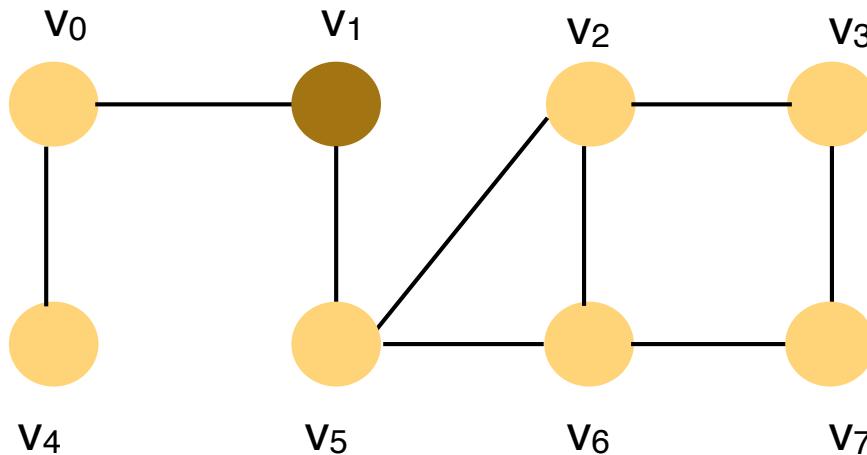


# algoritmi di visita di un grafo

# visita di un grafo

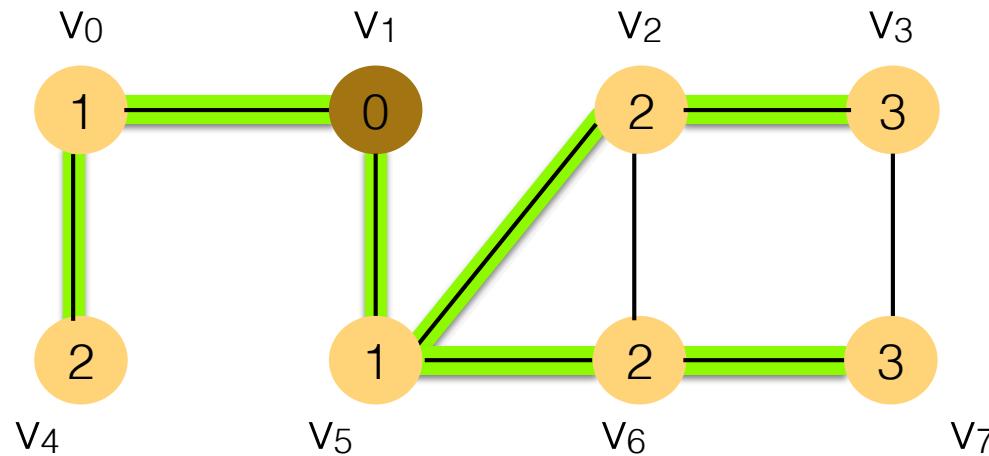
**PROBLEMA:** dato un grafo  $G$  e un suo nodo  $s$ , scoprire tutti i nodi di  $G$  raggiungibili da  $s$ , e trovare la loro distanza minima da  $s$

- Dato un grafo  $G=(V,E)$  ed un **nodo sorgente  $s$**  in  $V$ , definiamo un **algoritmo** che **visita sistematicamente il grafo  $G$**  per scoprire tutti i *nodi raggiungibili da  $s$*



# algoritmo Breadth-First Search

la visita è detta in **ampiezza** perché l'algoritmo scopre tutti i vertici a distanza **1**, poi scopre tutti quelli a distanza **2**, ...

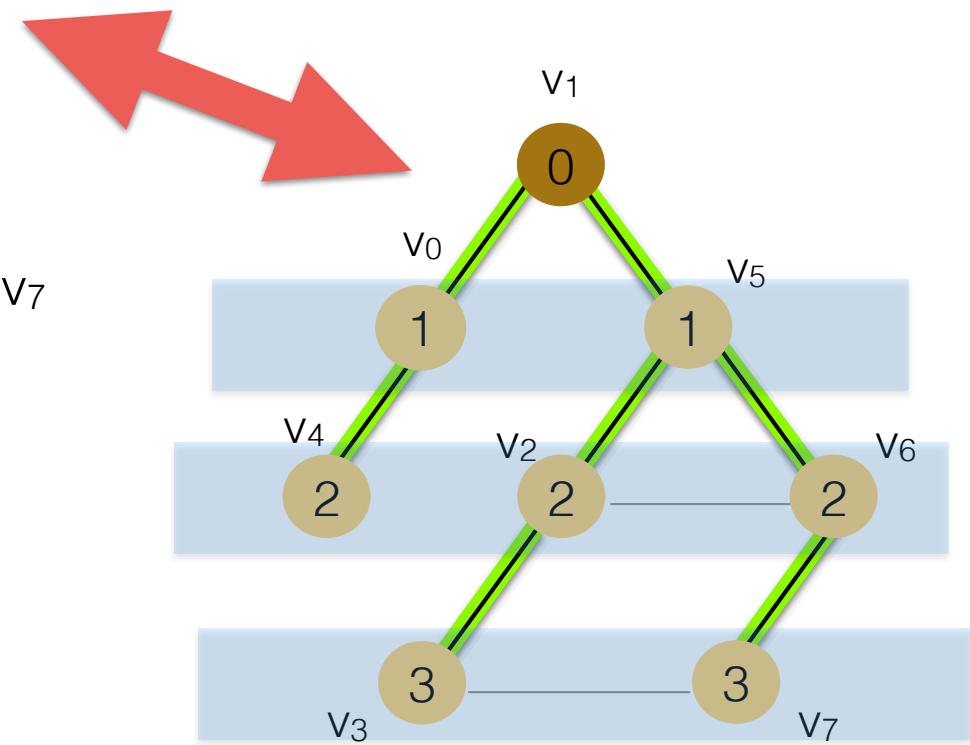
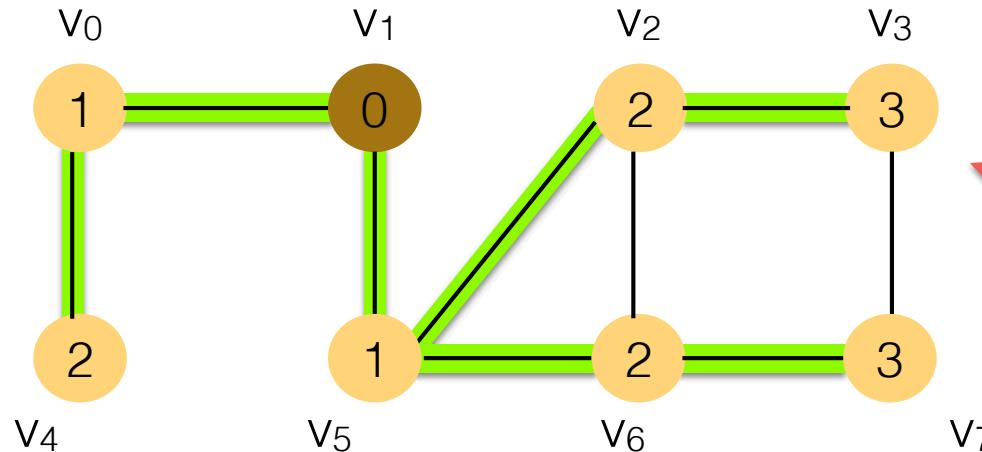


nel contempo

- **calcola la distanza minima**, cioè la lunghezza del cammino minimo, **tra ogni nodo e la sorgente s** (i numeri inseriti nei vertici)
- e produce l'**albero della visita, i cui rami sono cammini minimi** (archi colorati di verde)

