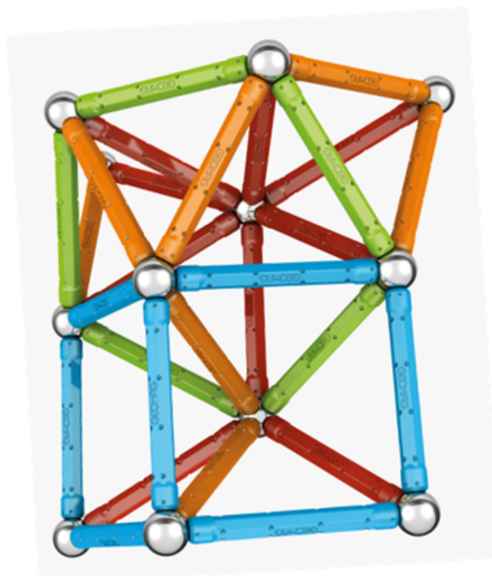
Grafi

Cosa sono

I grafi sono delle strutture dati date da un insieme di nodi V connessi da archi raggruppati in un insieme Un grafo si definisce come:

$$G = (V, E)$$

Essi possono essere rappresentati dal gioco dei bastoncini magnetici.



L'insieme degli archi *E* è definito dalla funzione che associa coppie di vertici ad un numero reale: *V*V -> R*

Tipi di grafi

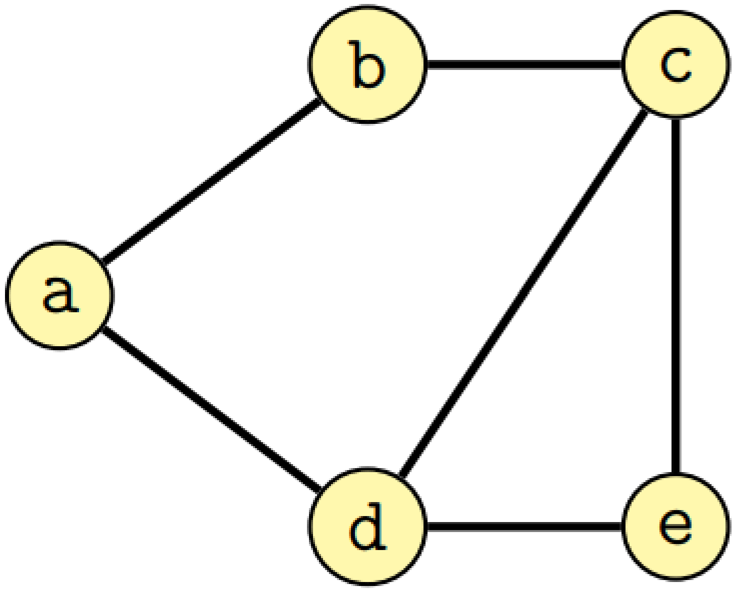
Dividiamo i grafi in base alla proprietà di essere (o meno) orientati:

- grafo orientato: gli archi hanno una direzione;
- grafo non orientato: gli archi **non** hanno una direzione.

E in base alla relazione tra il numero di archi e il numero di vertici.

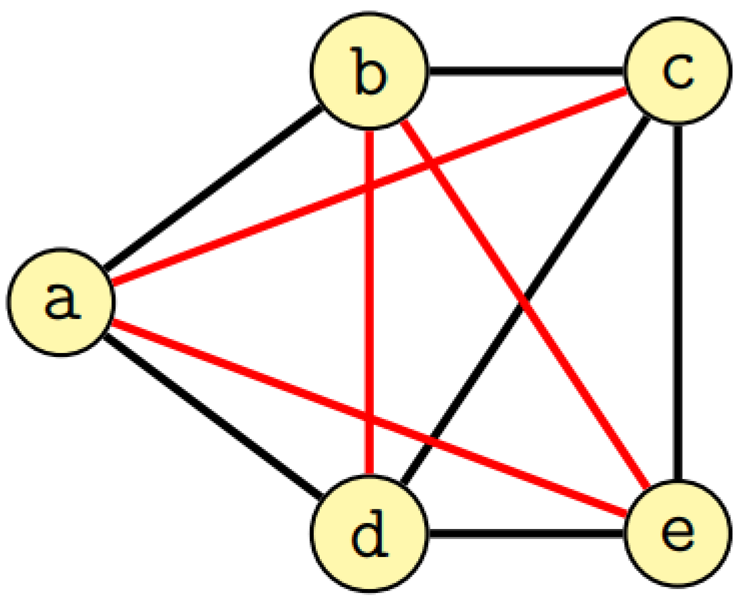
- grafo **sparso**: grafo con il numero di archi **MOLTO MINORE** rispetto al quadrato dei vertici (quindi *gli archi collegano solo **alcuni** nodi*).

$$|E| << |V^2|$$



- grafo **denso**: grafo con il numero di archi **MOLTO ELEVATO**, circa pari al quadrato dei vertici (*quindi gli archi collegano molti vertici*).

$$|E| \approx |V^2|$$



Adiacenza dei nodi

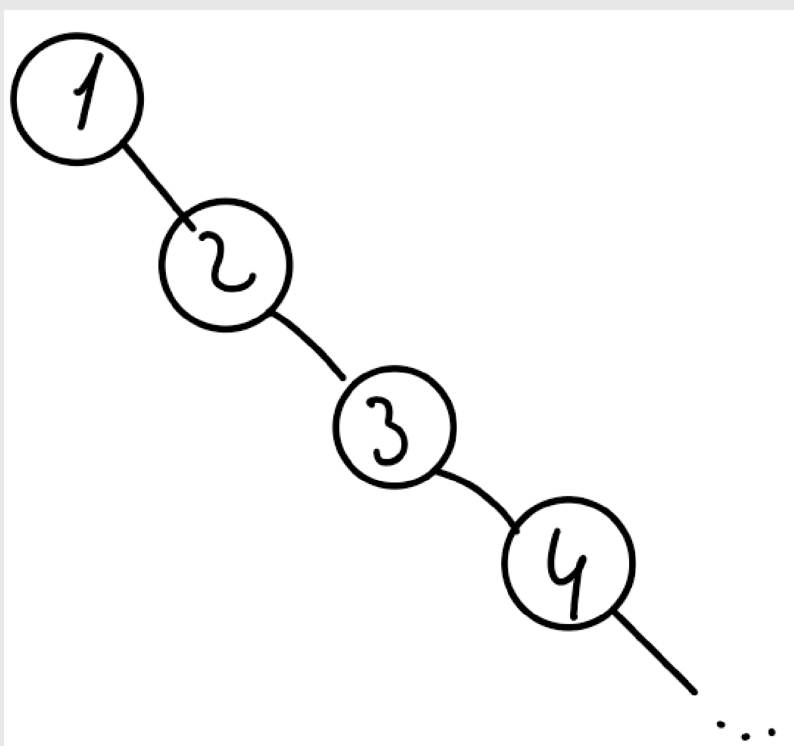
Un nodo si dice **adiacente** se esise un arco

$$e_{ij} = (v_i, v_j)$$

che li collega.

La presenza (o meno) di un orientamento è **fondamentale** per controllare **l'adiacenza dei nodi**.

- se l'arco **non è orientato** allora i due archi sono *vicendevolmente* adiacenti;
- se l'arco **è orientato** allo
- ra, per *A->B*, A è *adiacente* a B, ma *non viceversa*.

> una lista può essere vista come un grafo orientato, in quanto se inseriti in un BST i valori: `1 - 2 - 3 - 4 - 5 - 6 - 7`... si otterrà:
>
> > 
> >
> >
> > Anche l'albero può infatti essere visto come un grafo

> > orientato in quanto **ha un verso**.

Come rappresentare un grafo

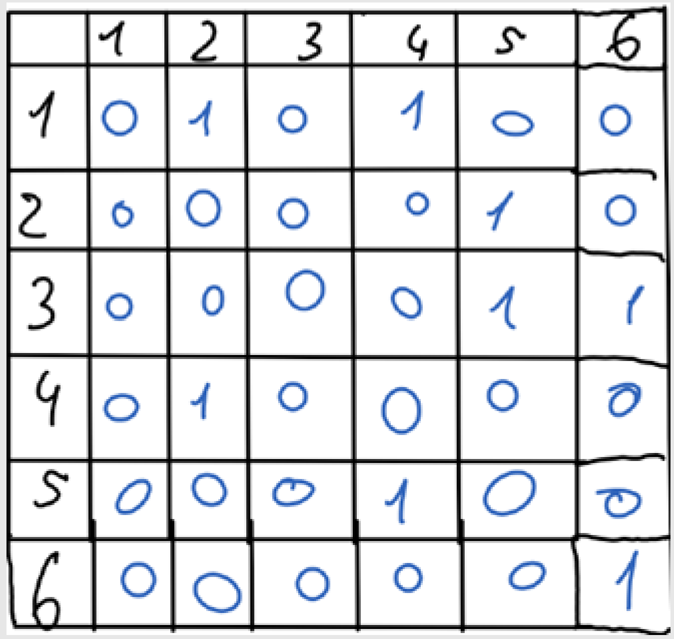
Un grafo può essere rappresentato come:

- una *lista di adiacenza* contenente una lista (*la struttura dati più generale*) di **vertici adiacenti** al vertice preso in considerazione; può essere infinita
- una *matrice di adiacenza* contenente 0 e 1 a seconda della adiacenza (1) o non adiacenza (0) di due nodi; quindi è rappresentato da *una matrice* nella quale è segnalata **l'adiacenza** posizionando l'`1` nell'elemento `[i, j]`.

In caso di grafo denso è conveniente utilizzare una matrice di adiacenza; mentre in caso di grafo sparso è conveniente utilizzare una lista di adiacenza. La scelta dell'utilizzo di una piuttosto che dell'altra ricade infatti sulla memoria occupata: per la matrice l'allocazione di memoria è fissa, quindi è sconveniente utilizzarla per un grafo con pochi archi.

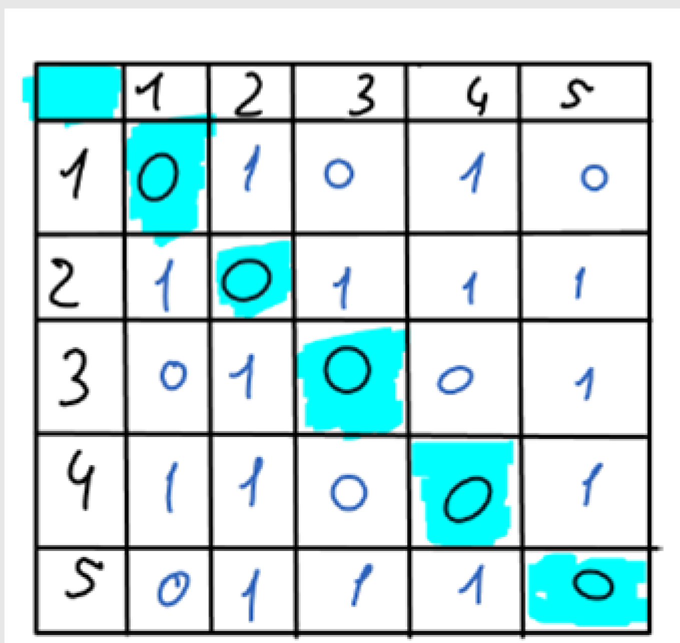
Inoltre la scelta ricade anche sulla dimensione del grafo: se esso è di dimensioni ridotte è preferibile utilizzare la *matrice di adiacenza*. D'altronde, la *lista di adiacenza* richiede più tempo per l'accesso ad un elmento perché occorre scorrere **tutta** la lista.

> sia che si parli di lista di adiacenza e sia che si parli di matrice di adiacenza, in ogni caso se il **grafo non è orientato** la struttura dati non è **simmetrica**.



	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Si nota facilmente nella matrice perché non posso ridurla considerando solo la diagonale superiore, a differenza di quando ho un grafo non orientato.



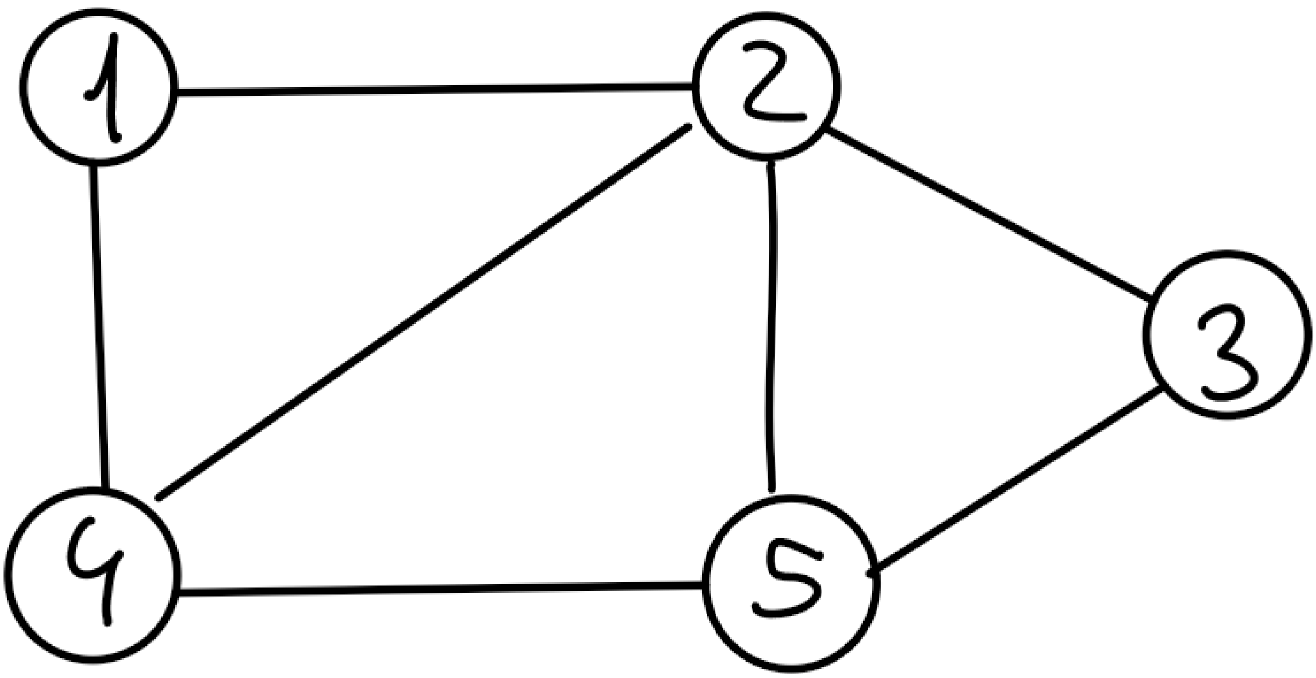
	1	2	3	4	5
1	0	1	0	1	0
2	1	0	1	1	1
3	0	1	0	0	1
4	1	1	0	0	1
5	0	1	1	1	0

Lista di adiacenza

Elementi indispensabili per la lista

Gli elementi indispensabili per la lista di adiacenza sono:

- il **vertice** del grafo (`GraphVertex<T>`) che non avrà nessun attributo;
- i **vertici adiacenti** che avranno gli attributi:
  - `List GraphVertex<T> vertices`;
  - `bool isOriented`;
  - `int nVertices`;



> in questo esempio gli adiacenti sono:
>
> > Per 1 = 2, 4
> > Per 2 = 1, 3, 4, 5
> > Per 3 = 2, 5
> > Per 4 = 1, 2, 5
> > Per 5 = 2, 3, 4

A seconda che i nodi siano finiti (o meno) posso scegliere di implementare:

- un **array di liste** (*per i nodi finiti*) dove ad ogni posizione dell'array corrisponde la lista di vertici adiacenti
- una **lista** (*per i nodi non finiti*) dove ad ogni nodo della lista corrisponde la lista di vertici adiacenti

Complessità

Lo spazio richiesto in memoria è di

$$\Theta(|V| + |E|)$$

Se il grafo è orientato si considerano **2|E|** mentre se non è orientato si considera **|E|**.

Procedimento

Per implementare la *lista di adiacenza* creiamo una classe **vertex** (vertice) ereditando dalla classe `list` in maniera **public**.
Successivamente istanziamo una seconda classe (`template`) che avrà come attributi *private* proprio la classe **vertex**.

