Corso di laurea in Informatica
Progettazione d'algoritmi
Didattica blended

Backtracking II

Angelo Monti

SAPIENZA
UNIVERSITÀ DI ROMA

ESERCIZIO 1

Progettare un algoritmo che prende come parametro l'intero n e stampa tutte le permutazioni dei numeri da 0 a $n-1$.

Ad esempio per $n = 4$ bisogna stampare $4! = 24$ permutazioni:

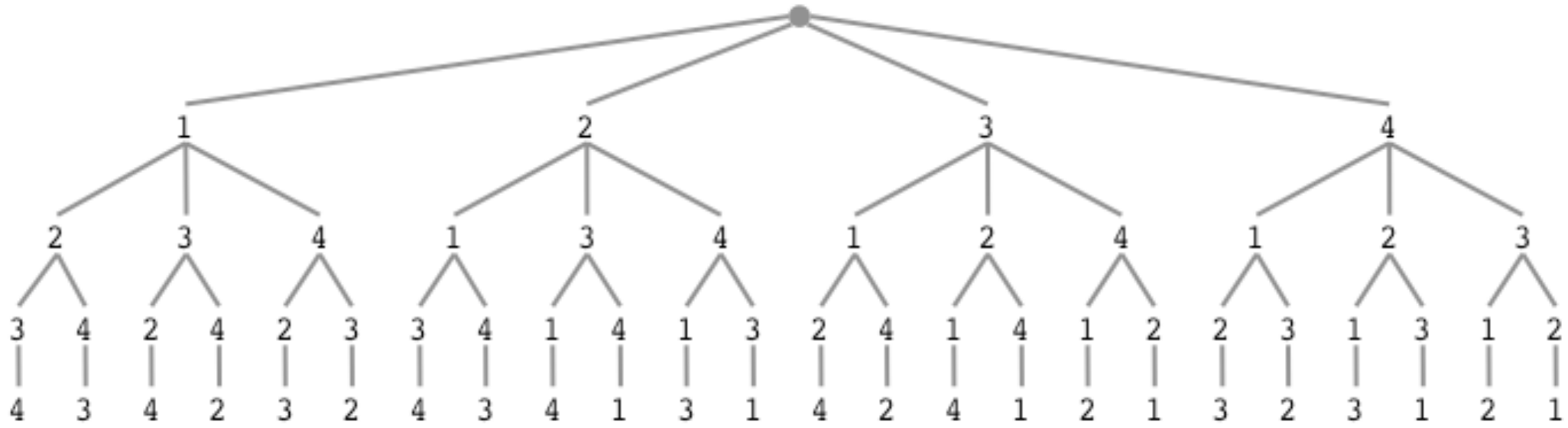
$[0,1,2,3]$ $[0,1,3,2]$ $[0,2,1,3]$ $[0,2,3,1]$ $[0,3,1,2]$ $[0,3,2,1]$ $[1,0,2,3]$

$[1,0,3,2]$ $[1,2,0,3]$ $[1,2,3,0]$ $[1,3,0,2]$ $[1,3,2,0]$ $[2,0,1,3]$ $[2,0,3,1]$

$[2,1,0,3]$ $[2,1,3,0]$ $[2,3,0,1]$ $[2,3,1,0]$ $[3,0,1,2]$ $[3,0,2,1]$ $[3,1,0,2]$

$[3,1,2,0]$ $[3,2,0,1]$ $[3,2,1,0]$

Albero delle permutazioni $(1, 2, 3, 4)$



- L'albero delle permutazioni ha $\Theta(n!)$ foglie e $\Theta(n!)$ nodi interni.