# algoritmo Breadth-First Search

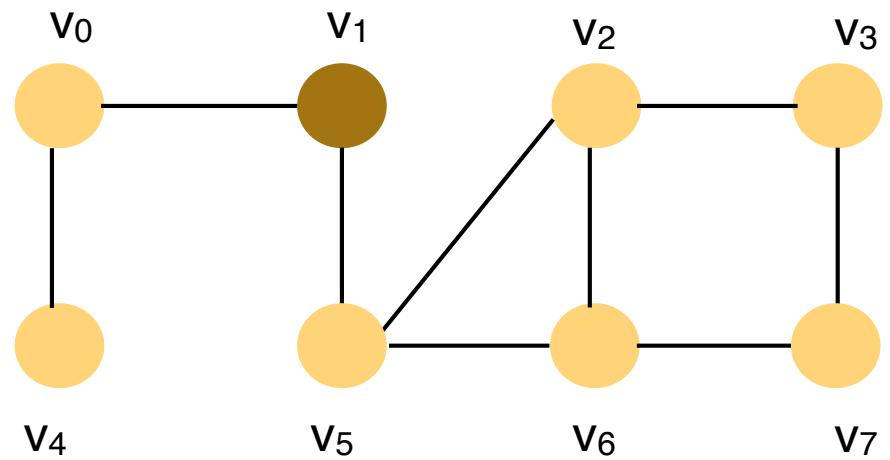
la visita è detta in **ampiezza** perché l'algoritmo scopre tutti i vertici a distanza  $k$  prima di scoprire quelli a distanza  $k+1$



usato anche da **The Oracle of Bacon** website.  
A breadth-first search from the vertex for  
**Kevin Bacon** finds the shortest chain to all  
other actors and actresses.

# definizione dell'algoritmo

- Dato un grafo G ed un *nodo sorgente* s, **definiamo una sequenza di passi** che permettono di visitare tutti i nodi raggiungibili da s
- L'algoritmo si definisce in **pseudo-codice**
- L'algoritmo andrà poi **implementato** in un programma/software **eseguibile**, scritto in uno specifico linguaggio di programmazione



```
lista = [[1,4],  
[0,5],  
[3,5,6],  
[2,7],  
[0],  
[1,2,6],  
[2,5,7],  
[3,6]]
```

# algoritmo Breadth-First Search



Idea

- per mantenere traccia del punto in cui si è arrivati nella visita **coloriamo i nodi**:

○ mai visitato

● visitato e sulla frontiera

● visitato e non più su frontiera

- memorizziamo la **Frontiera** come una **coda** (elenco FIFO) di nodi



- ogni nodo **v** memorizza 3 informazioni, utili per costruire l'albero di visita con i cammini minimi e relative distanze
  - **v.color** il colore di **v**
  - **v.dist** la distanza di **v** dal nodo sorgente **s**
  - **v.padre** il nodo che precede **v** nel cammino minimo che separa **v** da **s**

# Passi di Inizializzazione:

1. tutti i nodi: assegna colore bianco, distanza  $\infty$  e padre - (indefinito)
2. nodo s: assegna colore grigio e distanza 0 (padre - )
3. inserisce s nella Frontiera

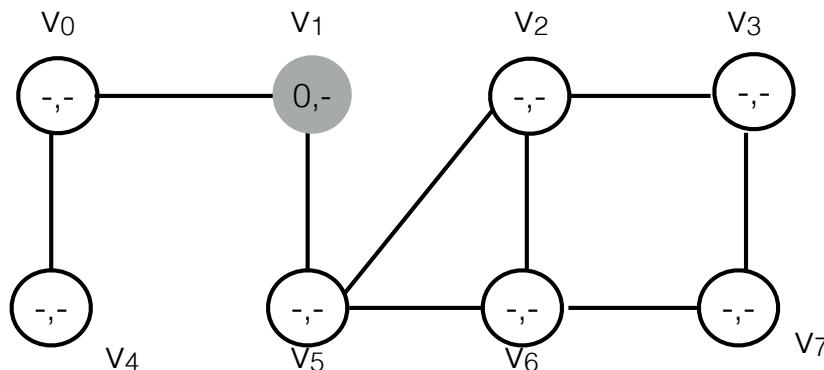
## pseudo-codice

`:=` indica l'assegnamento,  
`forEach` sta per una iterazione  
`v.color`, bianco, `Front.add(s)`, ecc..  
sono **scritture intuitive**, non  
costrutti di programmazione

```
foreach v in V
    v.color := bianco
    v.dist  :=  $\infty$ 
    v.padre := -
    s.color := grigio
    s.dist  := 0
    Front.add(s)
```

Front

V1



mai visitato

visitato e **sulla frontiera**

visitato e non più su  
frontiera

## Corpo:

**finché** la frontiera non è vuota

prendo il primo nodo in coda, ***u***

**per ogni** nodo ***v*** adiacente a ***u***

**se** ***v*** è bianco **allora**

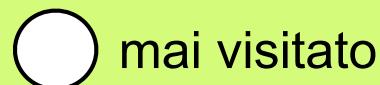
**aggiungo *v* alla frontiera**  
**colorandolo di grigio** e  
aggiorno *v.dist* e *v.padre*

coloro ***u*** di nero

```
while Front.isEmpty == false
    u := Front.pop
    foreach v in AdjList(u)
        if v.color == bianco
            v.color := grigio
            v.dist := u.dist + 1
            v.padre := u
            Front.add(v)
    u.color := nero
```



Idea



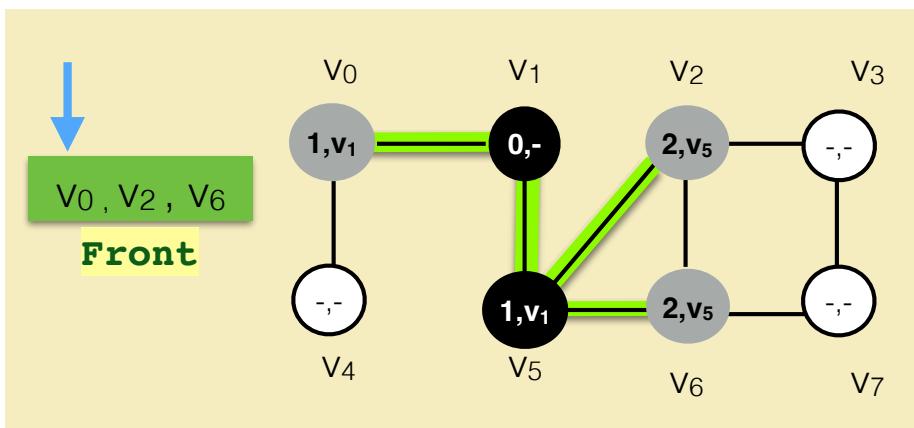
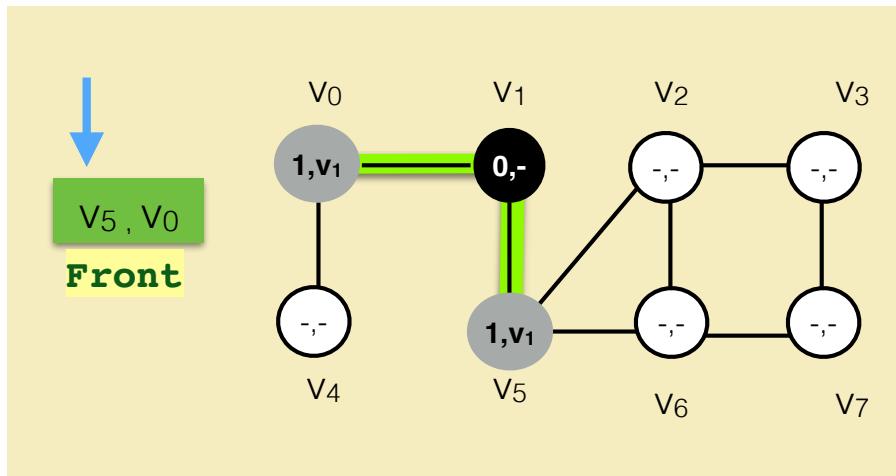
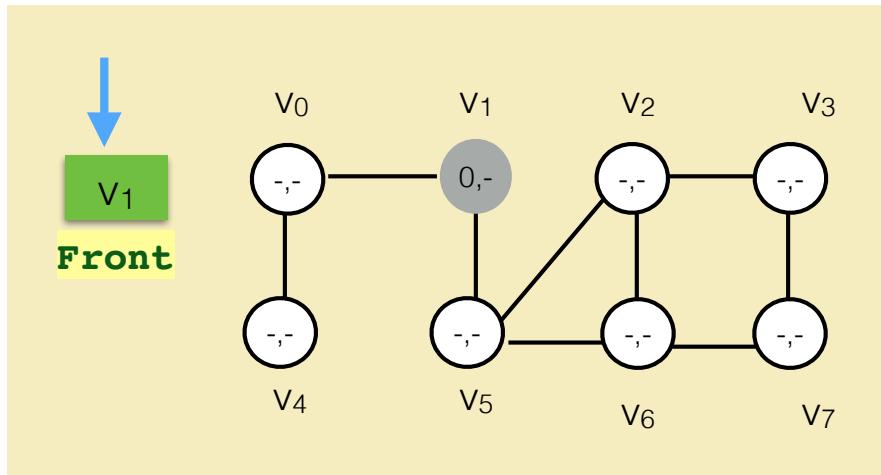
mai visitato