GraphVertex

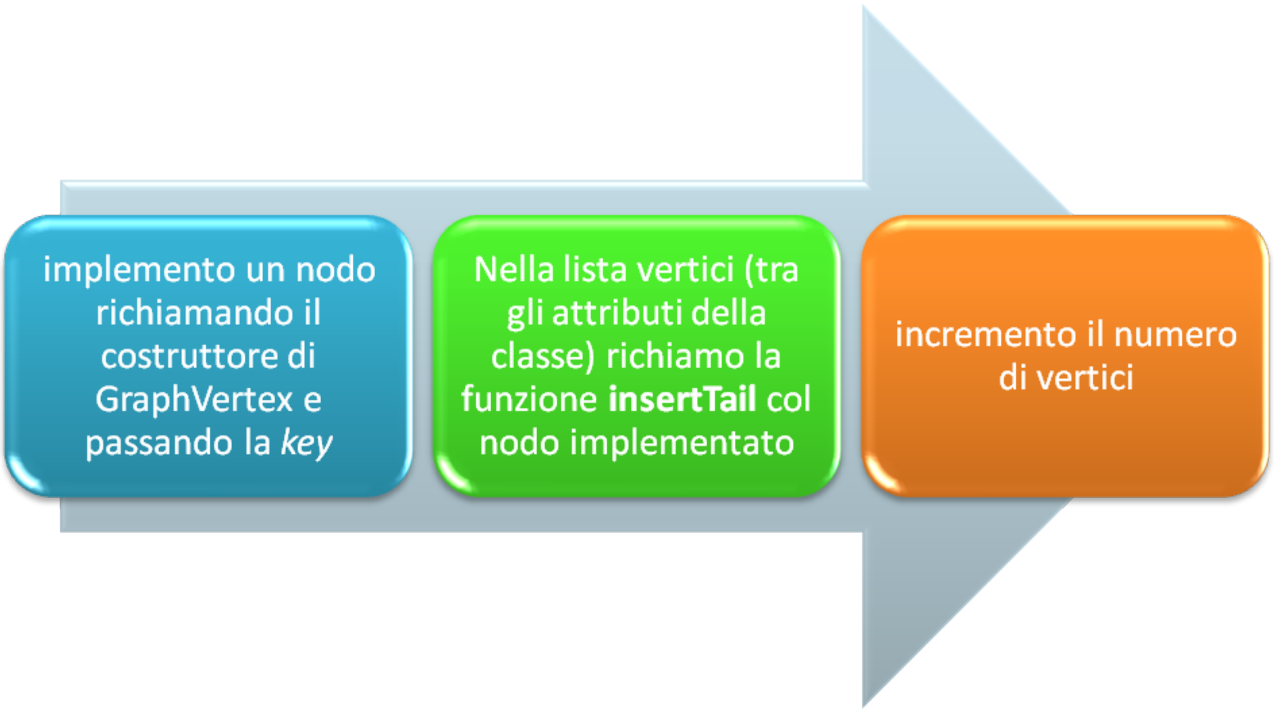
GraphVertex è una classe `template` che eredita in maniera *public* dalla classe `list` (già esistente). Tale classe ha come metodi *public* il costruttore e l'overload di stampa.

GraphList

GraphList è una classe `template` che ha come attributi *private* la lista di vertici (`GraphVertex`), il numero di vertici e un booleano indicante se il grafo è orientato o meno. Tale classe ha come metodi *public* il costruttore (che *inizializza* l'orientamento), l'aggiunzione di un nodo, l'aggiunzione di un arco, la ricerca di un nodo e l'overload di stampa.

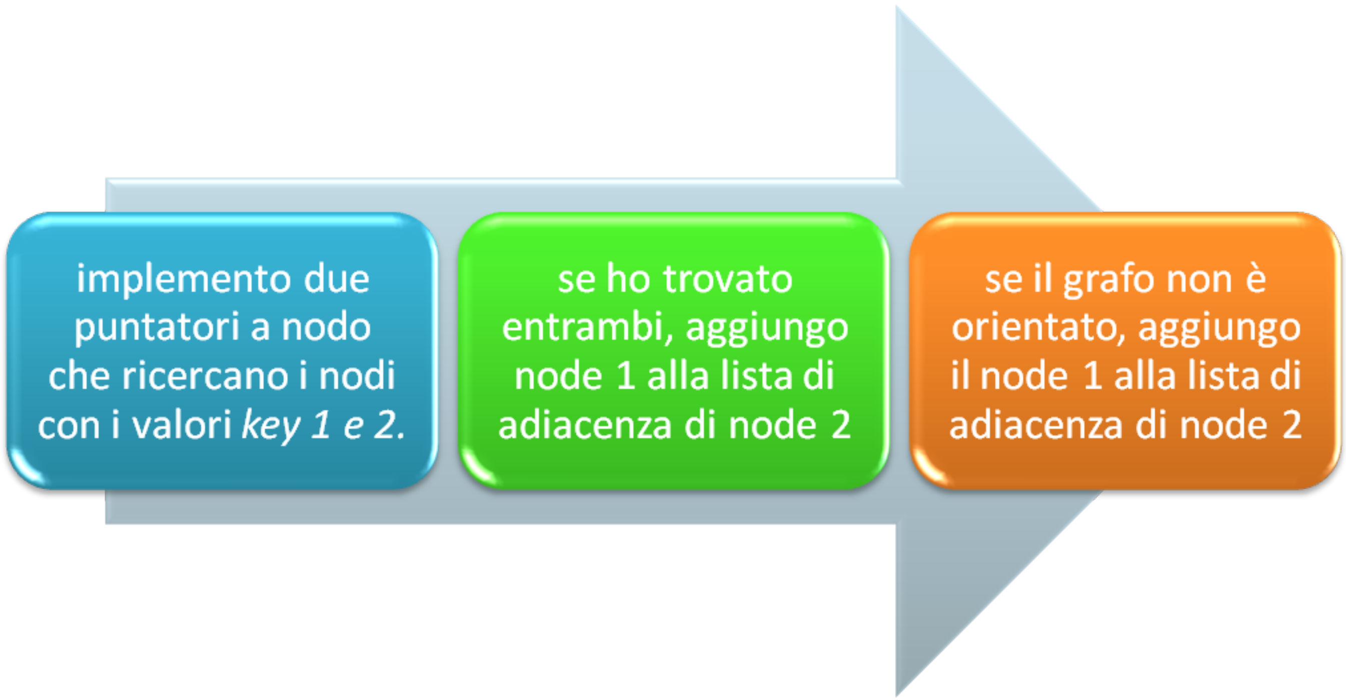
addVertex

addVertex è il metodo *public* della classe `GraphList` dedito all'aggiunzione di un nodo. Non ritorna nulla ed è passato un parametro *key*.



addEdge

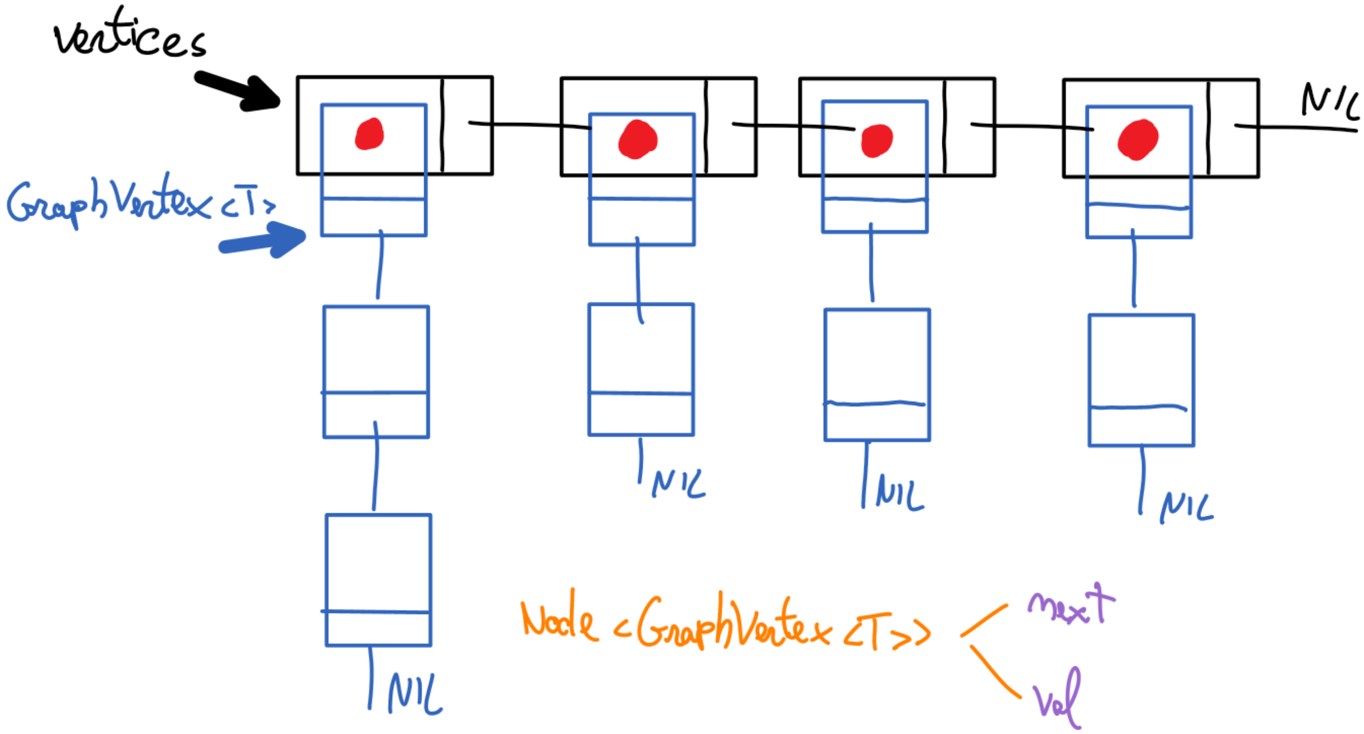
addEdge è il metodo *public* della classe `GraphList` dedito all'aggiunzione di un arco. Non ritorna nulla e sono passati due parametri (template) key1 e key2.

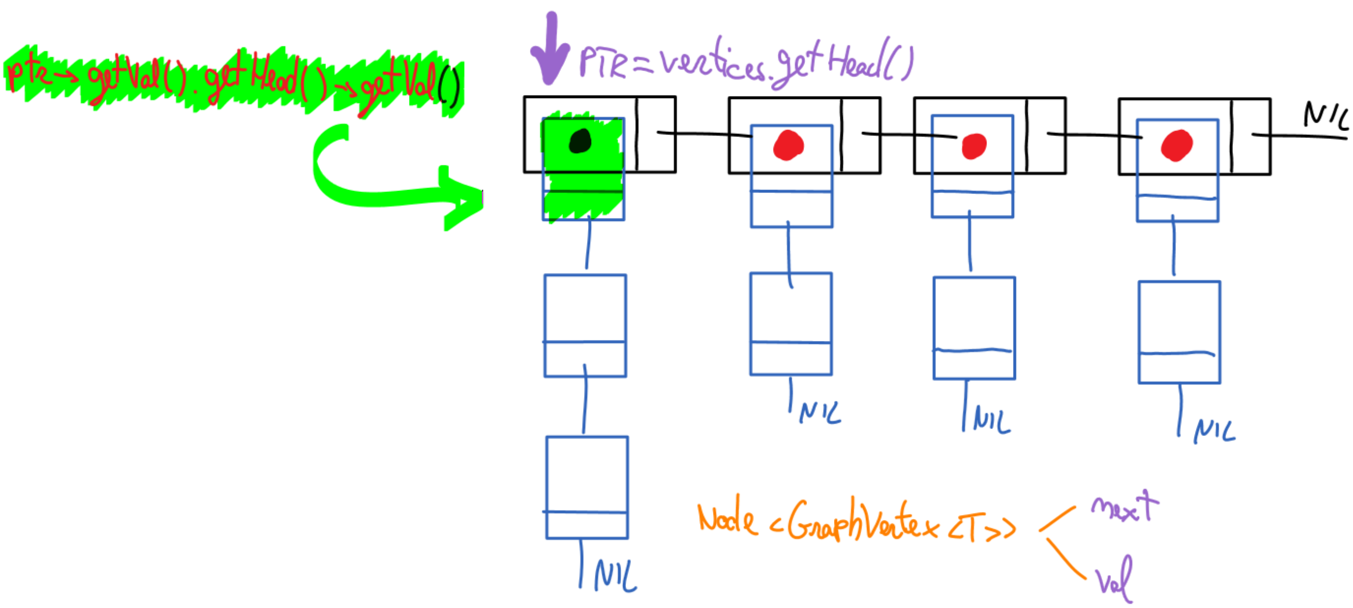


Search

Search è il metodo *public* della classe `GraphList` dedito alla ricerca di un nodo. Ritorna il puntatore a nodo ed è passato il parametro *key*.

Ho una lista di una lista. Quando si cerca un valore, ci interessa la head all'interno della lista contenuta nel nodo.
Per comprendere meglio:





Codice

Per prima cosa controlliamo se `GRAPH_LIST_H` è già definita, altrimenti la definiamo. Incluaidamo le librerie necessarie e l'header "*list.h*"

```C++
#ifndef GRAPH_LIST_H
#define GRAPH_LIST_H
```

```
#include<iostream>
#include "list.h"
using namespace std;
```

Successivamente istanziamo una classe **template** `GraphVertex` che eredita in maniera *public* da `List`.

```C++
template <typename T>
class GraphVertex : public List<T>{
```

ed implementiamo come metodi public:

1. il costruttore (che richiama la funzione `insertTail` dalla classe madre) a cui è passato il parametro *key*.

```C++
GraphVertex(T key) : List<T>(){
    List<T> :: insertTail(key);
}
```

2. l'overload di ostream con cui *rendo chiaro qual è il vertice e quali sono i suoi nodi adiacenti* stampando prima la testa della lista e successivamente inizializzando un puntatore al successivo della testa finché il puntatore non corrisponde a `nullptr` stampo i valori a seguire (richiamando dalla classe madre la funzione `getNext()`)

```C++
    friend ostream& operator<< (ostream& out,
GraphVertex<T>& v){
        out << "Graph Vertex with key " << v.getHead()-
>getVal();
```

```
        out << ": " << "\tAdjecency List: ";
        Node <T>* ptr = v.getHead->getNext();
        while(ptr){
            out << "->" << ptr->getVal();
            ptr = ptr->getNext();
        }
        return out;
    }
};
```

Dopodiché istanziamom la classe **template** `GraphList` a cui passiamo come attributi *private* la lista di vertici (`GraphVertex`), un intero contatore dei vertici e un valore booleano indicante se il grafo è orientato o meno.

```C++
    template <typename T>
    class GraphList{
        List <GraphVertex<T>> vertices;
        int nvertices;
        bool isOriented;
```

ed implementiamo i metodi *public*:

1. costruttore (che inizializza `isOriented` a true per poi inizializzarlo al valore passato)

```C++
    public:
    GraphList(bool isOriented = true) :
isOriented(isOriented){};
```

2. `addVertex` a cui passo un valore template key, all'interno inizializzo un vertice a cui passo il valore key, dopodiché richiamo la funzione `insertTail` sulla lista vertici passando come parametro il nodo appena inizializzato e incremento infine il numero di vertici

```C++
void addVertex(T key){
    GraphVertex<T> toAdd(key);
    vertices.insertTail(toAdd);
    nvertices++;
}
```

3. `addEdges` a cui passo due valori template key 1 e key 2. Inizializzo due puntatori che richiamino la funzione di ricerca dei nodi (passando appunto come valori di ricerca i due key) e se li ho trovati entrambi inserisco il valore del nodo1 in coda (`inserTail`) al nodo 2. Controllo infine se il grafo non è orientato e in quel caso faccio il procedimento in maniera speculare.

```C++
void addEdge(T key1, T key2){
    Node<GraphVertex<T>> *node1 = this->search(key1);
    Node<GraphVertex<T>> *node2 = this->search(key2);

    if (node1 && node 2){
        node1->getVal().insertTail(key2);
        if (!this->isOriented)
            node2->getVal().insertTail(key1);
        }
    }
```

4. `search` a cui passo come parametro un valore (template) key e che restituisce un puntatore a nodo. Per prima cosa controlla se la lista è vuota e ritorna eventualmente il puntatore nullo, altrimenti inizializza un puntatore all'elemento della testa (con la funzione `getVal`) e scorre la lista (`getNext()`) finché il puntatore è **diverso da** `nullptr`. Se il valore è stato trovato ritorna il puntatore altrimenti ritorna il puntatore nullo.

```C++
    Node<GraphVertex<T>>* search (T key){
        if (this->isEmpty()){
            return NULL;
        }

        Node<GraphVertex<T>>* ptr = vertices.getHead-
>get.Val();
        while (ptr){
            if (key == ptr->getVal)
                return ptr;
            ptr = ptr->getNext();
            }
            return NULL;
        }
```