  - nodi interni$= \sum_{i=0}^{n} \frac{n!}{i!} < n! \cdot \sum_{i=0}^{\infty} \frac{1}{i!} \leq n! \cdot \sum_{i=0}^{\infty} \frac{2}{2^i} = n! \frac{2}{1-\frac{1}{2}} = 4 \cdot n!$
  - ho usato
    1. $i! \geq \frac{2^i}{2}$
    2. $\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$ per $x < 1$

Algoritmo esaustivo:

```
def perm(n):
    presi=[0 for i in range(n)]
    perm1(n, 0, [], presi)


def perm1(n,i,sol,presi):
    if i==n:
        print(sol)
    else:
        for j in range(n):
            if not presi[j]:
                sol.append(j)
                presi[j]=1
                perm1(n,i+1,sol,presi)
                sol.pop()
                presi[j]=0

>>> perm(3)
[0, 1, 2]
[0, 2, 1]
[1, 0, 2]
[1, 2, 0]
[2, 0, 1]
[2, 1, 0]
```

Considerazioni sulla complessità

- L'algoritmo ha complessità $\Theta(n! \cdot n)$ = $2^{\Theta(n \log n)}$
  - l'albero delle permutazioni ha $\Theta(n!)$ nodi interni e $n!$ foglie
  - ciascun nodo interno richiede tempo $\Theta(n)$ e ciascuna foglia richiede tempo $\Theta(n)$.
- Qualunque algoritmo per questo problema richiede tempo $\Omega\,(n! \cdot n)$
  - le soluzioni da stampare sono $n!$ e il tempo per stampare ciascuna di queste è $\Theta(n)$

L'algoritmo proposto è ottimo.

ESERCIZIO 2

Progettare un algoritmo che prende come parametro l'intero n e stampa tutte le permutazioni dei numeri da 0 a $n-1$ dove nelle posizioni pari compaiono numeri pari.
La complessità dell'algoritmo deve essere $O\left(S(n) \cdot n^2\right)$ dove $S(n)$ è il numero di permutazioni da stampare.

Ad esempio per $n = 5$ delle $5! = 120$ permutazioni bisogna stampare le seguenti 12:

$[0, 1, 2, 3, 4]$ $[0, 1, 4, 3, 2]$ $[0, 3, 2, 1, 4]$ $[0, 3, 4, 1, 2]$

$[2, 1, 0, 3, 4]$ $[2, 1, 4, 3, 0]$ $[2, 3, 0, 1, 4]$ $[2, 3, 4, 1, 0]$

$[4, 1, 0, 3, 2]$ $[4, 1, 2, 3, 0]$ $[4, 3, 0, 1, 2]$ $[4, 3, 2, 1, 0]$

Algoritmo di backtraking:

```
def perm(n):
    presi=[0 for i in range(n)]
    perm1(n, 0, [], presi)


def perm1(n, i, sol, presi):
    if i==n:
        print(sol)
    else:
        for j in range(n):
            if not presi[j] and j%2 == i%2:
                sol.append(j)
                presi[j]=1
                perm1(n,i+1, sol, presi)
                sol.pop()
                presi[j]=0
```

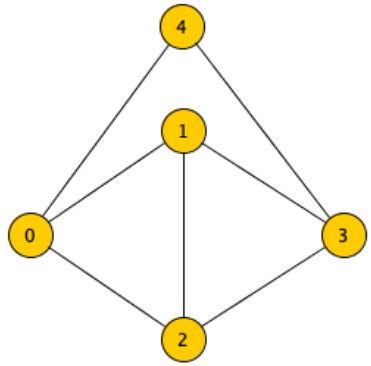
Sia $S(n)$ il numero di permutazioni da stampare.

- L'algoritmo ha complessità $O\left(S(n) \cdot n^2\right)$
  - l'albero di ricorsione è di altezza n.
  - solo i nodi che portano ad una delle $S(n)$ soluzioni vengono effettivamente generati
  - i nodi interni effettivamente generati saranno $O(S(n) \cdot n)$ e le foglie effettivamente generate saranno $S(n)$
  - ciascun nodo interno richiede tempo $\Theta(n)$ e ciascuna foglia richiede tempo $\Theta(n)$
  - il tempo in totale sarà $O\left(S(n) \cdot n\right) \cdot \Theta(n) + S(n) \cdot \Theta(n) = O\left(S(n) \cdot n^2\right)$

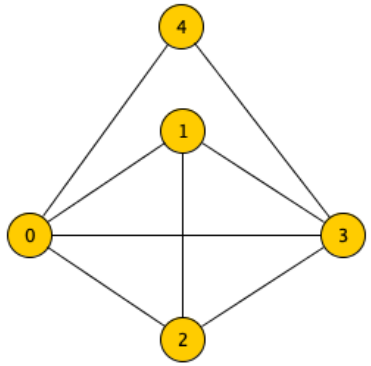
Il problema della 3-colorazione.