visitato e **sulla frontiera**

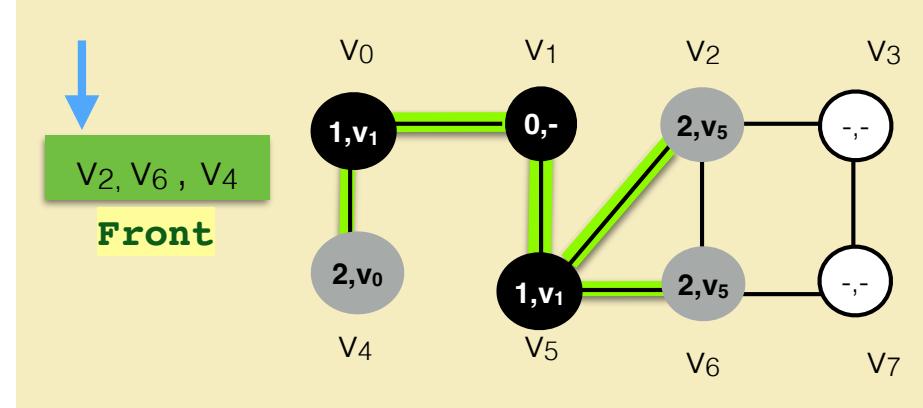


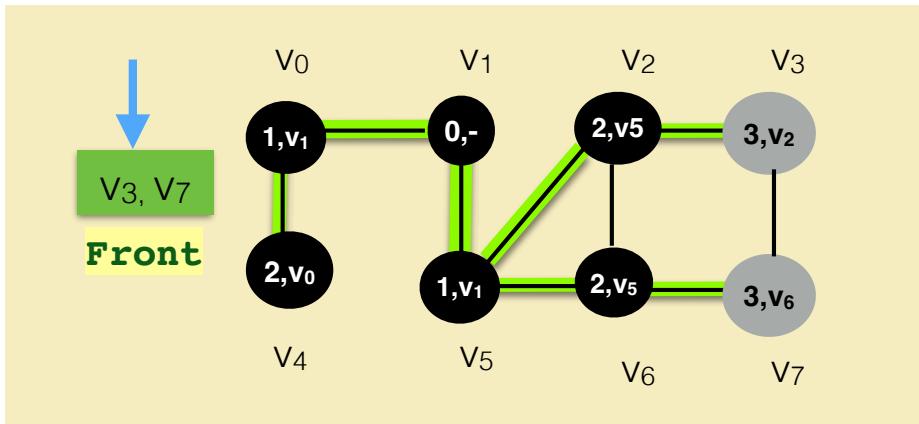
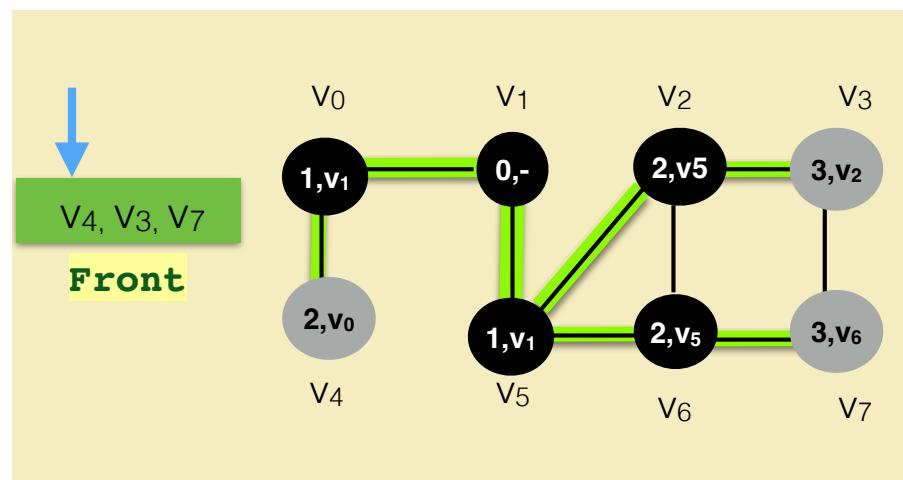
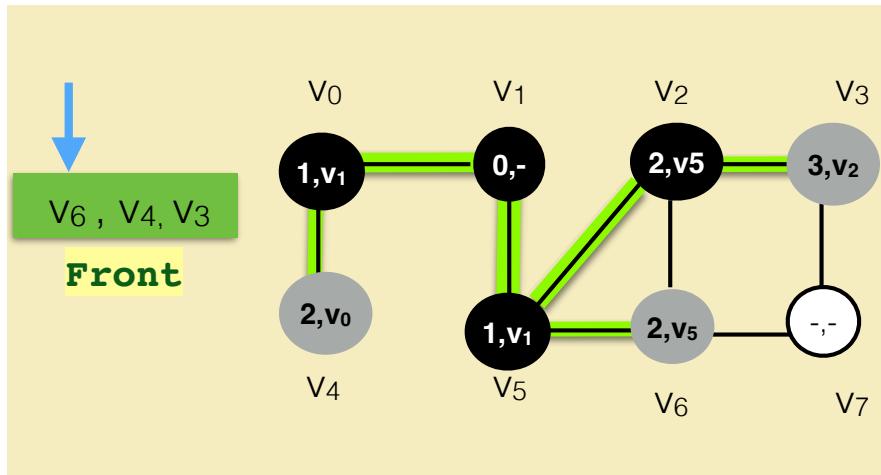
visitato e non più  
su frontiera



```

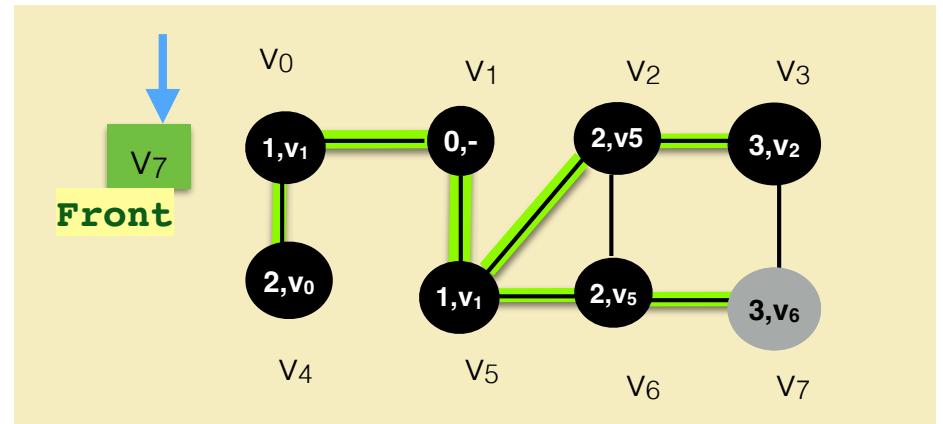
while Front.isEmpty==false
  u := Front.pop
  forEach v in AdjList(u)
    if v.color == bianco
      v.color := grigio
      v.dist := u.dist + 1
      v.padre := u
      Front.add(v)
    u.color := nero
  