5. l'overload di ostream&

```C++
    friend ostream& operator<< (ostream& out,
GraphList<T>& g){
        out << g.vertices;
        return out;
    }
```

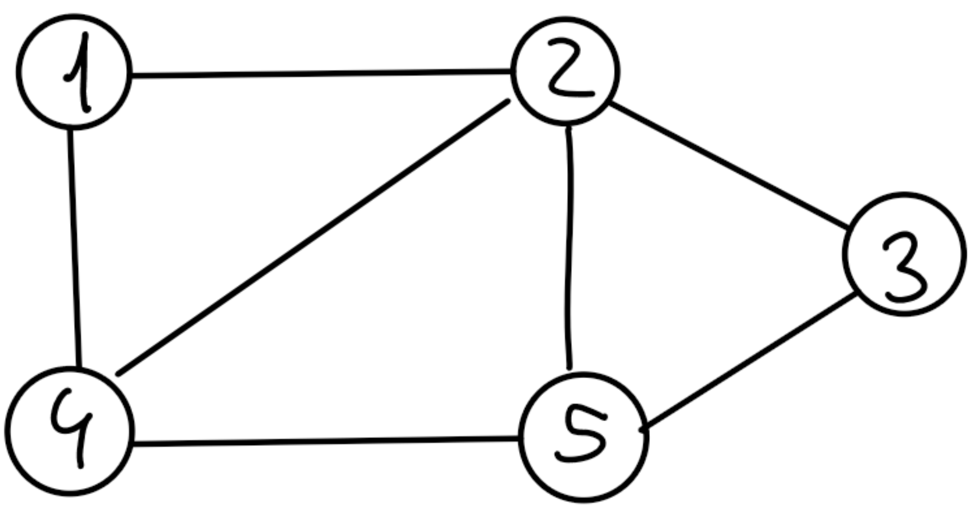
```
};
#endif
```

Matrice di adiacenza

Elementi indispensabili per la matrice

Gli elementi indispensabili per la matrice di adiacenza sono:

- `Vertex<T>` con attributo:
    - T key;
- Grafo con matrice di adiacenza `Graph<T>` che ha gli attributi:
    - dimensione massima (`maxSize`);
    - array di puntatori ai vertici (`Vertex<T>** vertices`);
    - matrice di adiacenza (`bool** adj`);
    - numero di vertici attuali (`nVertices`).



> dal grafo in esempio ottengo:

	1	2	3	4	5
1	0	1	0	1	0
2	1	0	1	1	1
3	0	1	0	0	1
4	1	1	0	0	1
5	0	1	1	1	0

Complessità

La complessità nella matrice è sempre costante.
Lo spazio richiesto in memoria è pari a

$$|V|^2$$

E la complessità è pari a

$$\Theta(|V|)^2$$

a prescindere dal valore di E.

Procedimento

Per implementare la *matrice di adiacenza* creiamo una classe **vertex** (vertice). Successivamente istanziamo una seconda classe (anch'essa `template`) che avrà come attributi *private* proprio la classe **vertex**.

Vertex

Vertex è una classe `template` che ha come attributi *private* il valore (template) key e la classe friend (*non ancora istanziata*) `Graph`. Mentre come metodi *public* ha il costruttore, una funzione di **controllo** (`operator==`) e l'overload di ostream.

Operator==

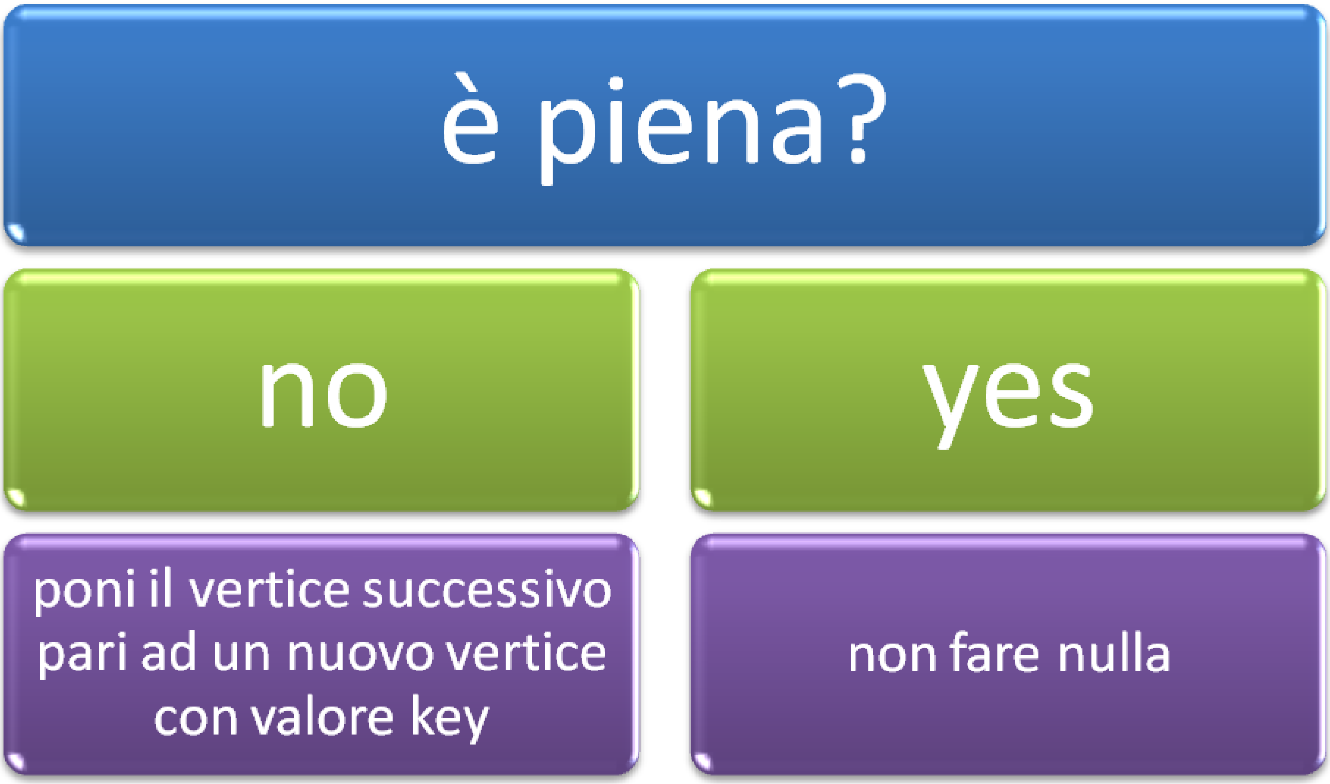
Il metodo che fa l'overload dell'operatore "`=` "serve per controllare che la chiave presa in considerazione sia pari alla key di un vertice confrontato.

Graph

Graph è una classe `template` che ha come attributi *private* una matrice di vertici (**Vertex**), una matrice di booleani (per indicare le *adiacenze*), due interi (numero elementi massimi e numero di elementi attuali). Mentre come metodi *public* ha il costruttore, `addVertex`, `addEdge`, `search` e l'overload di ostream.

AddVertex

addVertex è il metodo *public* della classe `Graph` dedito all'aggiunzione di un vertice. Non ritorna nulla ed è passato un parametro (template) *key*.



Add Edge

addEdge è il metodo *public* della classe `Graph` dedito all'aggiunzione di un arco. Non ritorna nulla e sono passati parametri (template) *key* 1 e 2.

Search

search è il metodo *public* della classe `Graph` dedito alla ricerca di un nodo. Ritorna un intero ed è passato un parametro (template) *key*.

Codice

Per prima cosa controllo se è stato definito `GRAPH_H` ed includo le librerie necessarie.

```C++
#ifndef GRAPH_H
#define GRAPH_H
#include<iostream>
using namespace std;
```

Successivamente istanzio una classe `template` che abbia come attributi *private* un valore template key e una classe template *non ancora definita* `Graph` come **friend**.

```C++
template <typename T>
class Vertex{
    private:
    T key;
    template <typename U>
    friend class Graph;
```

E implemento come metodi public:

1. il costruttore, a cui passo come parametro un valore key (e implemento key private) e implemento **il nodo** richiamando il costruttore già definito e usando il parametro `NULL`.

```C++
public:
Vertex (T key) : key(key){};
Vertex():Vertex(NULL){};
```

2. l'overload di = che ritorna il valore booleano `true` se il valore della chiave è pari al valore del vertice

```C++
bool operator=(Vertex<T>& v){
    return this->key==v.key;
}
```

3. l'overload di ostream

```C++
    friend ostream& operator<< (ostream& out, Vertex<T>&
v){
        out<<v.key;
```

```
        return out;
    }
```

4. il setter del nodo

```C++
    void setKey(T key){
        this->key = key;
    }
```

Successivamente istanzio una classe `template` che abbia come attributi *private* una matrice di `Vertex<T>` e una di booleani (indicante i nodi *adiacenti*) e due interi (uno per la dimensione massima e uno per la dimensione attuale)

```C++
    template <typename T>
    class Graph(){
        private:
        Vertex<T>** vertices;
        bool** adj;
        int maxSize = 0;
        int nVertices = 0;
```

E implemento come metodi *public*:

1. il costruttore, a cui passo come parametro la dimensione massima e che inizializza un array di `Vertex` di tale dimensione e un array di booleani - impostati tutti a 0 - di tale dimensione

```cpp
public:
Graph(int max_size): maxSize(max_size){
    vertices = Vertex<T>* [maxSize];
    adj = new bool*[maxSize];
    for(int i = 0; i<maxSize; i++)
        adj[i] = new bool[maxSize] {0};
    }
```

2. `addVertex` a cui passo come parametro un valore template key. controllo per prima cosa se l'array è completo, altrimenti inizializzo l'elemento successivo dell'array ad un nuovo `Vertex` con parametro *key*.

```cpp
void addVertex(T key){
    if (this->nVertices == this->maxSize)
        return;

    vertices[nVertices++] = new Vertex<T>(key);
}
```

3. `addEdge` a cui passo come parametri due key. Per prima cosa controllo se sono presenti queste due key utilizzando **due interi** successivamente controllo se entrambe sono diverse da -1 e in quel caso pongo pari a `true` sia il valore di posizione [i, j] che quello di posizione [j, i]. Altrimenti controllo quale delle due è diversa da -1 e a seconda indico quale delle due non è stata trovata

```cpp
void addEdge(T key1, T key2){
    int i = this->search(key1);
    int j = this->search(key2);
```

```
        if (i!=-1 && j != -1){
            adj[i][j] = 1;
            adj[j][i] = 1;
        }
        else if (i == 1)
            cerr << "There is no vertex with key " << key1
<< endl;
        else
            cerr << "There is no vertex with key " << key2
<< endl;
    }
```

4. `search` a cui passo come parametro il valore (template) da cercare e ritorna un intero. Per prima cosa controllo se il grafo non ha nodi e in quel caso ritorno -1. Altrimenti scorro tutto l'array e se trovo il nodo ritorno l'indice, altrimenti ritorno -1.

```C++
    int search(T key){
        if (this->nVertices == 0)
            return -1;

        for (int i = 0; i<this->nVertices; i++){
            if (vertice[i]->key == key)
                return i;
                }

        return -1;
    }
```

5. overload di ostream

```
C++
    friend ostream& operator (ostream& out, Graph<T>* g){
        for(int i = 0; i<g.nVertices; i++){
            out << "v[" << i "]=" << g.vertices[i] <<
"\t";
        }

        out << endl;

        for(int i = 0; i<g.nVertices; i++){
            for(int j = 0; j<g.nVertices; j++){
                if(g.adj[i][j]){
                    out << "(" << i << ", " << j << ")" <<
endl;
                }
            }
        }
        return out;
    }

 };
 #endif
```

Bisogna fare attenzione a:

1. passaggio della dimensione nel main;
2. utilizzo delle parentesi appropriate
3. numero di puntatori utilizzati
4. utilizzo dell'operatore freccia / puntatore

Visita dei grafi

I grafi possono essere visitati in diversi modi, in particolare ne andiamo a vedere 2:

1. visita dei grafi **in ampiezza**
2. visita dei grafi **in profondità**.

Visita dei grafi in ampiezza

Questo tipo di visita è definita *in ampiezza* perché procede per **distanza dal nodo sorgente**. Quindi al passo k avrò scoperto i nodi a distanza k dalla sorgente.

Alla fine si noterà che **i valori di discovery** dei vari nodi corrispondono al numero di **cammini minimi** che esistono fra *la sorgente e il nodo considerato*.

> La visita **in ampiezza** di un grafo - a differenza delle altre visite - può avvenire *indipendentemente* da qualsiasi nodo, quindi occorre stabilire per prima cosa un **nodo sorgente**, ovvero un nodo di ***partenza***.

Procedimento

Significato dei colori

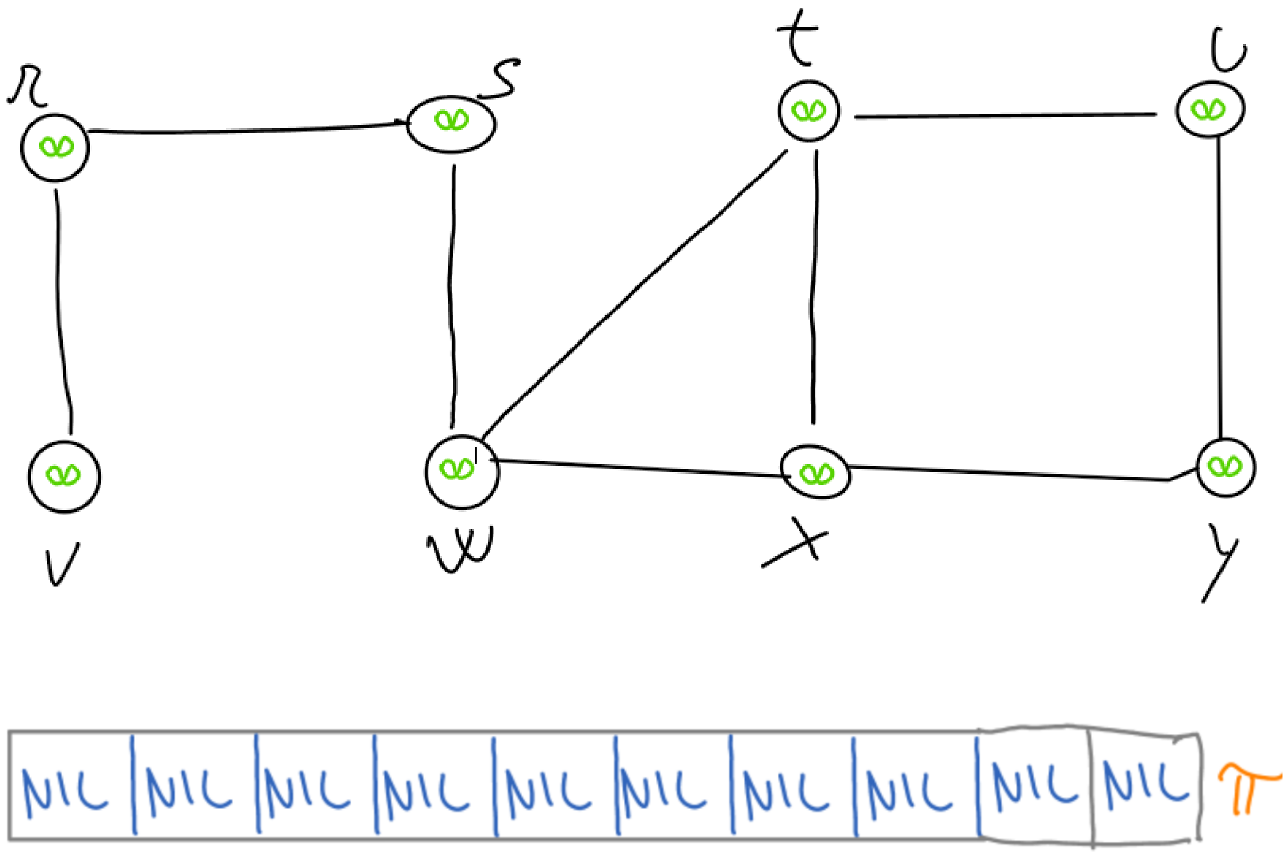
Prima di spiegare il procedimento è necessario comprendere cosa si intenderà con questi tre colori:

1. **bianco**: è il colore dei nodi non ancora scoperti (*all'inizio tutti i nodi sono bianchi*).
2. **grigio**, è il colore dei nodi *parzialmente* scoperti
3. **nero**, è il colore dei nodi scoperti (*alla fine tutti i nodi sono neri*).