Per un grafo (non diretto) G, una 3-colorazione lecita è una 3-colorazione sol degli n nodi di G tale che, per ogni arco $\{i, j\}$ di G, $sol[i] \neq sol[j]$.
Dato un grafo non diretto G.

Progettare un algoritmo che prende come parametro un grafo G e determina se il grafo ammette una 3-colorazione sol o meno.
Più precisamente: se il grafo è 3-colorabile l'algoritmo restituisce una possibile 3-colorazione sol, in caso contrario non restituisce nulla.



G1



G2

Ad esempio utilizzando i tre colori $'b'$, $'r'$ e $'v'$:

- per il grafo 3-colorabile

$$G1 = \{0 : [1, 2, 4], 1 : [0, 2, 3], 2 : [0, 1, 3], 3 : [1, 2, 4], 4 : [0, 3]\}$$

  l'algoritmo restituisce una delle seguenti possibili $3-$colorazioni:

  $['r',' v',' b',' r',' v']$ $['r',' v',' b',' r',' b']$ $['r',' b',' v',' r',' v']$ $['b',' r',' v',' b',' v']$

  $['r',' b',' v',' r',' b']$ $['v',' r',' b',' v',' r']$ $['v',' r',' b',' v',' b']$ $['b',' v',' r',' b',' r']$

  $['v',' b',' r',' v',' r']$ $['v',' b',' r',' v',' b']$ $['b',' r',' v',' b',' r']$ $['b',' v',' r',' b',' v']$

- per il grafo

$$G2 = \{0 : [1, 2, 3, 4], 1 : [0, 2, 3], 2 : [0, 1, 3], 3 : [0, 1, 2, 4], 4 : [0, 3]\}$$

  che non è 3-colorabile l'algoritmo non restituisce nulla.

Idea dell'algoritmo basato su backtraking:

- considero i vertici uno dopo l'altro, e ad ogni vertice assegno un colore che non vada in contrasto con vertici adiacenti già colorati.
- Se arrivo a colorare l'ultimo vertice allora il grafo è $3-$colorabile altrimenti, se non ho colore da assegnare ad un vertice (ha tre suoi vicini gia colorati con i tre diversi colori) riconsidero l'ultimo vertice colorato e cerco per lui un colore alternativo.

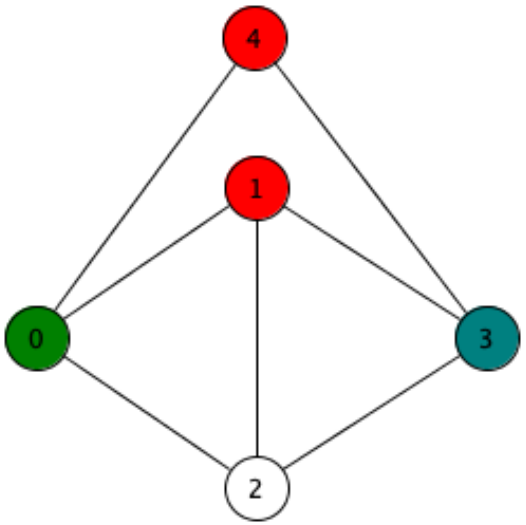
nota che l'eventuale soluzione è una stringa ternaria i cui elementi sono i tre simboli $'b'$, $'r'$ e $'v'$.

Algoritmo di backtracking che restituisce una eventuale 3-colorazione del grafo.