```

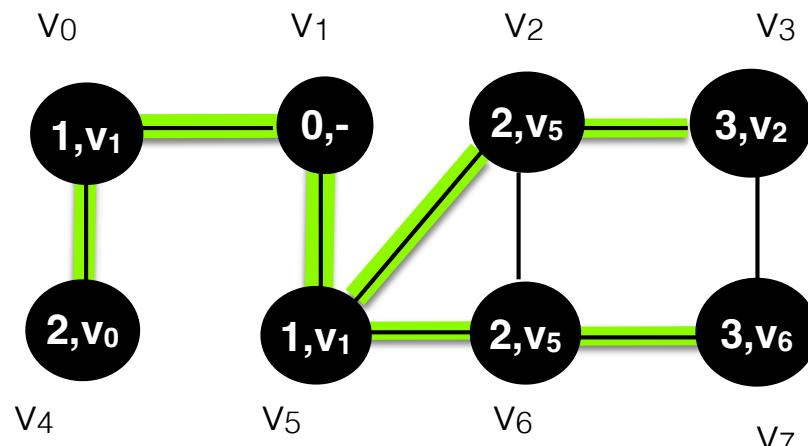




```

while Front.isEmpty==false
  u := Front.pop
  forEach v in AdjList(u)
    if v.color == bianco
      v.color := grigio
      v.dist := u.dist + 1
      v.padre := u
      Front.add(v)
    u.color := nero
  
```





```

while Front.isEmpty==false
    u := Front.pop
    forEach v in AdjList(u)
        if v.color == bianco
            v.color := grigio
            v.dist := u.dist + 1
            v.padre := u
            Front.add(v)
        u.color := nero
    
```

## come facciamo a sapere che il risultato è giusto?

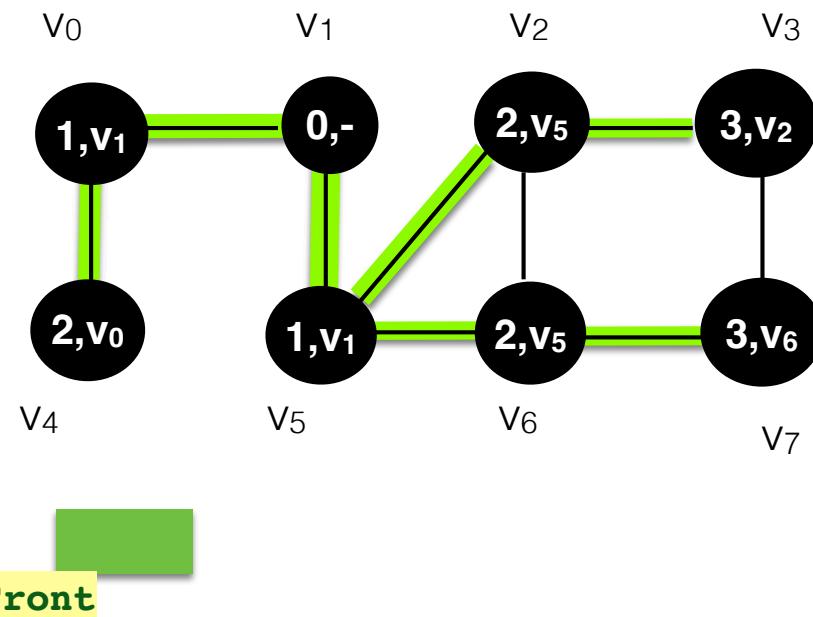
magari, se eseguito su un *altro grafo*, questo algoritmo dimentica di visitare qualche nodo, o segna un cammino che non è minimo

# Proprietà di un Algoritmo

- Dato un algoritmo, scritto in pseudo-codice, si devono dimostrare due proprietà:
- **correttezza**
  - dato un *qualsiasi* grafo G ed un suo nodo s, l'algoritmo BST **termina e visita correttamente** (...) tutti i nodi
- **efficienza - complessità**
  - si fornisce una stima di **quanti passi di esecuzione** sono necessari per l'esecuzione completa dell'algoritmo
  - si fornisce una stima di **quanta memoria viene occupata**, data dalla dimensione delle strutture dati richieste

complessità  
temporale

complessità  
spaziale



```

while Front.isEmpty==false
    u := Front.pop
    forEach v in AdjList(u)
        if v.color == bianco
            v.color := grigio
            v.dist := u.dist + 1
            v.padre := u
            Front.add(v)
        u.color := nero
    
```

**come faccio a sapere che il risultato è giusto?**

- **correttezza** dell'algoritmo BFS
- **efficienza: complessità** dell'algoritmo ( $O(|V|+|E|)$ )

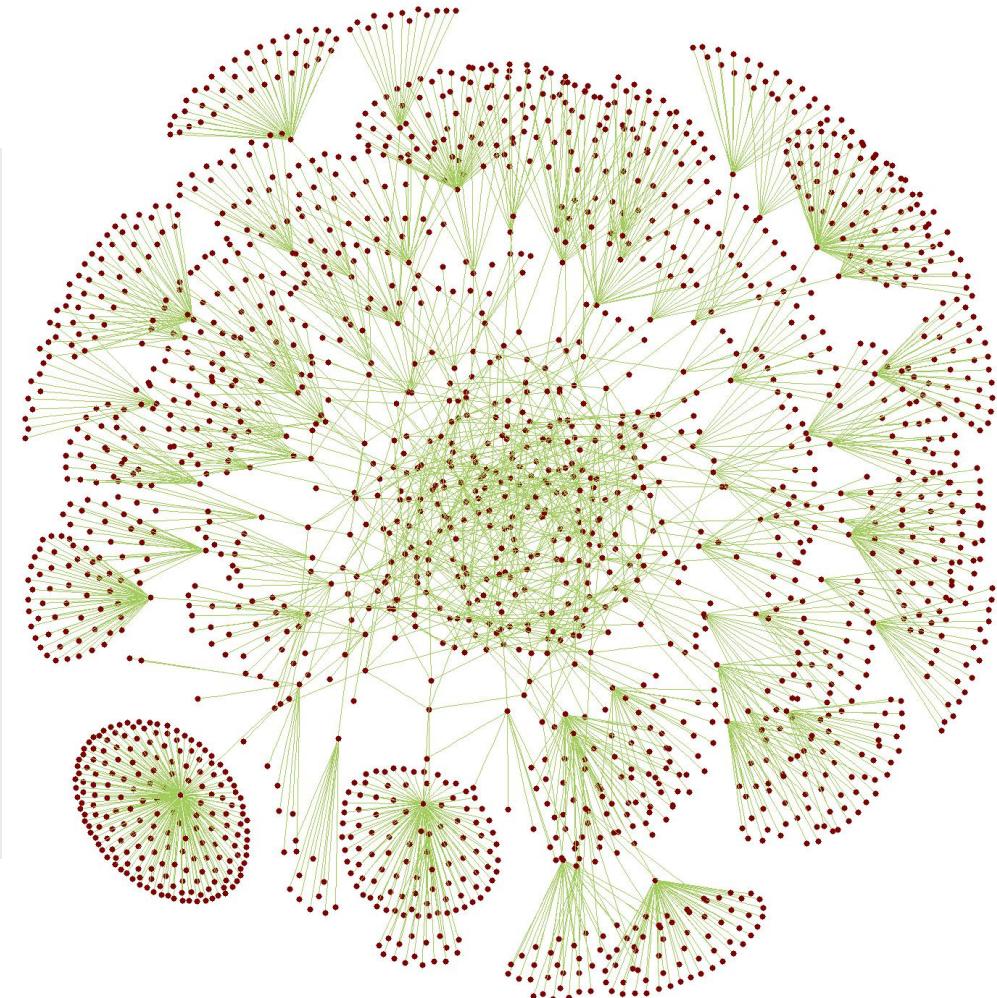
**Problema:** dato un grafo G e nodo iniziale s, visitare tutti i nodi e trovare i cammini minimi da s a ogni nodo