Inizializzazione

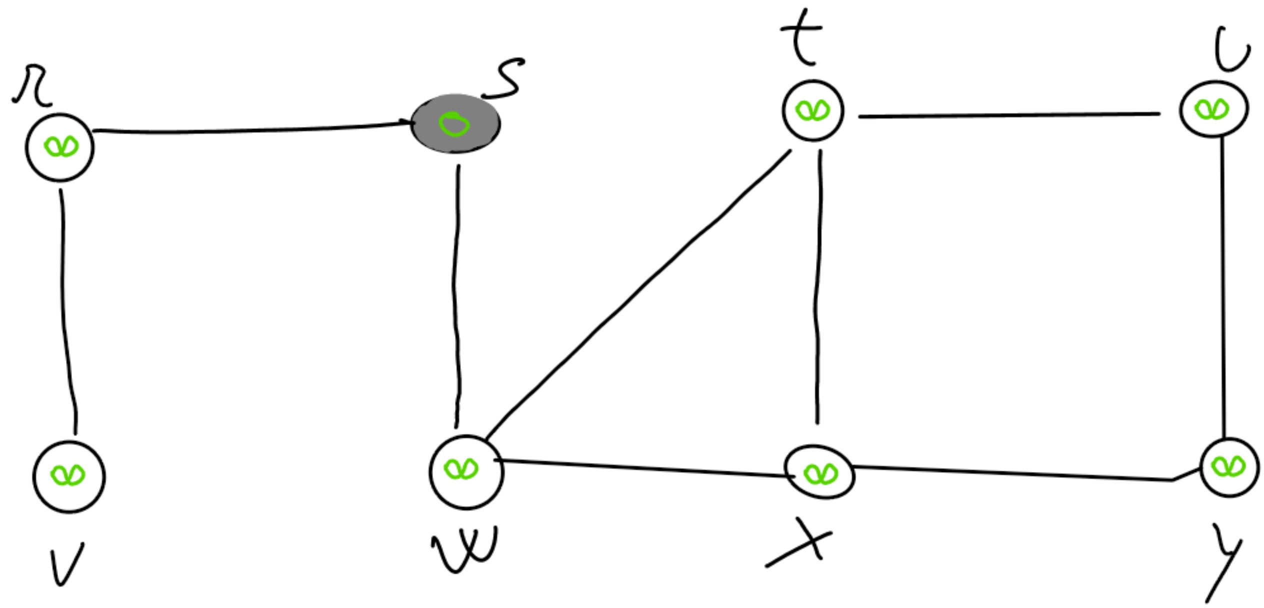
Prima di procedere con la visita si inizializzano:

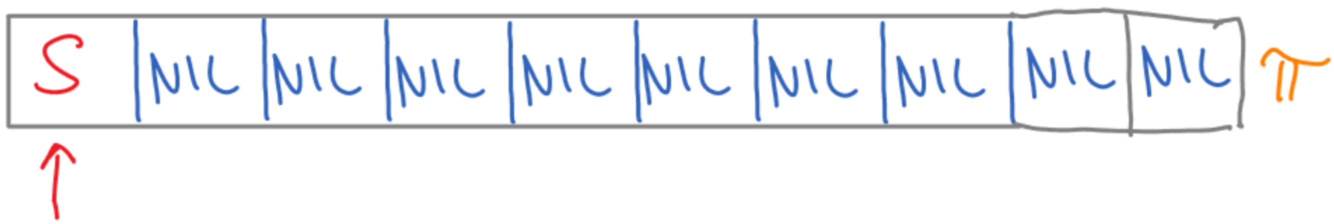
- un **array discovery** `d` (contenente *l'istante di scoperta di un nodo*) con soli valori infiniti
- una **coda dei predecessori** `π` (contente i *predecessori dei nodi considerati in quell'istante*) con valori `NIL`
- tutti i nodi del colore **bianco**



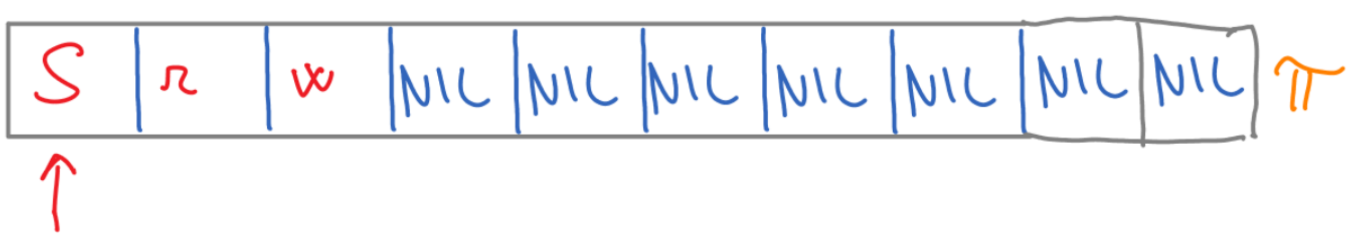
Svolgimento

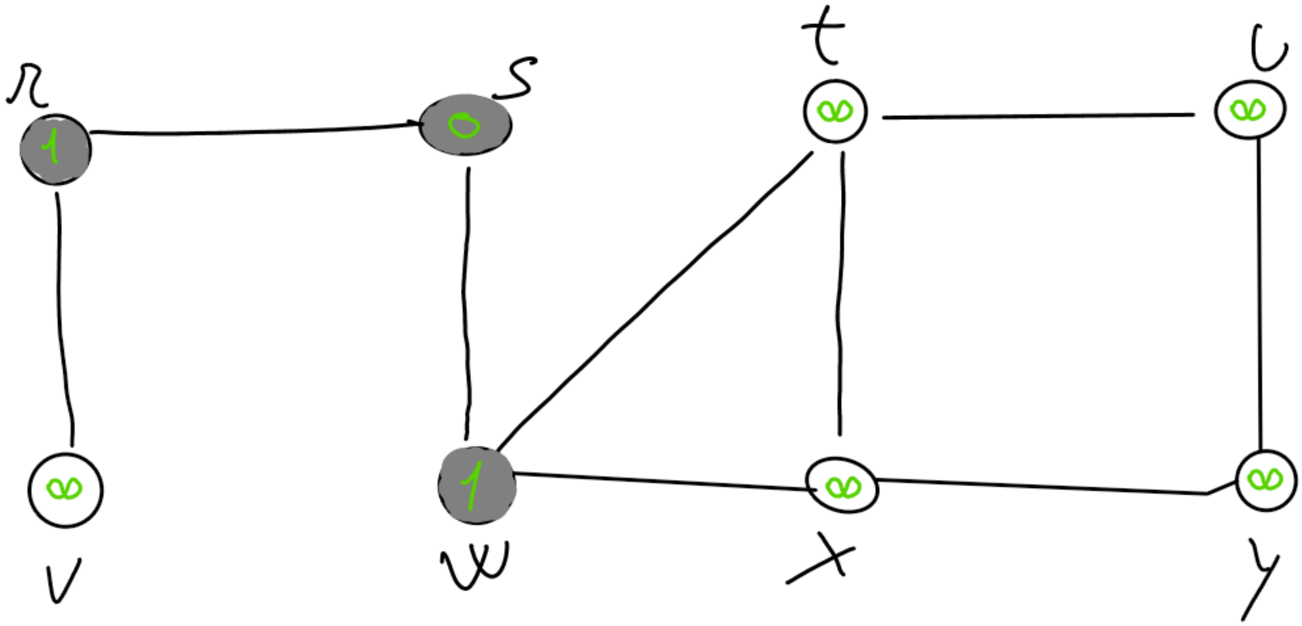
Una volta inizializzati i vari elementi, scelgo il **nodo sorgente**.
Aggiungo il nodo alla *coda dei predecessori* e lo pongo di colore grigio.
Controllo se il nodo ha *nodi adiacenti* ***bianchi***.



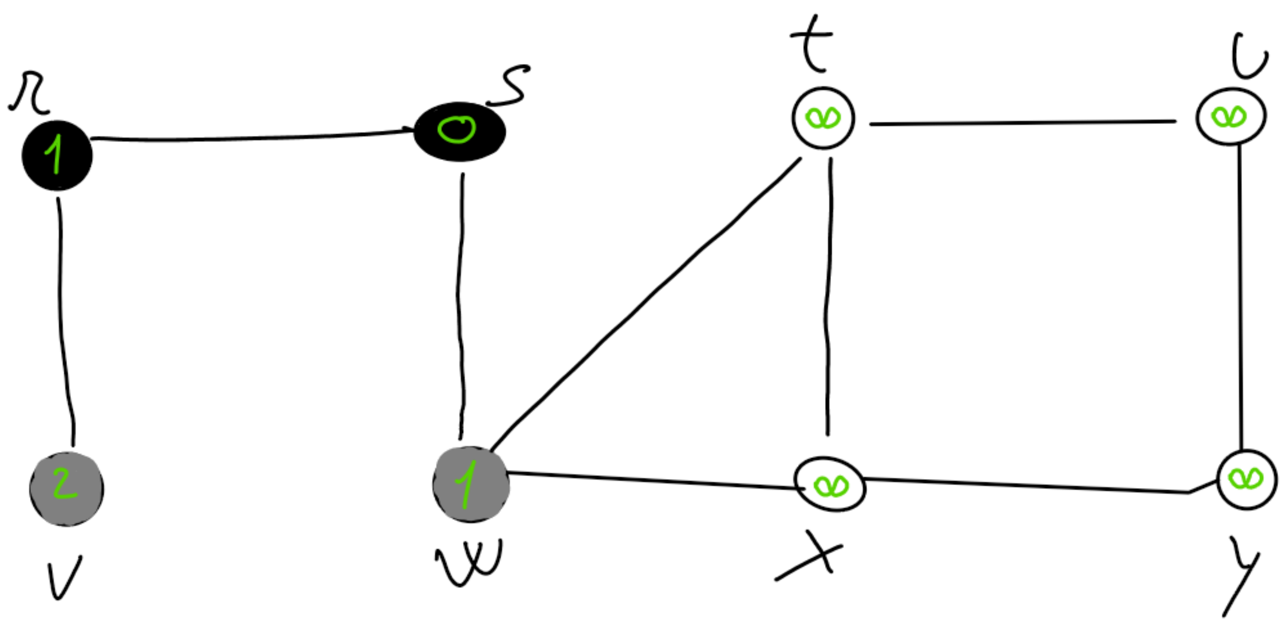


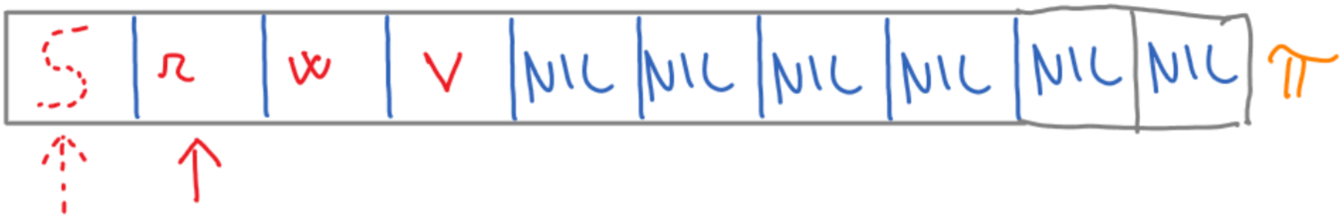
Aggiungo alla coda i nodi adiacenti **bianchi** che in questo modo diventano grigi, dato che *vengono visitati*. Il loro valore di discovery vale `predecessore+1` (quindi 1).



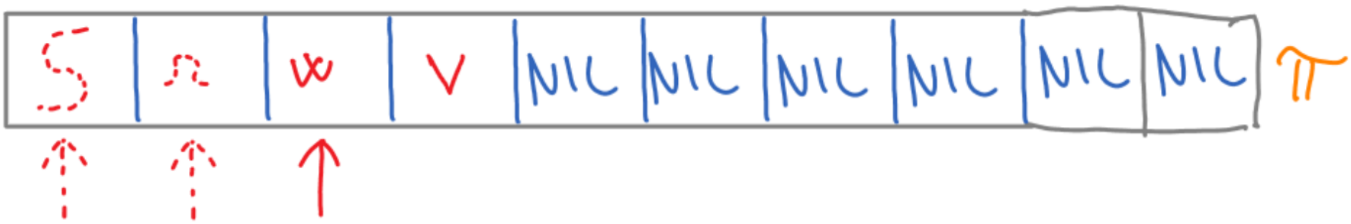


Dato che s non ha più nodi adiacenti bianchi lo "coloro" di **nero**.
Elimino dalla coda s e faccio avanzare il mio puntatore.
Il valore a seguire è r: ha nodi adiacenti bianchi?
Aggiungo v alla coda e così diventa grigio.
Dato che r non ha altri nodi adiacenti bianchi lo coloro di **nero**.

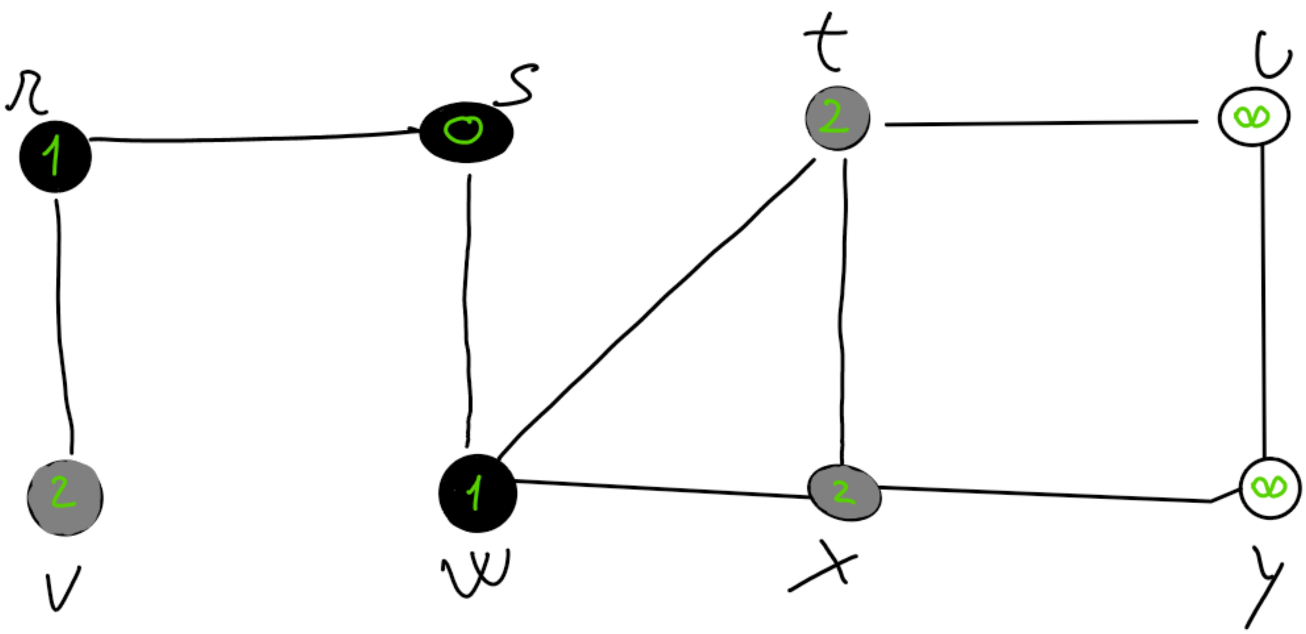


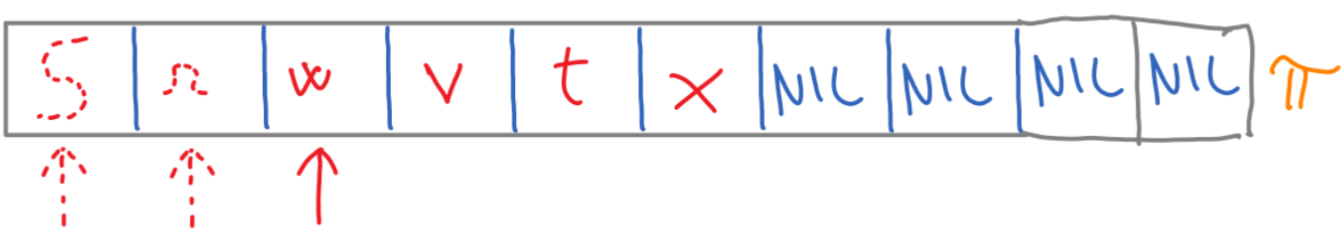


Elimino r dalla coda e faccio avanzare il mio puntatore.