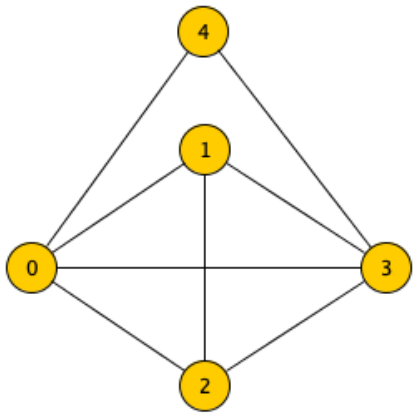
```
def col3(G):
    return col3R(G,0,[])


def col3R(G,i,sol):
    if i==len(G):
        return sol
    else:
        C=set()
        for j in G[i]:
            if j<i: C.add(sol[j])
        C={'b','r', 'v'}-C
        for c in C:
            sol.append(c)
            if col3R(G,i+1,sol): return sol
            sol.pop()
    return None
```

```
>>>G1={
0:[1,2,4],
1:[0,2,3],
2:[0,1,3],
3:[1,2,4],
4:[0,3]
}
>>> col3(G1)
['v', 'r', 'b', 'v', 'r']
```




```
>>>G2={
0:[1,2,3],
1:[0,2,3],
2:[0,1,3],
3:[0,1,2,4],
4:[0,3]
}
>>> col3(G2)
```



Complessità dell'algoritmo proposto:

La funzione di taglio introdotta (vale a dire la scelta di non colorare un vertice intermedio con un colore uguale a quello di un vertice adiacente già colorato) non garantisce che quella colorazione porterà ad una foglia che rappresenta una colorazione. La complessità quindi non sarà necessariamente proporzionale al numero di 3-colorazioni esistenti.

la Complessità dell'algoritmo è $O\left(3^n \cdot n\right)$

- l'albero di ricorsione è un albero ternario di altezza n che ha dunque al più $\frac{3^n-1}{3-1} = \Theta(3^n)$ nodi (ricorda infatti che un albero t-ario completo, con $t > 1$, ha esattamente $\sum_{i=0}^{n} t^i = \frac{t^n-1}{t-1}$ nodi)
- ogni nodo interno richiede tempo $O(n)$ mentre al più una foglia verrà raggiunta e richiederà tempo $O(1)$.
- il tempo totale è $O\left(3^n \cdot n\right)$

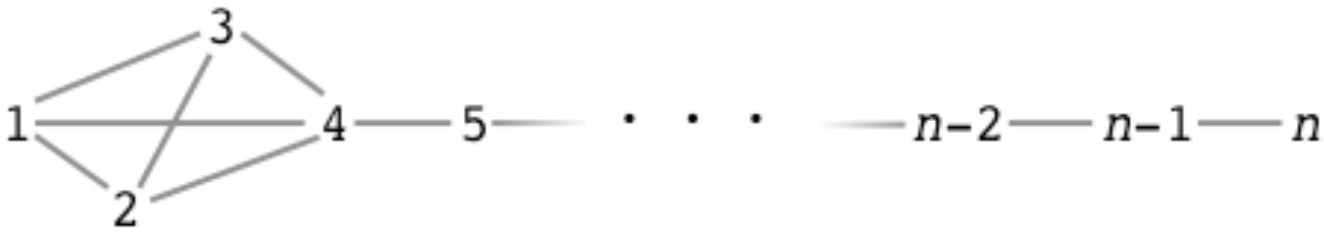
- Dare una buona stima della complessità è piuttosto difficile proprio perché dipende fortemente dal grafo di input.
- Possiamo però vedere come l'algoritmo si comporta su alcuni input.

- Consideriamo il grafo G mostrato nella figura qui sotto (con la relativa numerazione dei nodi):

- È chiaro che G non è 3-colorabile ma il programma *col*3 quanto tempo impiegherà per accorgersene?

- Per i primi $n-3$ nodi l'algoritmo trova sempre due colori leciti (per il nodo 1 ne trova tre) quindi la ricorsione genera almeno un albero binario completo di profondità $n-3$. Un tale albero contiene almeno 2^{n-3} nodi e dunque la complessità sarà almeno pari a $\Omega(2^n)$ che è esponenziale in n.