## BFS

```
foreach v in V
    v.color := bianco
    v.dist := ∞
    v.padre := nil
s.color := grigio
s.dist := 0
Front.add(s)

while Front.isEmpty==false
    u := Front.pop
    foreach v in AdjList(u)
        if v.color == bianco
            v.color := grigio
            v.dist := u.dist + 1
            v.padre := u
            Front.add(v)
    u.color := nero
```

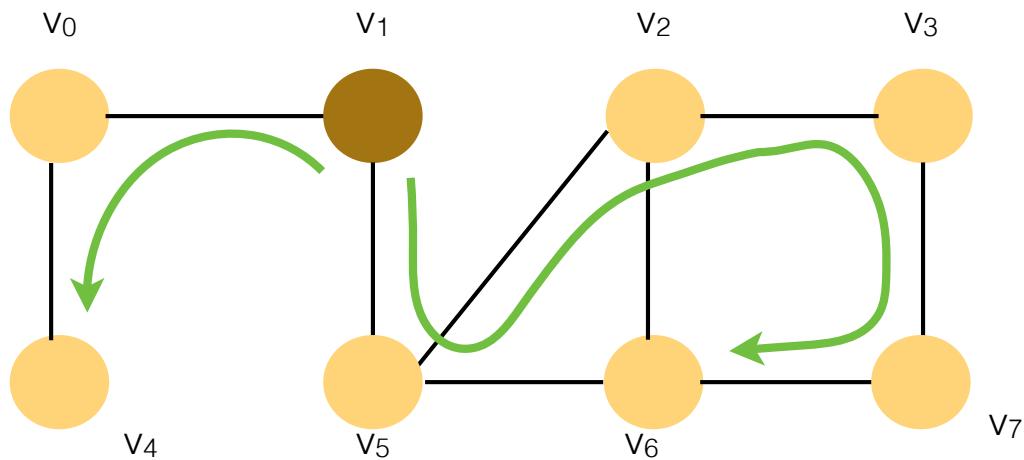


- 1. Dimostro che è corretto**
- 2. Lo implemento in Python**
- 3. Lo uso**

- La **correttezza** assicura che l'esecuzione **terminerà**
- la **complessità** fornisce una stima di **quanto durerà** l'esecuzione

# Algoritmo Depth-First Search

la visita è detta in **profondità** perché, invece di fare un passo alla volta in tutte le direzioni, ora **si va avanti in una direzione finché' possibile**, cioè ogni volta si esplora un arco uscente dall'ultimo nodo raggiunto



ordine di visita:  $v_1, v_5, v_2, v_3, v_7, v_6, v_0, v_4$

**Problema:** dato un grafo G e nodo iniziale s, visitare tutti i nodi e trovare i cammini minimi



Idea

BFS

```
forEach v in V
    v.color := bianco
    v.dist := ∞
    v.padre := nil
s.color := grigio
s.dist := 0
Front.add(s)

while Front.isEmpty==false
    u := Front.pop
    forEach v in AdjList(u)
        if v.color == bianco
            v.color := grigio
            v.dist := u.dist + 1
            v.padre := u
            Front.add(v)
    u.color := nero
```



Altra Idea !

DFS

```
forEach v in V
    ...
    ...
    ...
    ...
```

**Sarà migliore di BFS oppure no?**

- Su un grafo enorme meglio eseguire questo o BFS?
- Magari per certi G con una forma particolare è più veloce mentre per altri G è più lento.
- **Studio e comparo la complessità** dei due algoritmi

# Attenzione

- Questa parte sugli algoritmi è più difficile, anche perché l'abbiamo trattata molto superficialmente e solo tramite esempi.
- È importante però comprendere i concetti di base, e nel caso su questi ci siano dubbi/domande:
  1. provare a rendere molto precise e puntuali le domande/i dubbi. Anche questo aiuta a migliorare lo studio!
  2. chiedere: ad un compagno di studio, nel forum del corso, al prof.

# Cosa sapere sull'esecuzione di BFS

- capire **cos'è la visita in ampiezza**
- capire **cos'è la distanza e il cammino minimo** di un nodo da un altro

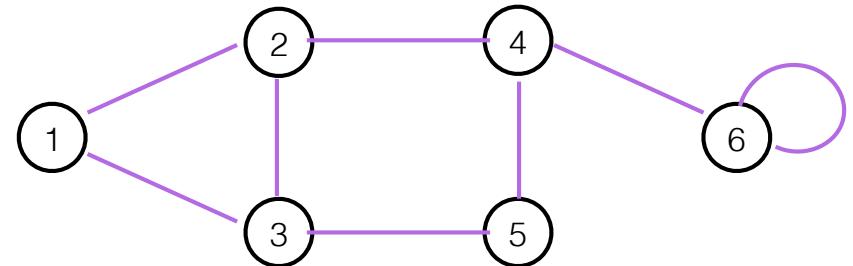
## Esempio di esercizi:

Indichiamo con **BFS( $G, n$ )** l'esecuzione di BFS sul grafo  $G$  a partire dal nodo  $n$ .

- Dato questo elenco di nodi del grafo  $G$  in figura:

**1, 2, 4, 5, 6, 3**

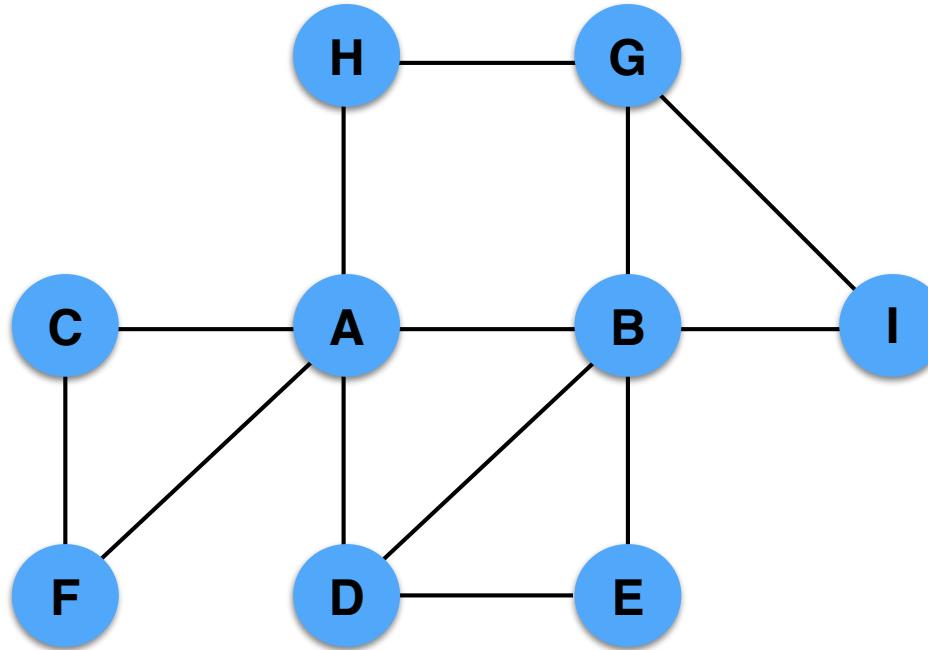
potrebbe essere l'ordine in cui **BFS( $G, 1$ )** visita i nodi del grafo?