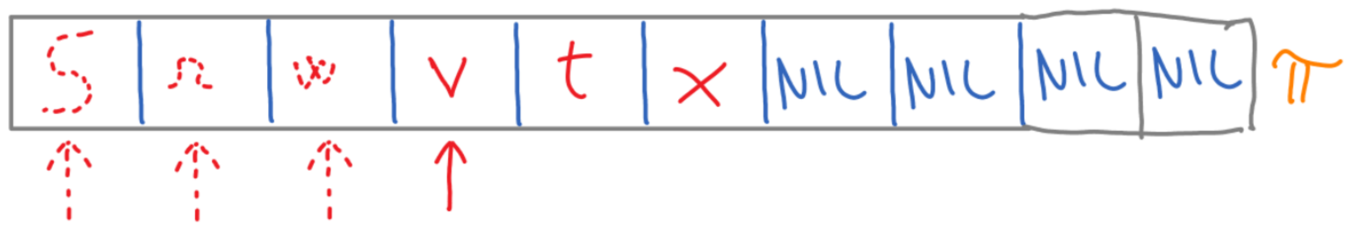


Il valore a seguire è w: ha nodi adiacenti bianchi?
Aggiungo t e x alla coda e così diventano grigi.
Dato che w non ha altri nodi adiacenti bianchi lo coloro di **nero**.

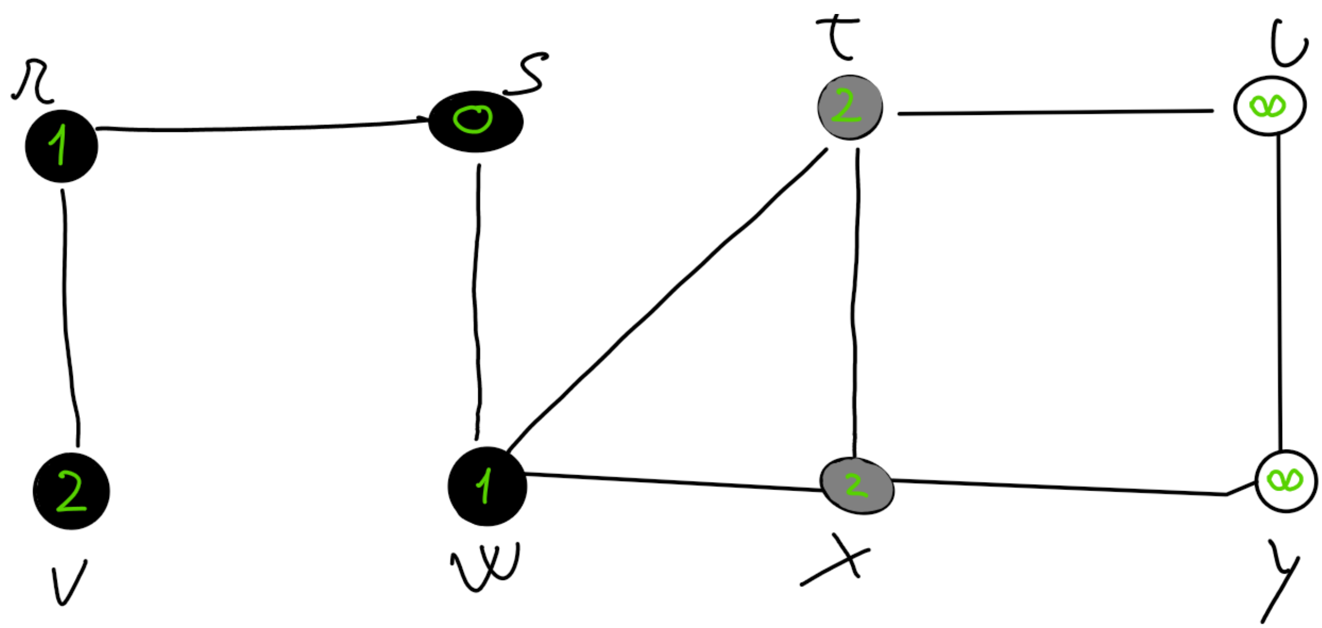




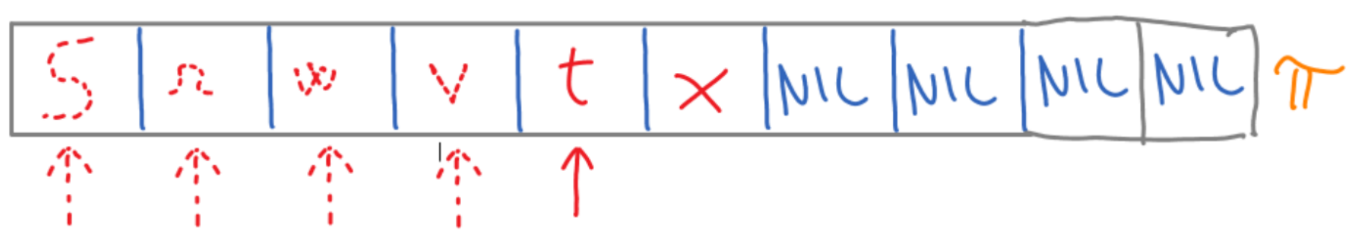
Elimino w dalla coda e faccio avanzare il mio puntatore.



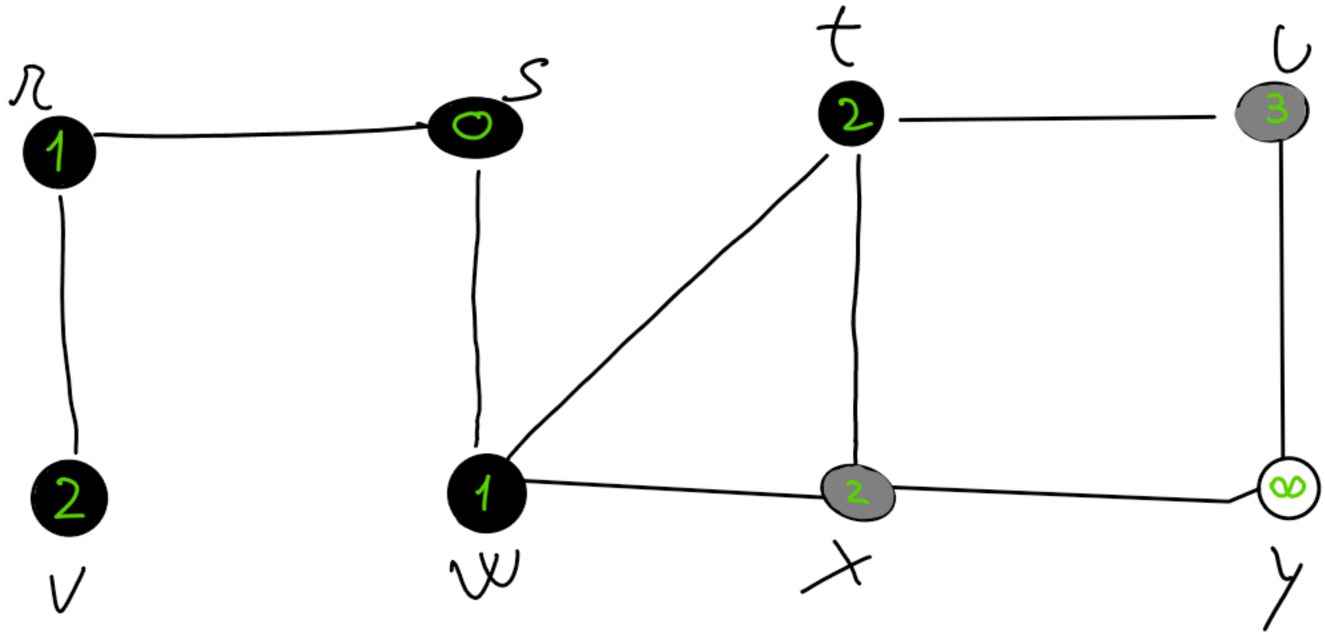
Il valore a seguire è v: ha nodi adiacenti bianchi?
Dato che v non ha altri nodi adiacenti bianchi lo coloro di **nero**.



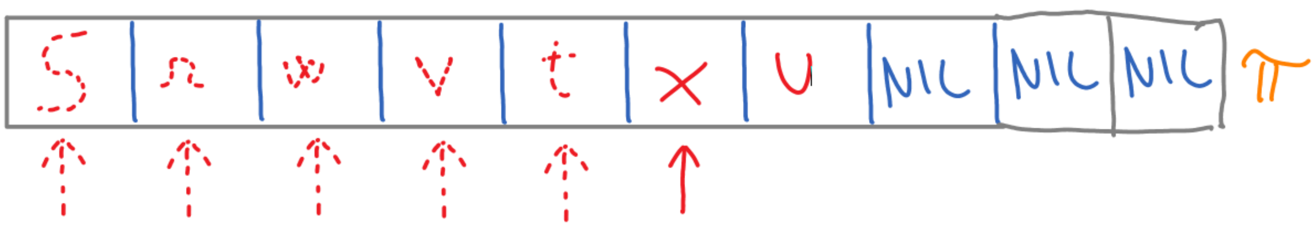
Elimino v dalla coda e faccio avanzare il mio puntatore.



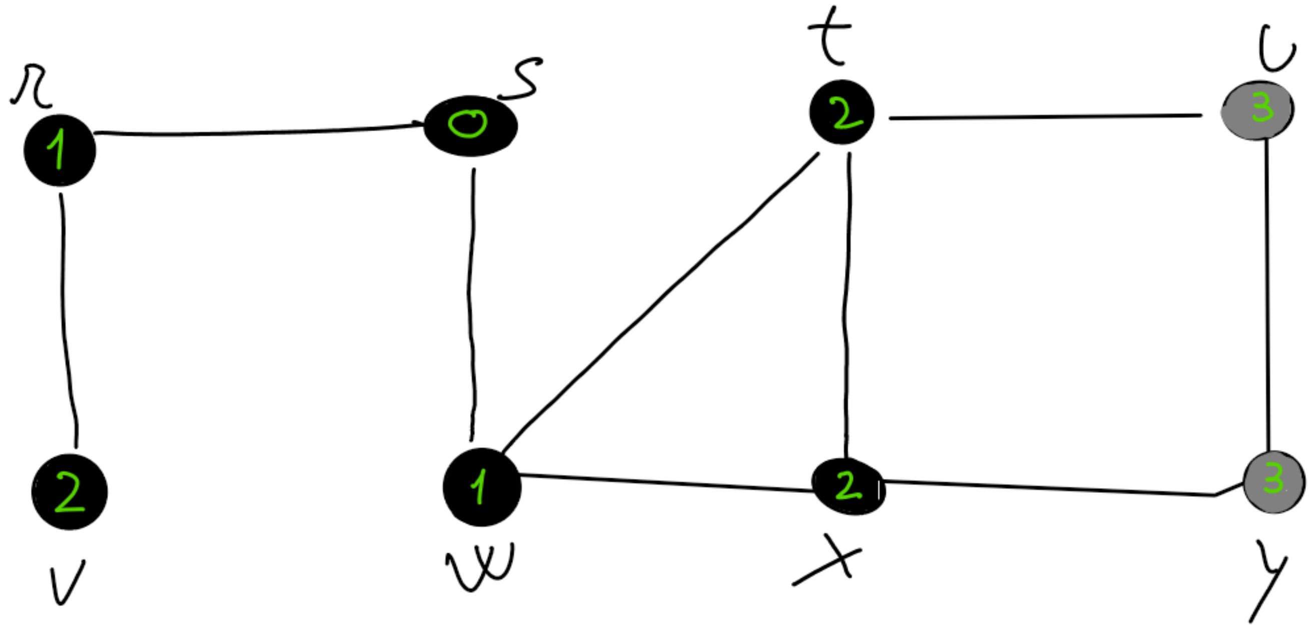
Il valore a seguire è t: ha nodi adiacenti bianchi?
Aggiungo u alla coda e così diventa grigio.
Dato che t non ha altri nodi adiacenti bianchi lo coloro di **nero**.

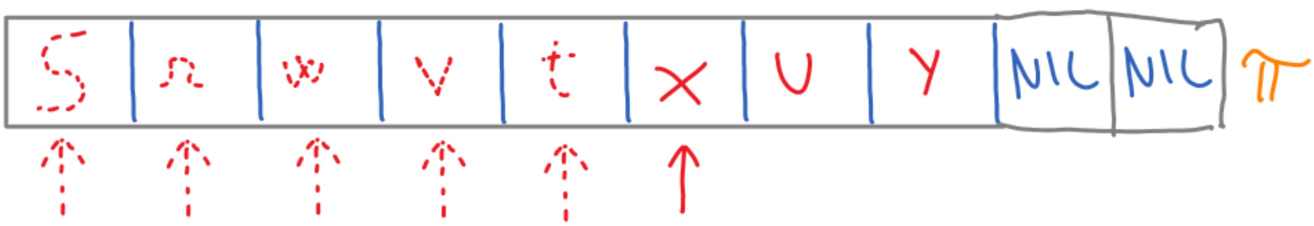


Elimino t dalla coda e faccio avanzare il mio puntatore

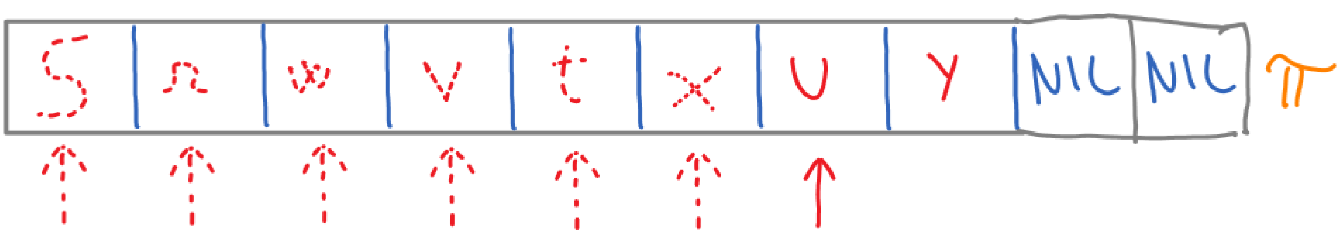


Il valore a seguire è x: ha nodi adiacenti bianchi?
Aggiungo y alla coda e così diventa grigio.
Dato che x non ha altri nodi adiacenti bianchi lo coloro di **nero**.