- Modifichiamo ora la numerazione dei nodi del grafo G come in figura:

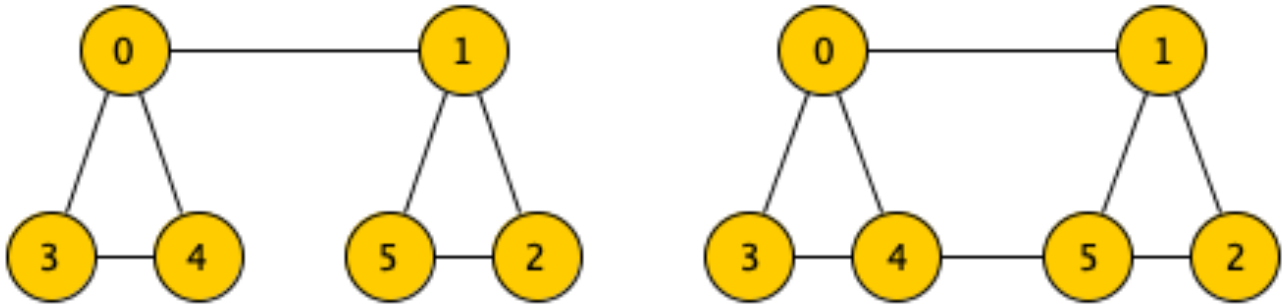


- Quanto tempo impiegherà ora l'algoritmo per accorgersi che G non ha una 3-colorazione lecita?
  La complessità si riduce enormemente perché quando l'algoritmo arriva al nodo 4 non trova mai colori leciti per cui la complessità è pari ad $O(1)$.

- Quindi la complessità del programma *col*3 non solo dipende dal grafo ma può dipendere fortemente anche dalla numerazione dei nodi.

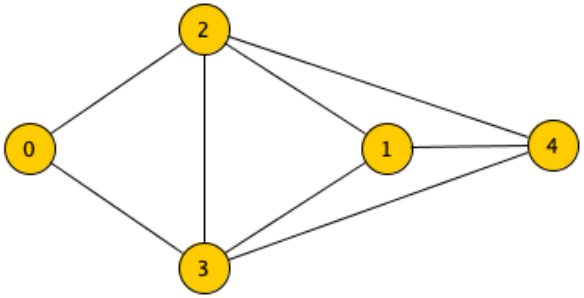
In un grafo un ciclo si dice *Hamiltoniano* se tocca tutti i nodi del grafo.

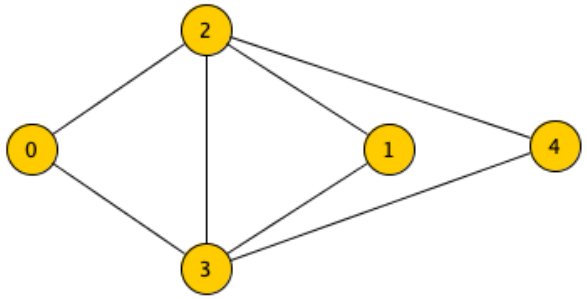
Il problema del ciclo Hamiltoniano: Dato un grafo G voglimo sapere se G ha o meno un ciclo Hamiltoniano



Il grafo a sinistra non ha ciclo hamiltoniano, il grafo di destra si.

Progettare un algoritmo che preso in input il grafo G restituisce un ciclo hamiltoniano se nel grafo è presente, nulla altrimenti.

$G1 =$ 

$G2 =$ 

- Il grafo

$$G1 = \{0 : [2, 3], 1 : [2, 3, 4], 2 : [0, 1, 3, 4], 3 : [0, 1, 2, 4], 4 : [1, 2, 3]\}$$

l'algoritmo restituisce uno dei seguenti possibili cicli hamiltoniani

$(0, 2, 1, 4, 3)$ $(0, 2, 4, 1, 3)$ $(0, 3, 1, 4, 2)$ $(0, 3, 4, 1, 2)$

- Il grafo

$$G2 = \{0 : [2, 3], 1 : [2, 3], 2 : [0, 1, 3, 4], 3 : [0, 1, 2, 4], 4 : [2, 3]\}$$

non ha cicli hamiltoniani e l'algoritmo non restituisce nulla.

Nota che:

- ogni nodo del grafo viene toccato una sola volta dal ciclo quindi un ciclo hamiltoniano è una permutazione $perm$ degli n vertici del grafo con la proprietà che esistono nel grafo tutti gli archi:

$$(perm[0], perm[1]), (perm[1], perm[2]), \ldots, (perm[n-2], perm[n-1]), (perm[n-1], perm[0]).$$

- per risolvere il problema possiamo quindi andare alla ricerca di una tale permutazione.