- Elencare in che ordine visita i nodi l'esecuzione **BFS( $G, 5$ )**

**N.B.**: può esserci più di una soluzione: **5,4,3,...** ma anche **5,3,4,....**

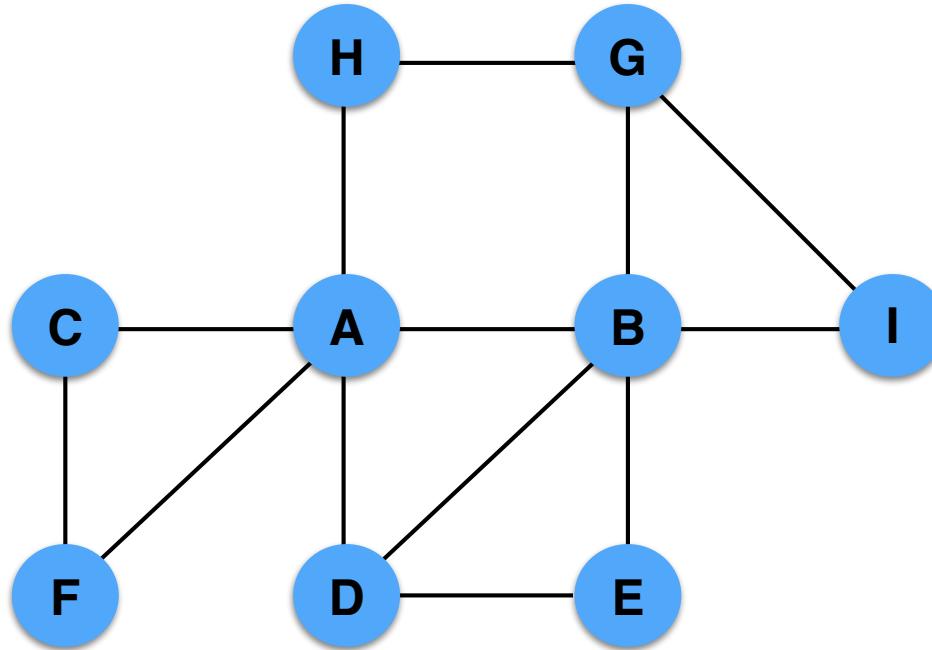
# Esercizio



- Eseguire l'algoritmo Breadth-First Search a partire dal nodo A
- Eseguire l'algoritmo Breadth-First Search a partire dal nodo I

È sufficiente indicare un ordine in cui vengono visitati i nodi, e indicare per ogni nodo le 3 informazioni (colore, padre, distanza) nello stato finale.

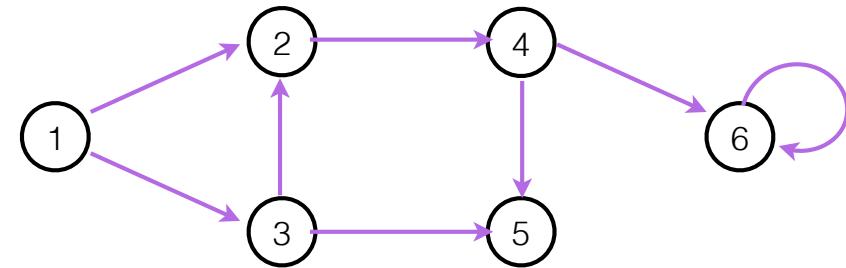
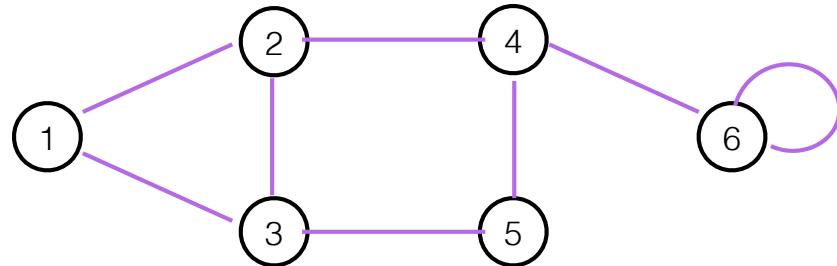
# Esercizio



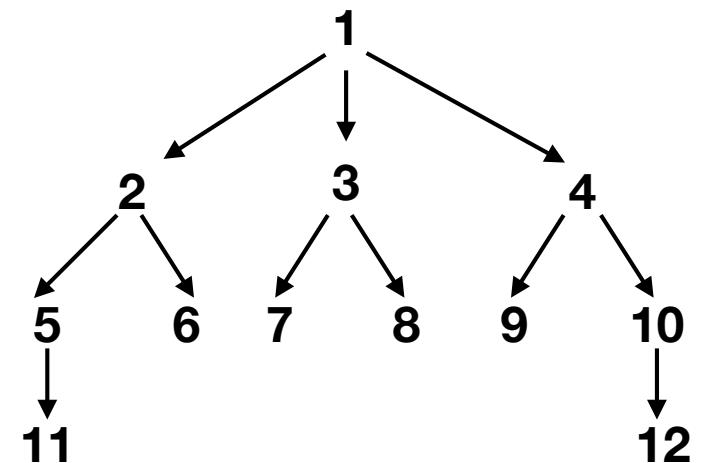
- Indicare un possibile ordine di visita Depth-first a partire dal nodo A
- Indicare un possibile ordine di visita Depth-first a partire dal nodo I

# Esercizio

Eseguire i due algoritmi di visita, BFS e DFS, sui seguenti **4** grafi, indicando semplicemente l'ordine in cui vengono visitati i nodi.



Considerare il grafo  $G$  di vertici= $\{1,2,3,4,5,6\}$  e archi  $\{(1,2),(2,4),(1,4),(3,4),(5,3),(4,5),(5,6),(6,1)\}$



# Algoritmo Breadth-first Search

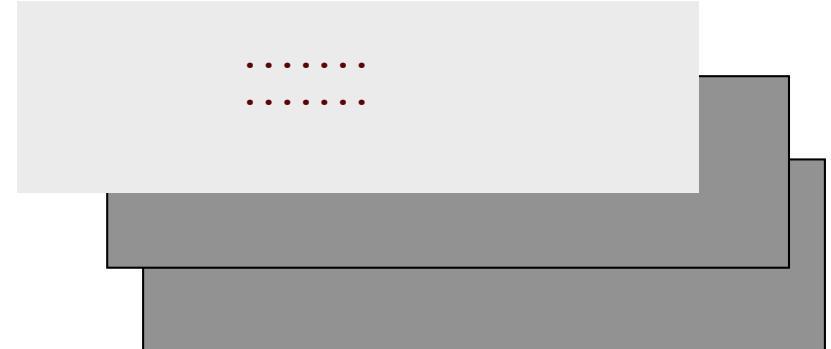
## Algoritmo: BFS

```
forEach v in V
    v.color := bianco
    v.dist := ∞
    v.padre := nil
s.color := grigio
s.dist := 0
Front.add(s)

while Front.isEmpty==false
    u := Front.pop
    forEach v in AdjList(u)
        if v.color == bianco
            v.color := grigio
            v.dist := u.dist + 1
            v.padre := u
            Front.add(v)
    u.color := nero
```

questo è *pseudo-codice*:  
per poterlo eseguire  
va **implementato**  
in un linguaggio di  
programmazione

## Software in Python



## Software in Java



## Software in C++





chiara? corretta?  
completa?