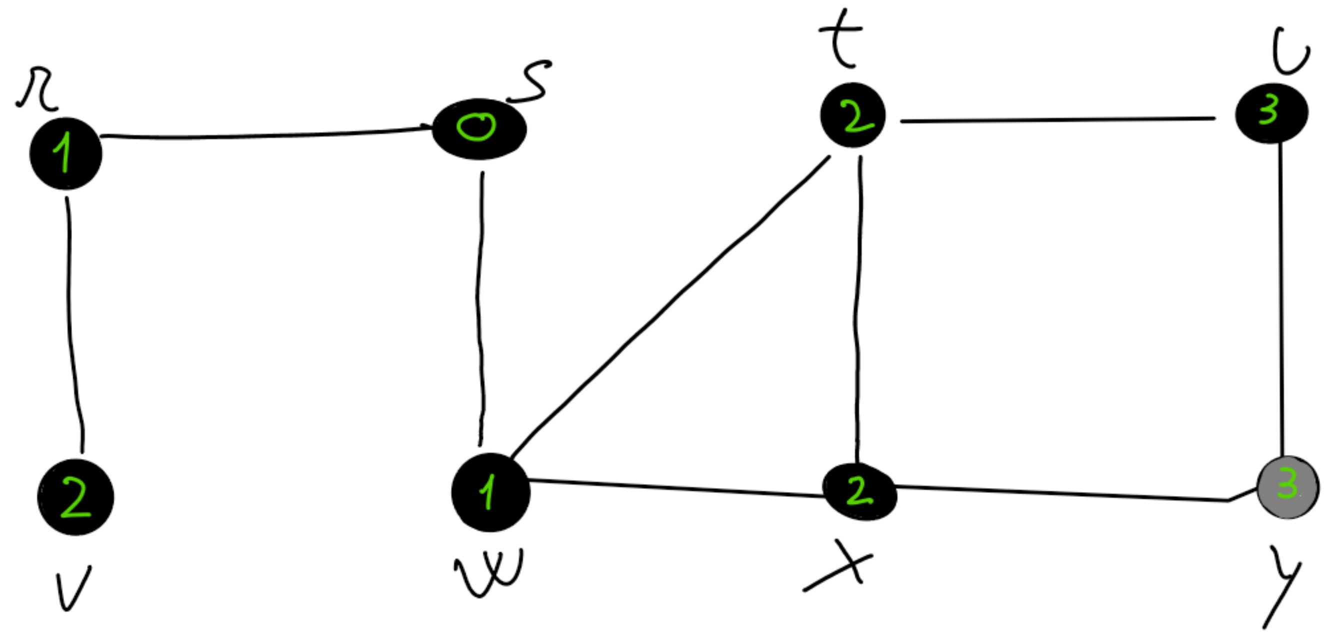




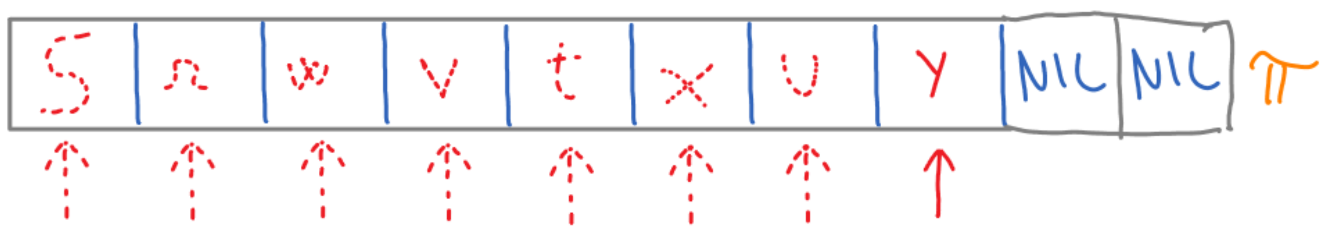
Elimino x dalla coda e faccio avanzare il mio puntatore



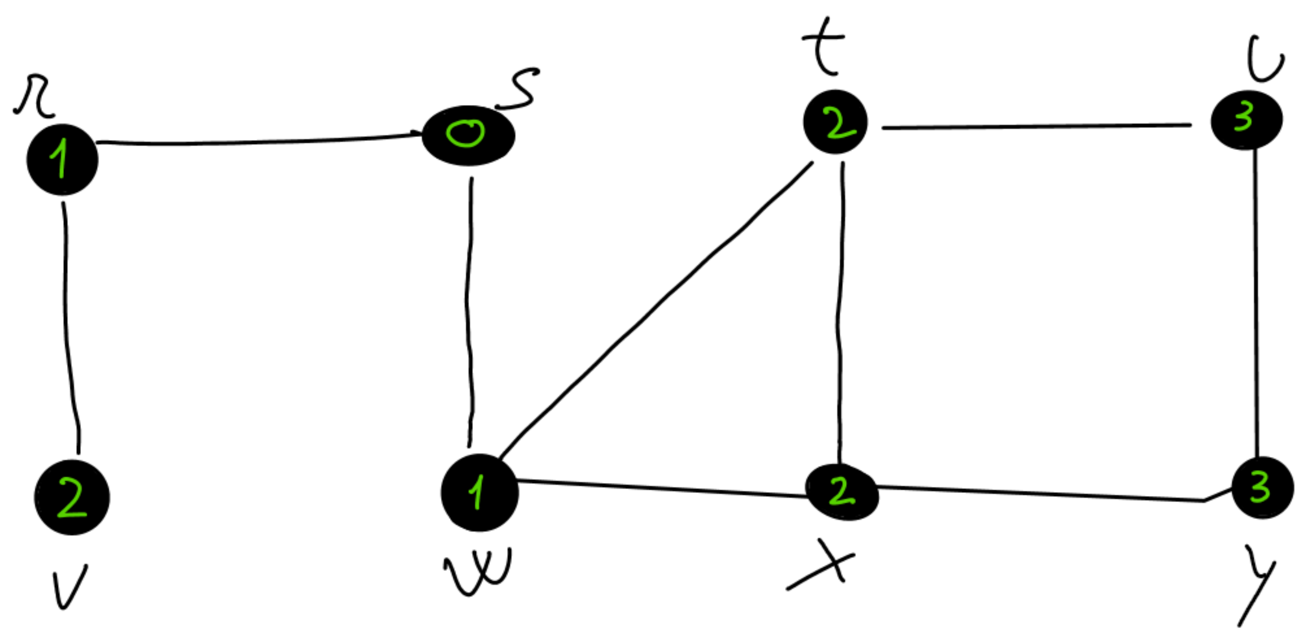
Il valore a seguire è u: ha nodi adiacenti bianchi?
Dato che u non ha altri nodi adiacenti bianchi lo coloro di **nero**.



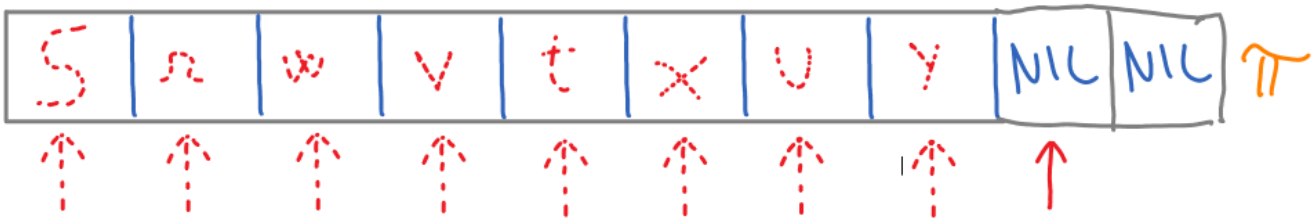
Elimino u dalla coda e faccio avanzare il mio puntatore



Il valore a seguire è y: ha nodi adiacenti bianchi?
Dato che y non ha altri nodi adiacenti bianchi lo coloro di **nero**.



Elimino y dalla coda e faccio avanzare il mio puntatore



Conclusione

Il valore a seguire è `NUL`? Ho finito.

Complessità

La complessità della **visita** è molto maggiore rispetto alla complessità dell'**inizializzazione**.

Complessità di inizializzazione:

$$O(|V|)$$

Complessità di visita:

$$O(|E|)$$

La complessità totale è pari a:

$$O(|E|) + O(|V|)$$

Pseudocodice

il seguente è solo uno pseudocodice, non va inserito in un codice.

Inizializzazione

```C++
BFS(G,s) // (grafo, nodo "s"orgente)

for (ogni vertice u nel grafo G)
    color[u] = white
    d[u] = inf //tempo di scoperta, discovery
    p[u] = NULL // predecessore di u

color[s] = grey
p[s] = null
d[s] = 0
```

Visita

```C++
Q = \coda vuota
Q.enqueue(s)
while (Q!= "vuoto" )
    u = Q.dequeue();
    for (ogni vertice v in adj[u])
        if(color[v] == white)
        color[v] = grey
        d[v] = d[u] + 1
        p[v] = u
        Q.enqueue(v)
    color[u] = black
```

Visita dei grafi in profondità

Questo tipo di visita è definita dalla *lista* (o *array*) di nodi adiacenti. Cioè prima si visitano tutti i nodi adiacenti al vertice e poi via via si passa ai vertici successivi della stessa distanza. Questo significa che *la scoperta non* avviene *per tutti i vertici* a **distanza k** in un **tempo k**.

> Nella visita **in profondità** non si ha un *nodo sorgente*, è semplicemente considerato **tutto il grafo**.

Procedimento

In questo caso la visita è da intendere come una **funzione ricorsiva** poiché se il nodo ha nodi adiacenti *bianchi* allora procedo con la visita, altrimenti la visita termina e il *nodo diventa nero*.

Significato dei colori

Prima di spiegare il procedimento è necessario comprendere cosa si intenderà con questi tre colori:

1. **bianco**: è il colore dei nodi non ancora scoperti (*all'inizio tutti i nodi sono bianchi*).
2. **grigio**, è il colore dei nodi *parzialmente* scoperti
3. **nero**, è il colore dei nodi scoperti (*alla fine tutti i nodi sono neri*).

Ordinamento topologico

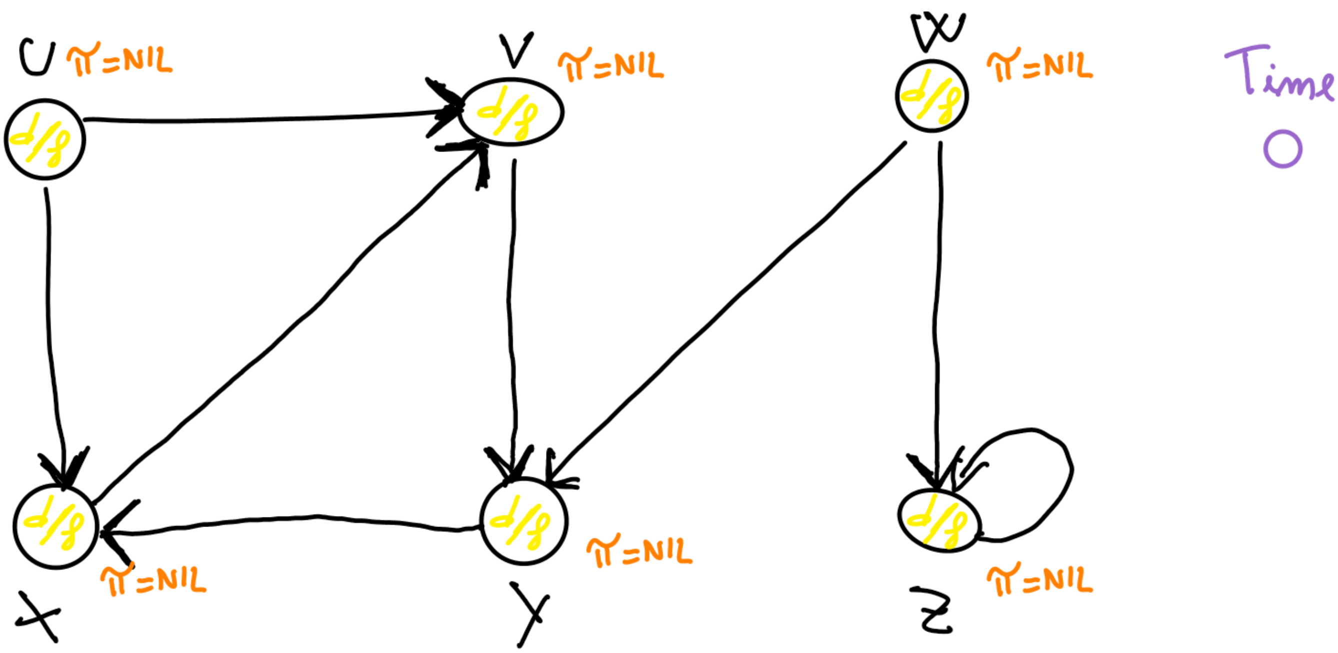
L'ordinamento topologico su un grafo orientato aciclico lo si ha quando si fa un ordinamento *in modo decrescente* rispetto al **tempo finale** della visita di ***ogni vertice***.

Inizializzazione

Prima di procedere con la visita si inizializzano:

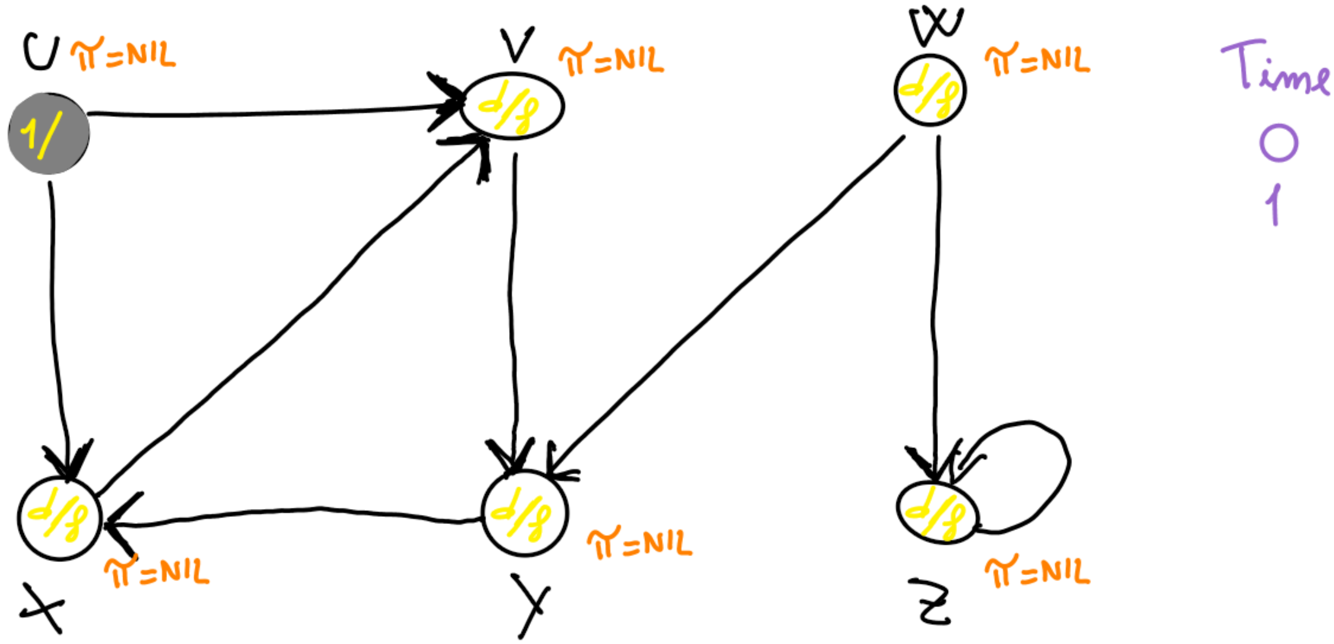
- un **array discovery** `d` (contenente *il tempo di scoperta di un nodo*) con soli valori infiniti

- un **array completamento** `f` (contenente *il tempo di completamento della visita cioè il passaggio a nero*)
- una **coda dei predecessori** `π` (contente i *predecessori dei nodi considerati in quell'istante*) con valori `NIL`
- una variabile `time` incrementata di volta in volta a dovere, *grazie a questa* sarà possibile definire d ed f.
- tutti i nodi del colore **bianco**



Svolgimento (*per un grafo orientato*)

Considero il nodo u e lo coloro di grigio.
Incremento il time di 1.
Discovery del nodo u è 1.



Considero la lista di adiacenza del nodo considerato, quindi di u.