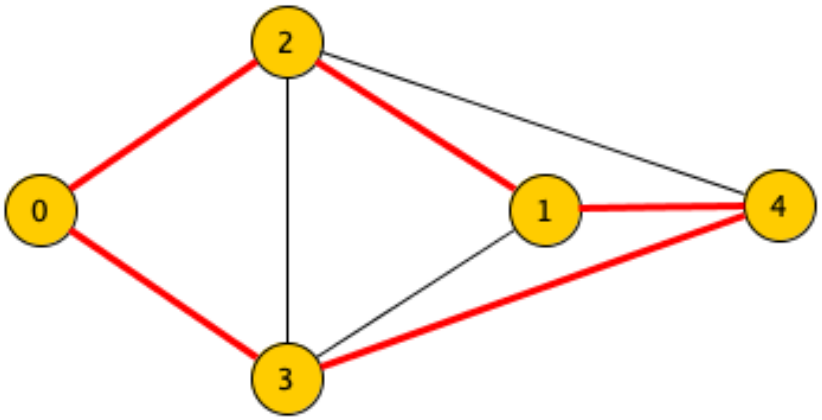
- Per potare lo spazio di ricerca nell'albero delle permutazioni possiamo
  - imporre che la permutazione inizi con il vertice 0
  - imporre che i prefissi della permutazione rappresentino cammini del grafo.

Algoritmo di backtracking per la ricerca del ciclo hamiltoniano

```
def ham(G):
    presi=[0 for v in range(len(G))]
    presi[0]=1
    return hamR(G,1,[0],presi)

def hamR(G,i,sol,presi):
    if i==len(G):
        if 0 in G[sol[-1]]: return sol
        else: return None
    for j in G[sol[-1]]:
        if not presi[j]:
            sol.append(j)
            presi[j]=1
            if hamR(G,i+1,sol,presi): return sol
            presi[j]=0
            sol.pop()
    return None
```

```
>>>G1={
0:[2,3],
1:[2,3,4],
2:[0,1,3,4],
3:[0,1,2,4],
4:[1,2,3]
}
>>> ham(G1)
[0, 2, 1, 4, 3]
```



Complessità dell'algoritmo proposto:

La funzione di taglio introdotta (vale a dire la scelta di non generare solo permutazioni che cominciano con 0 e i cui prefissi siano cammini del grafo) non garantisce che i nodi generati porteranno ad una foglia che rappresenta un ciclo hamiltoniano. La complessità quindi non sarà necessariamente proporzionale al numero di cicli hamiltoniani esistenti.

la Complessità dell'algoritmo è $O(n!)$

- l'albero di ricorsione è un albero delle permutazioni di altezza $n-1$ che ha dunque al $O((n-1)!)$
- ogni nodo interno richiede tempo $\Theta(n)$ mentre al più $(n-1)!$ foglie verranno raggiunte e ciascuna richiederà tempo $O(1)$.
- il tempo totale è $O((n-1)! \cdot n) = O(n!)$

- La complessità del programma dipende fortemente dal grafo di input.
- Ci sono grafi per i quali trova subito un ciclo Hamiltoniano (ad es. se il grafo è un ciclo) o determina che non ce ne sono (ad es. se il grafo è un albero).
- Ma ci sono tantissimi grafi per i quali impiega tempo esponenziale per dare una risposta. Basti pensare che se il grafo G non ha cicli Hamiltoniani, tutti i cammini dal nodo 1 saranno comunque visitati e se questi sono tanti la complessità sarà molto elevata.
- si consideri ad esempio il seguente grafo:

- Il grafo è la concatenazione di $k \geq 1$ cicli ognuno di 4 nodi, ha dunque un totale di $n = 3k + 1$ nodi.
- Ogni ciclo può essere attraversato in due modi diversi quindi il numero di cammini dal nodo 1 al nodo n è 2^k e questi saranno tutti esaminati dal programma ham.
- Quindi il programma impiegherà $\Omega\left(2^{\frac{n}{3}}\right)$ tempo per determinare che il grafo non ha cicli Hamiltioniani.