**“Specifica del problema”**



correttezza e  
complessità

**“Algoritmo risolutivo”**



```
int main()
{
    int palindrome, reverse=0;
    cout<<"Enter number: ";
    cin>>palindrome;

    while(palindrome > 0)
    {
        reverse = reverse * 10 + palindrome % 10;
        palindrome = palindrome / 10;
    }

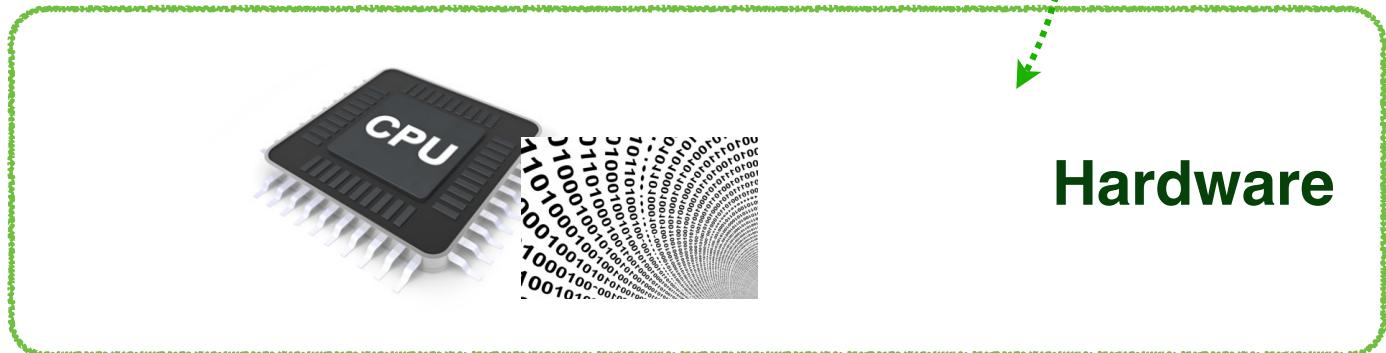
    cout<<(reverse == palindrome ? "The number is a palindrome." : "The number is not a palindrome.");
    return 0;
}
```

**Software**

errori sintattici e  
logici

compilatore

**Sistema Operativo**



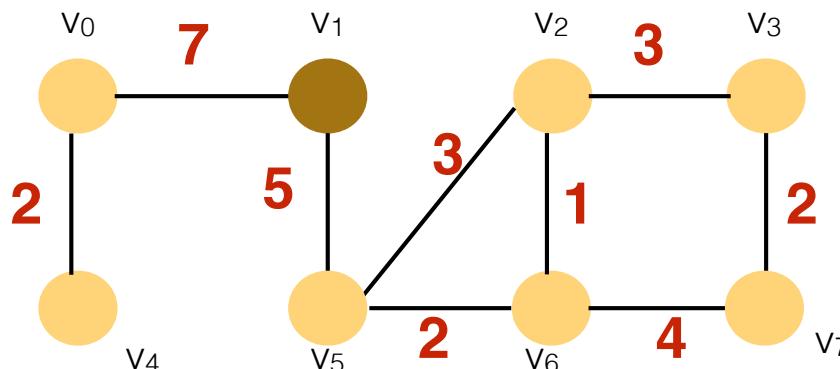
**Hardware**

# Approfondimento NON RICHIESTO per l'esame

## problemi di ottimizzazione

Tanti problemi reali si possono modellizzare come problemi su grafi:

- stoccaggio
- allocazione risorse ( aerei, aule..)
- percorsi di viaggio
- ...

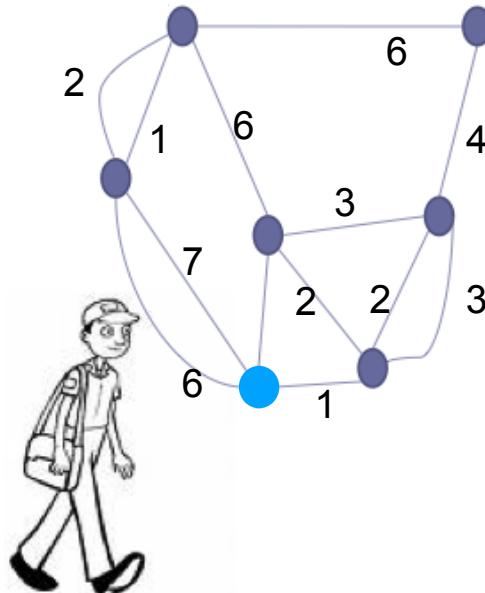


*grafo pesato*

# Problema del postino cinese

Un postino parte dall'ufficio postale, attraversa **tutte le strade** del quartiere e **ritorna** all'ufficio postale, percorrendo la **minima** distanza possibile.

Analogo: trovare il modo per pulire tutte le strade e tornare alla base con costo minimo

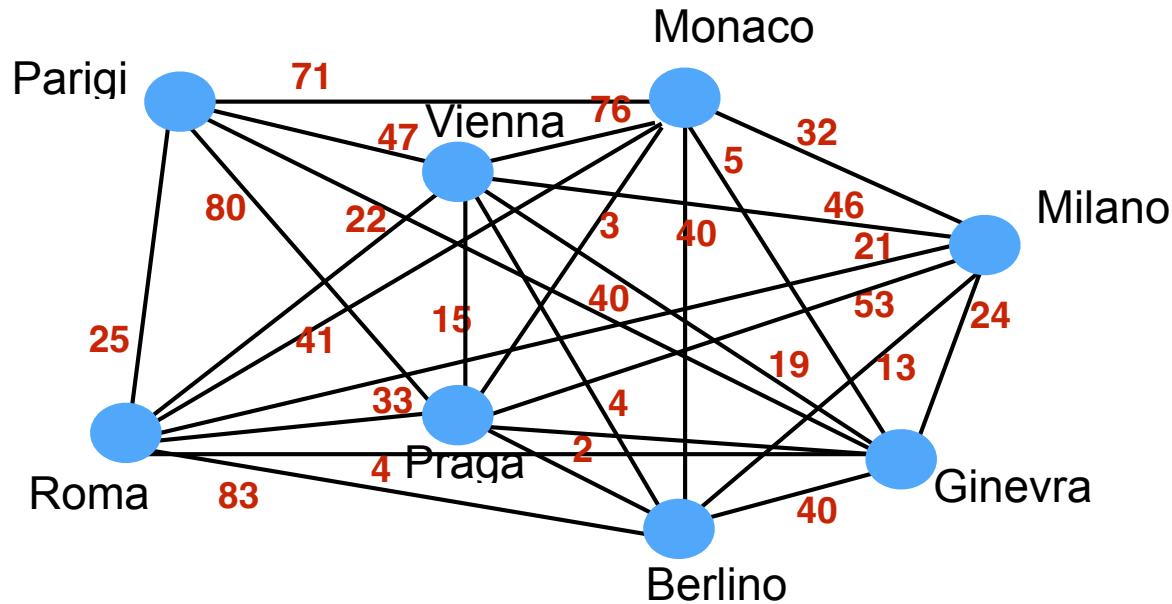


trovare un *ciclo* che passi su tutti gli archi del grafo **almeno** una volta e che sia di *lunghezza minima*