[u -> x -> v]

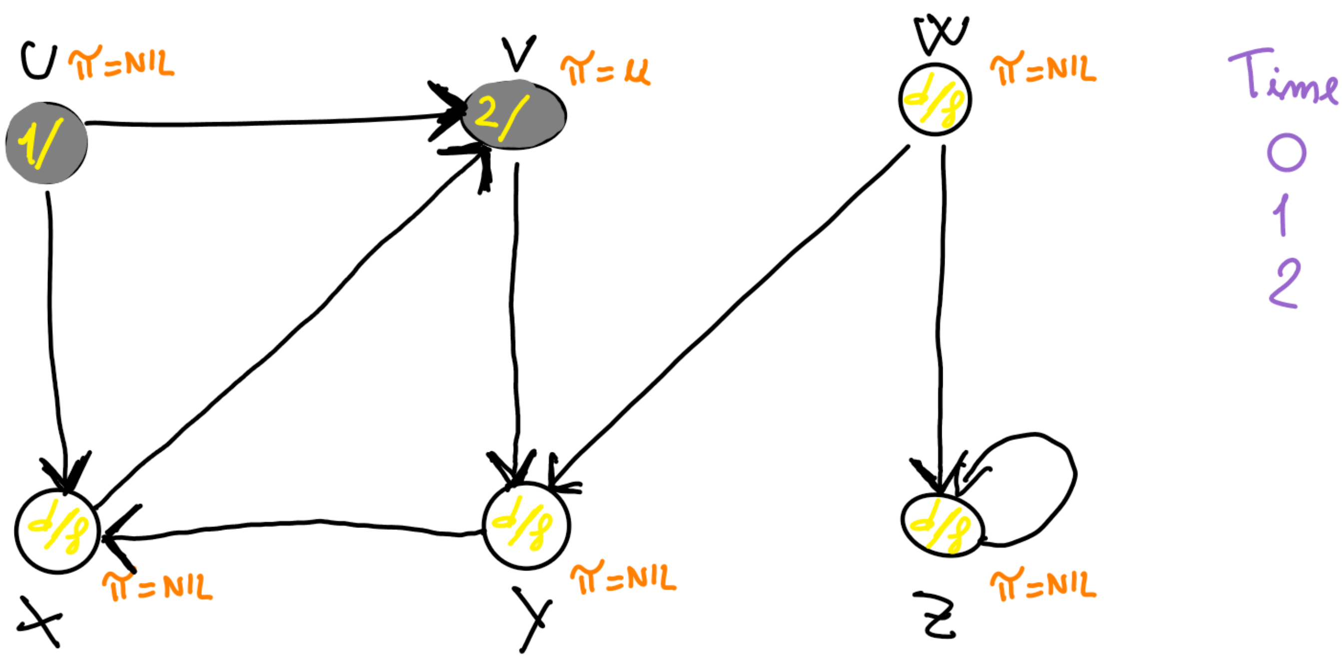
I nodi adiacenti ad u sono bianchi?

Imposto il predecessore ad u. `π(v) = u`

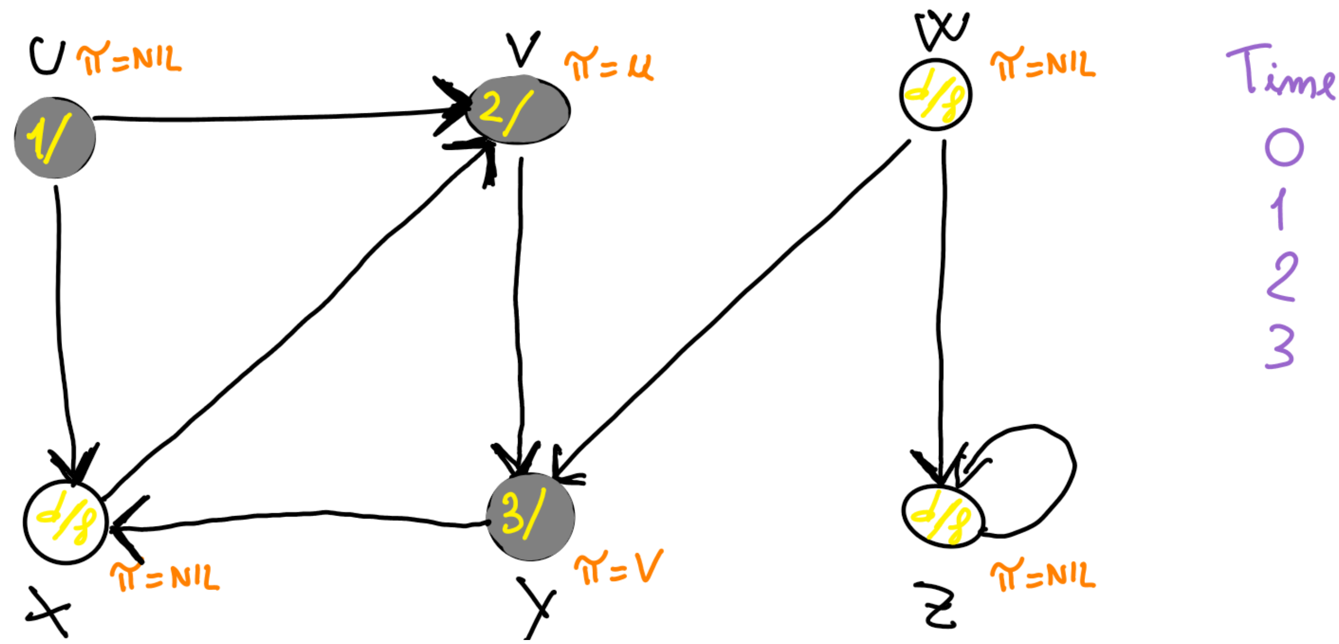
Considero il nodo v e lo coloro di grigio.

Incremento il time di 1. Diventa 2.

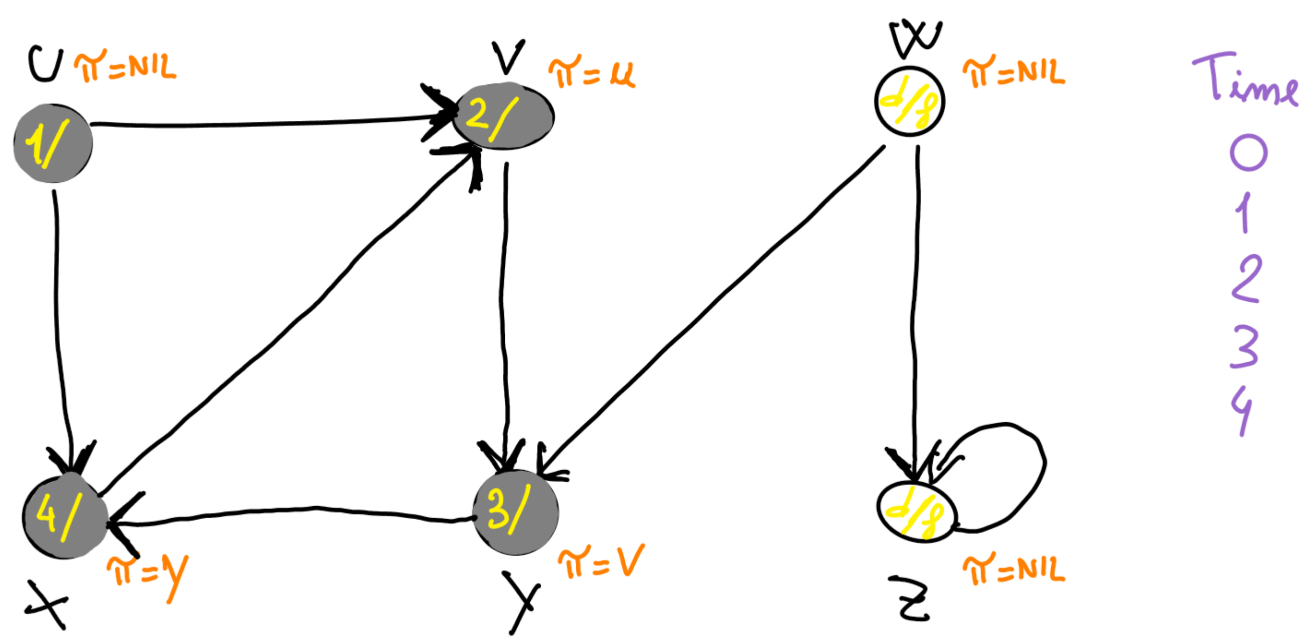
Discovery del nodo v è 2



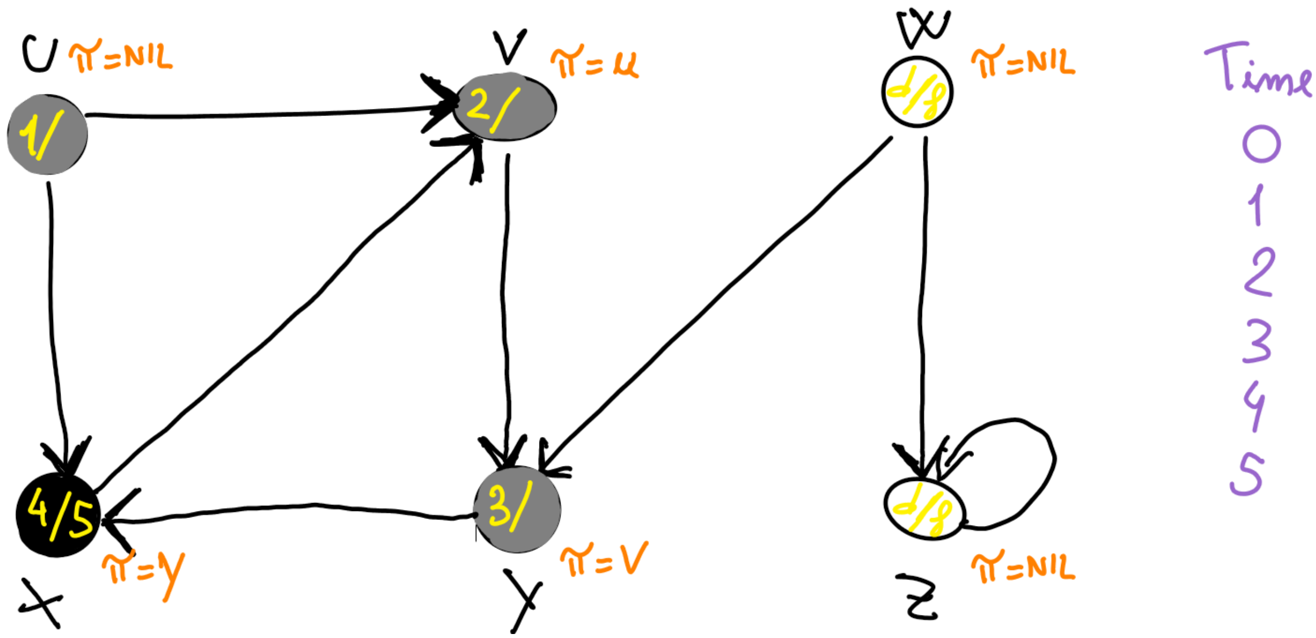
I nodi adiacenti a v sono bianchi?
Considero il nodo y e lo coloro di grigio.
Incremento il time di 1. Diventa 3.
Discovery del nodo y è 3.



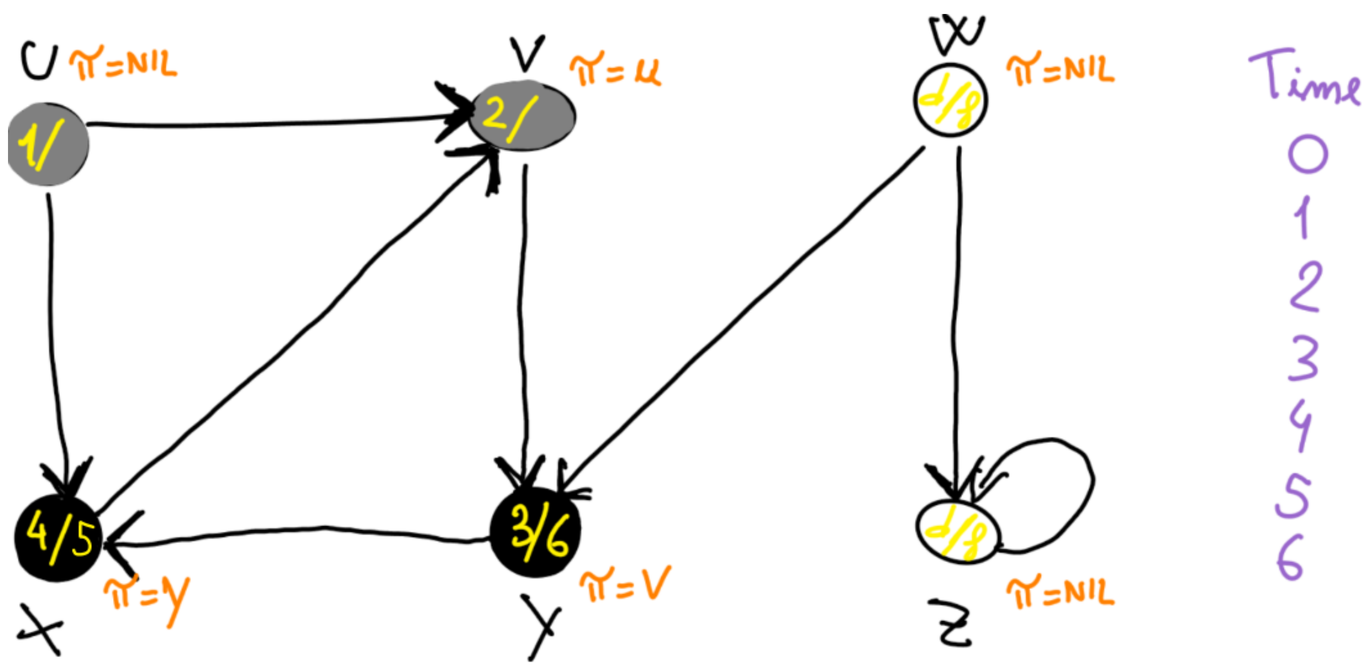
I nodi adiacenti a y sono bianchi?
Considero il nodo x e lo coloro di grigio.
Incremento il time di 1. Diventa 4.
Discovery del nodo x è 4.



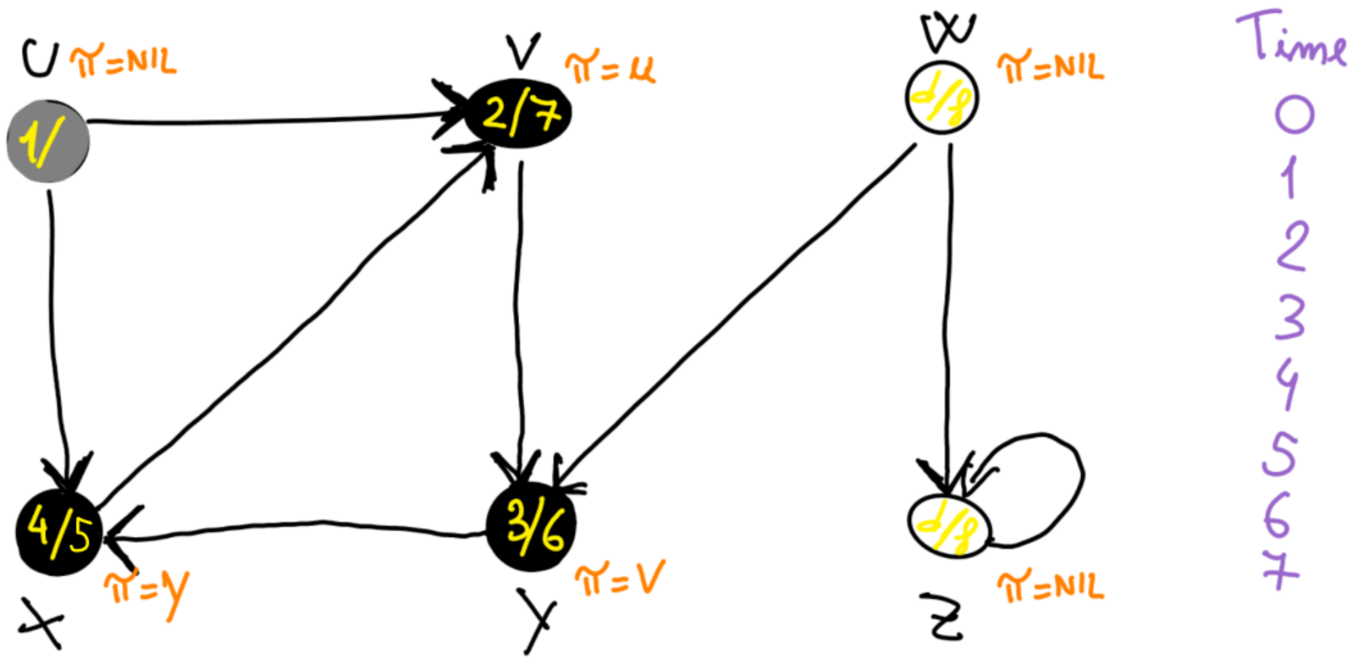
I nodi adiacenti a x sono bianchi?
Incremento il time di 1. Diventa 5.
Il tempo di fine scoperta di x è 5, x è **nero**.



