi – 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, misurata in numero di archi.
- Nel caso V_1 e V_2 non siano disgiunti, allora la distanza è 0.
- Scrivere un algoritmo mindist(Graph G, Set V_1, Set V_2) che restituisce la distanza minima fra V_1 e V_2.
- 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 scrivere un algoritmo $O(nm)$; esistono tuttavia algoritmi di complessità $O(n^2)$ (con matrice di adiacenza) e $O(m+n)$.

Grafi – 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!

Stessa distanza

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

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

Grafi – Grafi bipartiti

- Se G è bipartito, è 2-colorabile. Diamo colore 0 a tutti i nodi in una partizione, diamo colore 1 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 – 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 – Grafi bipartiti

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

---

int[ ] color(GRAPH G)

```
int[] colors = new int[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 – Grafi bipartiti

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

---

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

```
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
```

Distanza fra partizioni

mindist(Graph G, Set V_1, Set V_2)

```
Queue Q = Queue()
int[] dist = new int[1...G.n]
foreach u ∈ G.V() do
    if V1.contains(u) then
        Q.enqueue(u)
        dist[u] = 0
        if V2.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 +∞
```

Grafi – Connetti il grafo - 1

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

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

Grafi – Connetti il grafo - 2

numberToConnect(GRAPH G)

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

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