Dato questo problema, **la soluzione è un algoritmo** che, dato in input un qualsiasi grafo pesato  $G$ , fornisce in **output** un elenco di archi di  $G$  con la proprietà voluta

# problema del commesso viaggiatore

TSP: Travelling Salesman Problem

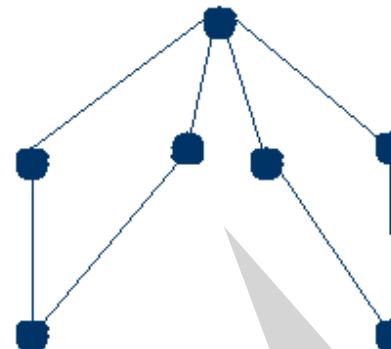
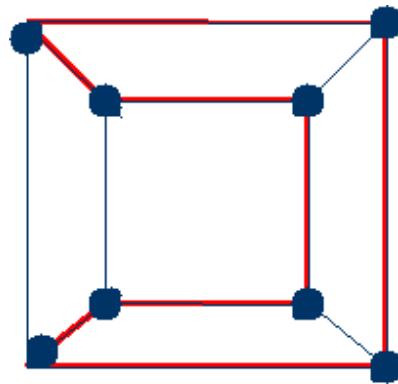


data una rete di città connesse tra loro con strade, **trovare** il percorso di **minore lunghezza** che un commesso viaggiatore deve seguire per visitare tutte le città una e una sola volta **e tornare a casa**

Si cerca **un algoritmo** che, dato in input un qualsiasi grafo pesato  $G$ , fornisce in **output** un elenco di nodi di  $G$  con la proprietà voluta

# problema del commesso viaggiatore

**Semplifichiamo:** togliamo i pesi dagli archi, e cerchiamo un ciclo che passa una e una sola volta per ogni nodo



Quindi non sempre esiste un cammino

**È un problema difficile!**

Più difficile del problema del postino

dovunque decida di partire, qui **non riesce** a visitare tutte le città e tornare a casa senza passare 2 volte per il nodo centrale.

Trovare il percorso del commesso viaggiatore = trovare un *ciclo hamiltoniano* in un grafo

# cicli hamiltoniani e TSP

**Problema 1:** Dato un grafo  $G$ ,  $G$  **ha** un ciclo hamiltoniano?

- è un problema di **decisione**
- **Soluzione/Algoritmo brute-force**: prendo l'elenco di **tutte** le permutazioni dei vertici di  $G$  e controllo per ognuna se è un ciclo in  $G$
- complessità: se ho  $n$  vertici in  $G$ , ci sono  $n!$  permutazioni possibili

**Problema 2:** Dato un grafo  $G$  e un cammino  $c = v_1, \dots, v_n$ ,  $c$  **è** un ciclo hamiltoniano?

- è un problema di **verifica**
- **Soluzione/Algoritmo**: controllo che  $v_1, \dots, v_n$  siano tutti distinti e collegati da un arco in  $G$
- complessità: se ho  $n$  vertici in  $G$ ,  $O(n)$

# cicli hamiltoniani e TSP

**Problema 1:** Dato un grafo G, G **ha**

- è un problema di **decisione**
- **Soluzione/Algoritmo brute-force**: prende l'elenco delle permutazioni dei vertici di G e controlla per ognuna se è un ciclo in G
- complessità: se ho n vertici in G, ci sono **n!** permutazioni possibili

$$5! = 120 \sim 10^2$$

$$10! = 3.628.800 \sim 10^6$$

$$20! = 2.432.902.008.176.640.000 \sim 10^{18}$$

**intrattabile!**

**Problema 2:** Dato un grafo G e un cammino  $c = v_1, \dots, v_n$ ,  
c è un ciclo hamiltoniano?

- è un problema di **verifica**
- **Soluzione/Algoritmo**: controllo che  $v_1, \dots, v_n$  siano tutti distinti e collegati da un arco in G
- complessità: se ho n vertici in G, O(n)

# Classi di Complessità

P

problemi che possono essere  
**decisi** con un algoritmo  
“veloce” (polinomiale)

NP

problemi che possono essere  
**verificati** con un algoritmo  
“veloce” (polinomiale)

Problema del postino cinese

TSP: Travelling Salesman Problem

una “decisione veloce” del TSP  
non è stata **ancora** trovata

# Classi di Complessità

P

problemi che possono essere  
**decisi** con un algoritmo  
“veloce” (polinomiale)

NP

problemi che possono essere  
**verificati** con un algoritmo  
“veloce” (polinomiale)

P = NP  
?

una “decisione veloce” del TSP  
**si potrà mai trovare?**

Se si dimostra che non esiste  
allora P != NP

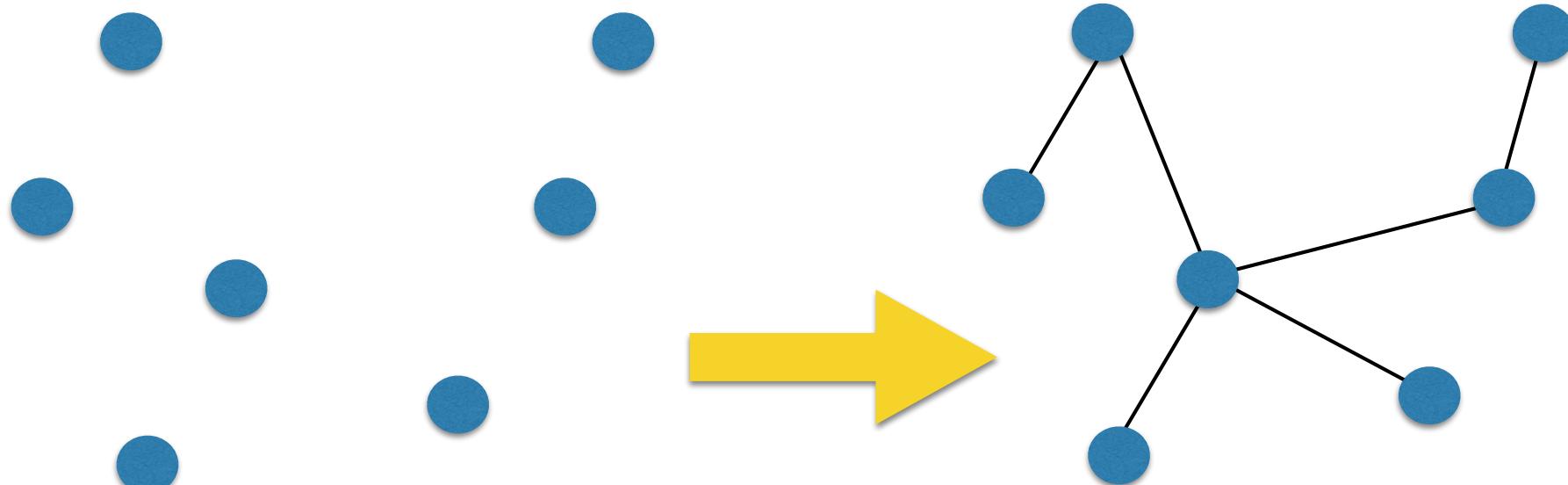


Clay Mathematics Institute

# connector problem

Design a railway network connecting a number of cities, with a minimum possible construction cost.

Or electrical network, cable TV, gas, etc.



# connector problem

la muffa (unicellulare)  
**Physarium Polycephalum**

reti comparabili per

- costo (lunghezza archi)
- efficienza (cammini minimi)
- resilienza (ridondanza analoga)

ma la muffa ha costruito la sua rete  
in **modo distribuito e senza alcun  
coordinamento globale!**

- *self-organization*
- *self-optimization*
- *self-repair*