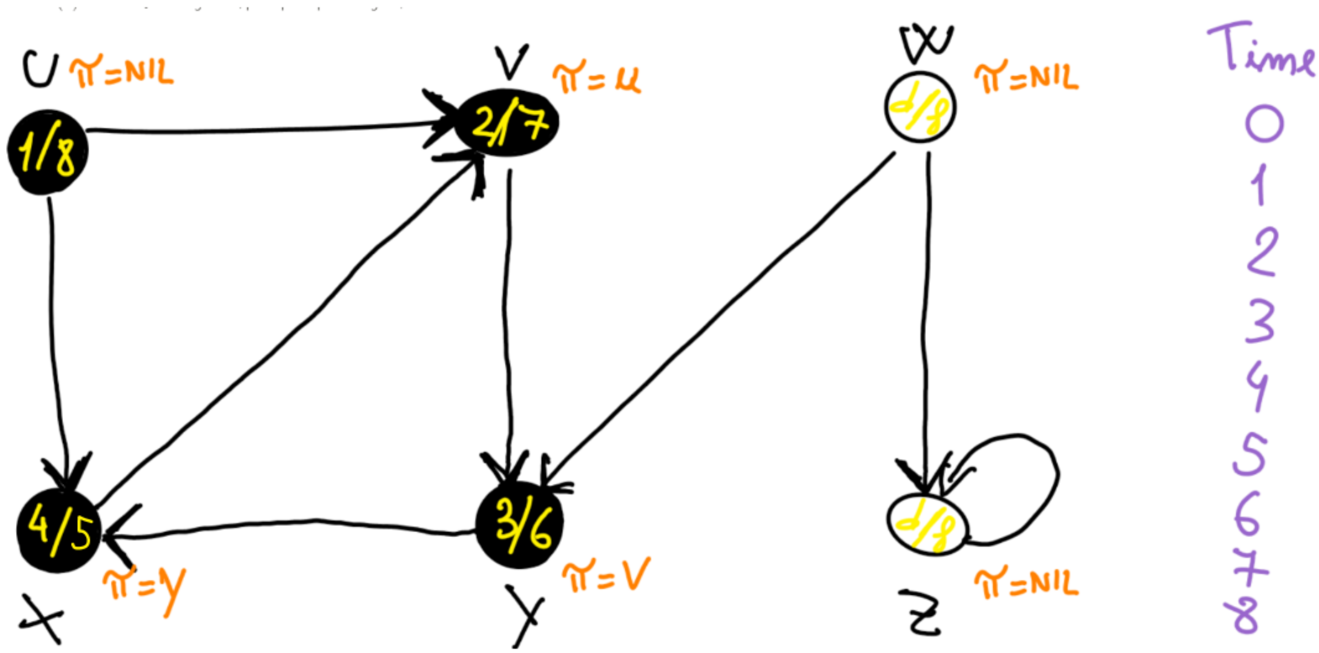
Torno indietro mediante i predecessori in π.
I nodi adiacenti a y sono bianchi?
Incremento il time di 1. Diventa 6.
Il tempo di fine scoperta di y è 6, y è **nero**.



Torno indietro mediante i predecessori in π.
I nodi adiacenti a v sono bianchi?
Incremento il time di 1. Diventa 7.
Il tempo di fine scoperta di v è 7, v è **nero**



Torno indietro mediante i predecessori in π.
I nodi adiacenti a u sono bianchi?
Incremento il time di 1. Diventa 8.
Il tempo di fine scoperta di u è 8, u è **nero**



Torno indietro mediante i predecessori in π. Ho `NULL`

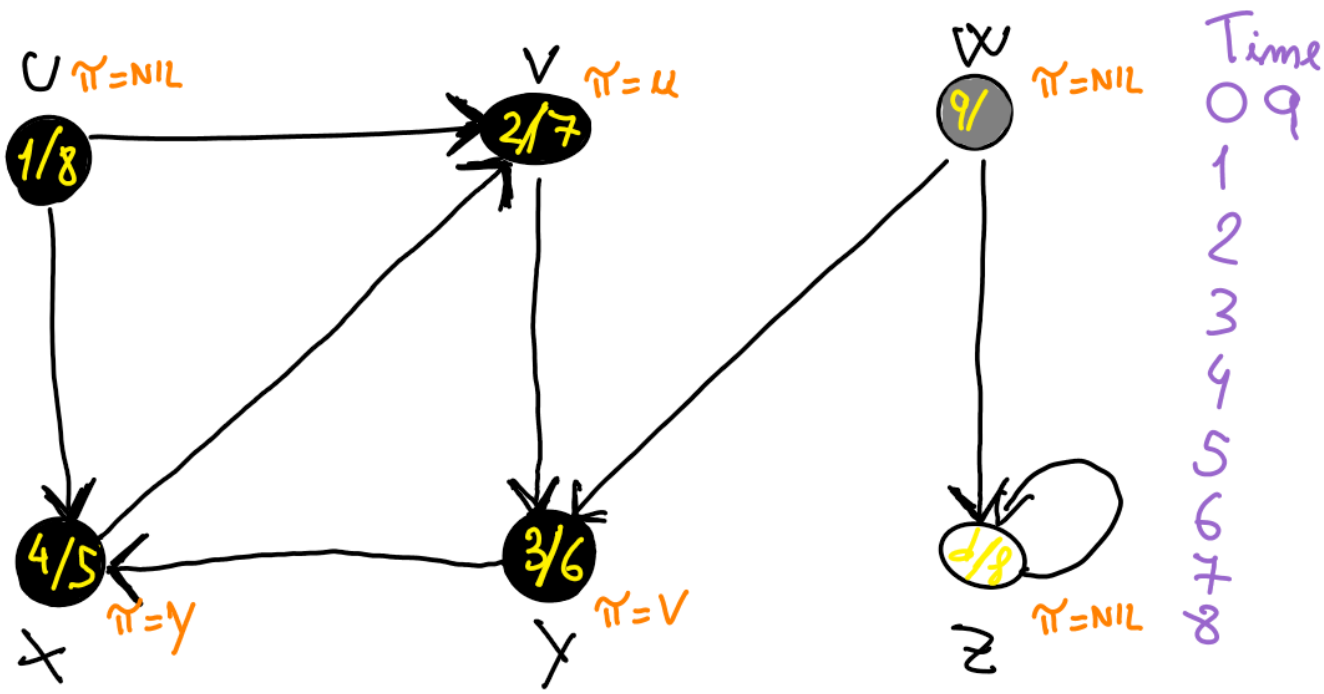
Ho raggiunto tutti i nodi del grafo? No.

Procedo con l'altro lato del grafo.

Considero il nodo w e lo coloro di grigio.

Incremento il time di 1. Diventa 9.

Discovery del nodo w è 9

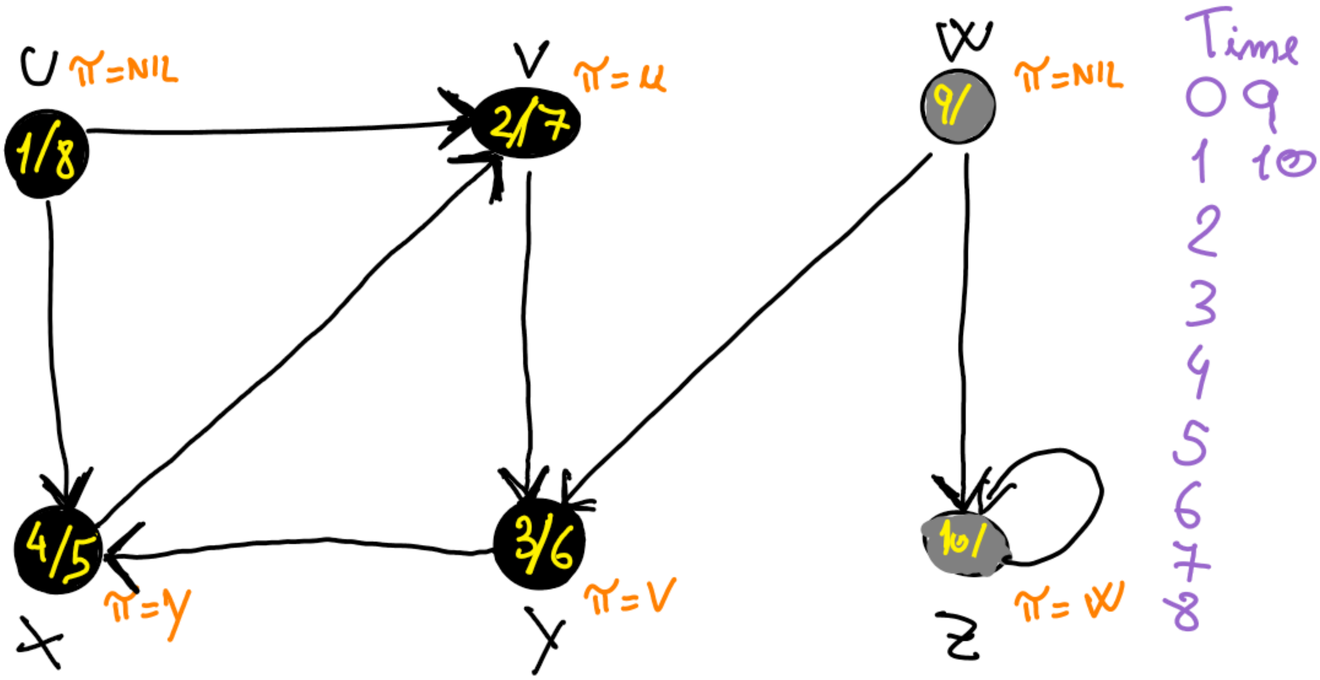


I nodi adiacenti a w sono bianchi?

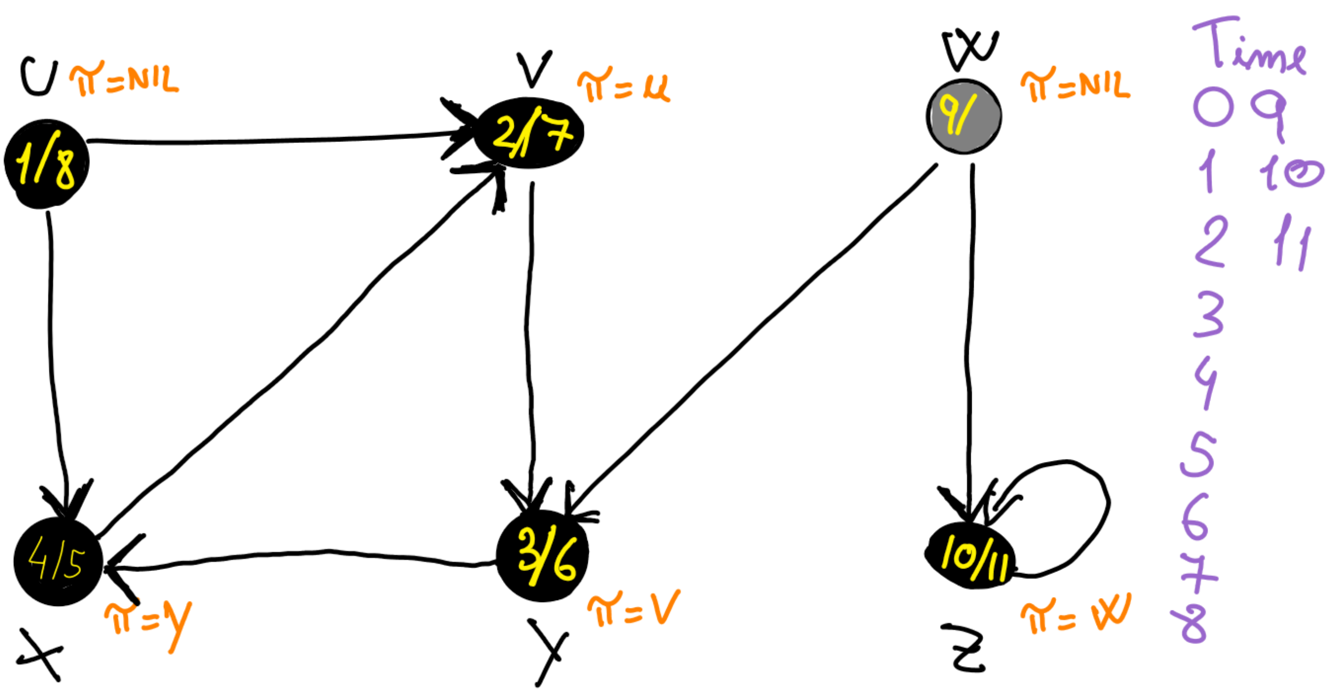
Considero il nodo z e lo coloro di grigio.

Incremento il time di 1. Diventa 10.

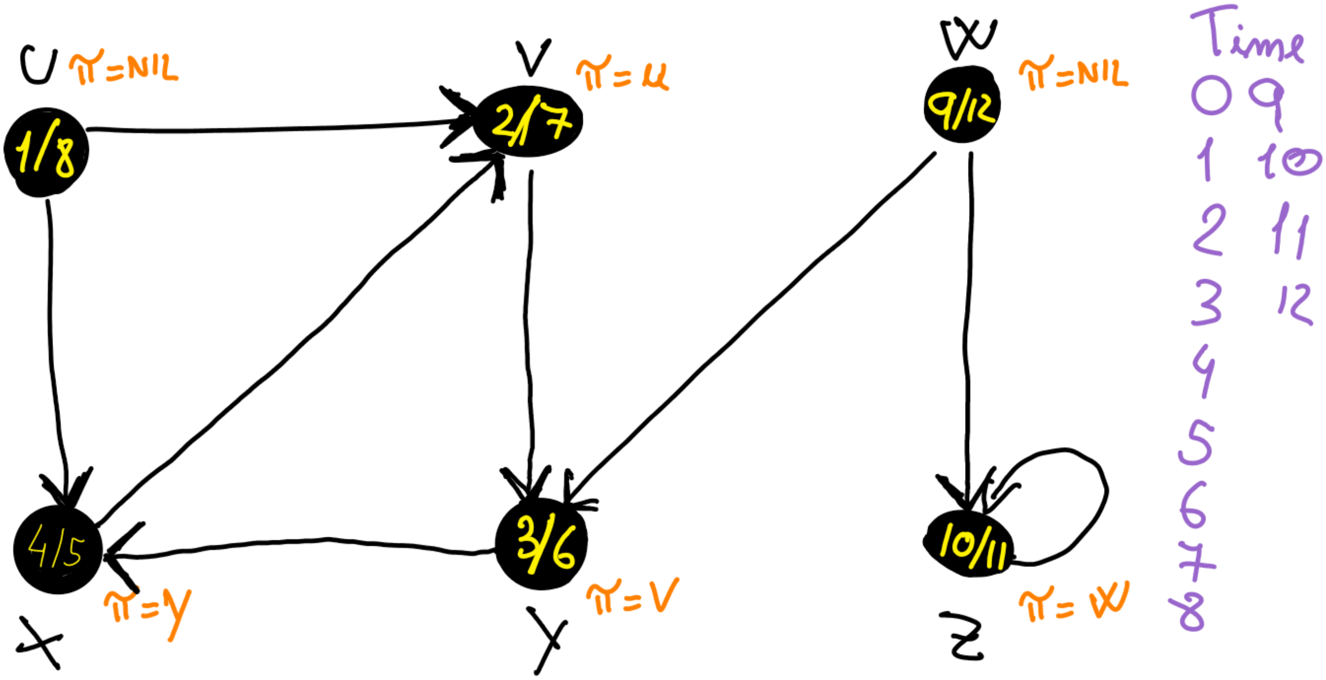
Discovery del nodo z è 10.



I nodi adiacenti a z sono bianchi?
Incremento il time di 1. Diventa 11.
Il tempo di fine scoperta di z è 11, z è **nero**



Torno indietro mediante i predecessori in π.
I nodi adiacenti a w sono bianchi?
Incremento il time di 1. Diventa 12.
Il tempo di fine scoperta di w è 12, w è **nero**



Conclusione

Torno indietro mediante i predecessori in π. Ho `NULL`
Ho raggiunto tutti i nodi del grafo? Sì.
Ho finito la procedura.

Complessità

La complessità della **visita** è molto maggiore rispetto alla complessità dell'**inizializzazione**.

Complessità di inizializzazione:

$$O(|V|)$$

Complessità di visita:

$$O(|E|)$$

La complessità totale è pari a:

$$O(|E|) + O(|V|)$$

Pseudocodice

```C++
DFS(g)
for (ogni vertice u in G) //|V| volte
    color[u] = white
    d[u] = 0
    f[u] = inf
    p[u] = NULL
global time = 0
for (ogni vertice u in G)
    if (color[u] == white)
        DFS - visit(u)
```

```C++
DFS - visit(u)
    color[u] = gray
    time = time+1
    d[u] = time
    for (v in adj[u]) // |E| volte
        if (color[v] == white)
            p[v] = u
            DFS - visit(v)
color[u] = black
time = time+1
f[u] = time
```