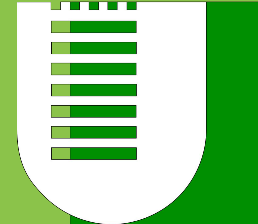
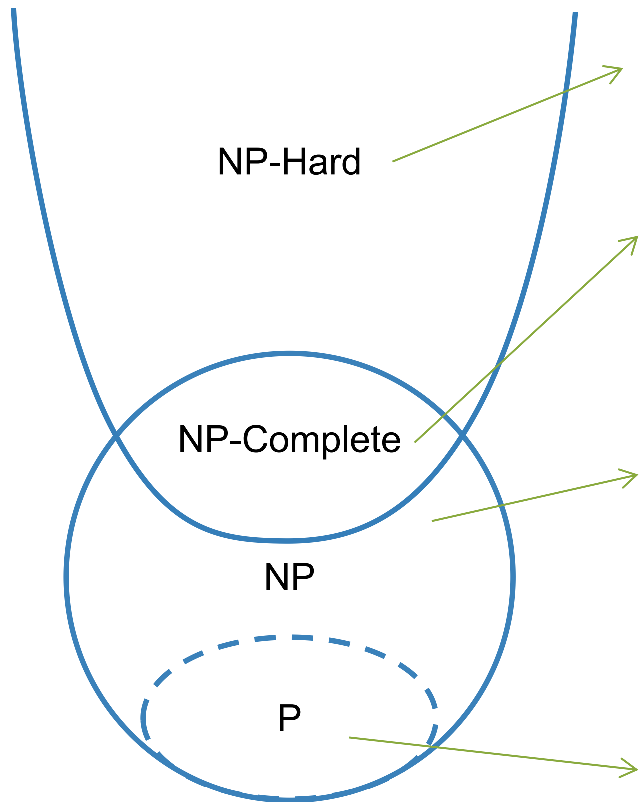


Soluzioni Challenge 2019/2020

Alessandro Pellegrini
pellegrini@diag.uniroma1.it

Classi di complessità



Problemi almeno difficili come i più difficili problemi in NP
Esempio: problema della sottosomma a zero

I più difficili problemi della classe NP
Esempio: Problema del Commesso Viaggiatore [$O(n!)$]

Problemi risolvibili da una macchina non deterministica in tempo polinomiale
Esempio: Fattorizzazione di un numero
Si veda: Kleinjung *et al.*, Factorization of a 768-bit RSA modulus, 2010.

Problemi risolvibili da una macchina deterministica in tempo polinomiale
Esempio: programmazione lineare
Tempo di esecuzione: $O(f(n))$

Hexadoku

Sudoku

- Il sudoku è un problema NP-completo
- I problemi NP-completi per cui esistono algoritmi (approssimati o randomizzati) di soluzione non sono molti
- È possibile applicare una trasformazione da un problema ad un altro, per sfruttare un'implementazione differente

- Aspetti di base:
  - n^2 simboli
  - griglia $n^2 \times n^2$
  - n^2 sottogriglie, ciascuna $n \times n$

Approccio naïve: bruteforce

```
solve(grid):
    for i in range(16):
        for j in range(16):
            if grid[i][j] != '':
                continue
            for numero in range(16):
                grid[i][j] = numero
                if solve(grid): return True
                if verify(grid): return True
    return False
```

Problemi

- L'algoritmo in questione è approssimabile a una DFS
- Vengono testate ripetutamente soluzioni inammissibili

- In ogni caso, le griglie possibili sono:
  - Sudoku 9x9: circa $6{,}67 \cdot 10^{21}$
  - Sudoku 16x16: circa $5{,}96 \cdot 10^{98}$ (stimato)

- È una forma di *esplosione combinatoria*

Backtracking

```
def solve(grid):
    for i in range(0,16):
        for j in range(0, 16):
            if grid[i][j] != '':
                continue
            choices = tries(grid, i, j)
            if len(choices) == 0: return False
            for numero in choices:
                tmp = grid[i][j]
                grid[i][j] = numero
                if solve(grid): return True
                if verify(grid): return True
                grid[i][j] = tmp
            return False
```

Problemi

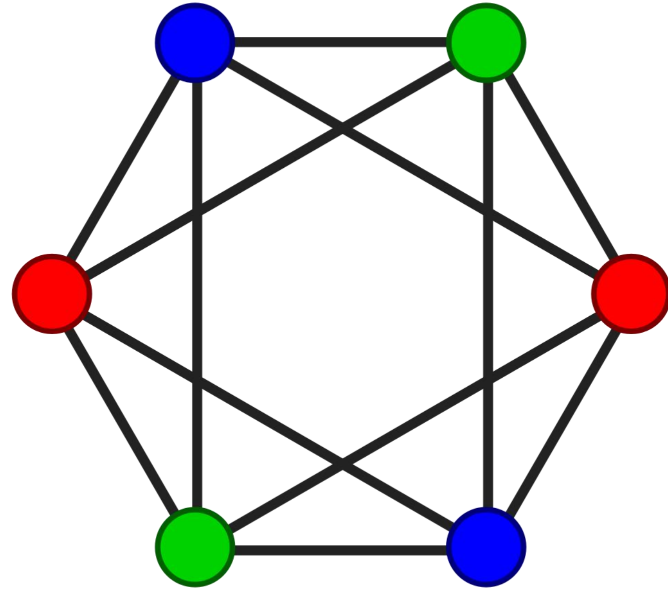
- Lo spazio di ricerca è ancora molto vasto, ancodra DFS
- Alcuni input sono incredibilmente sfavorevoli
  - Un Sudoku con soluzione 987654321 nella prima riga e con pochi suggerimenti richiede di esplorare molte soluzioni

5	3	1	2	7	6	8	9	4
6	2	4	1	9	5	2		
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

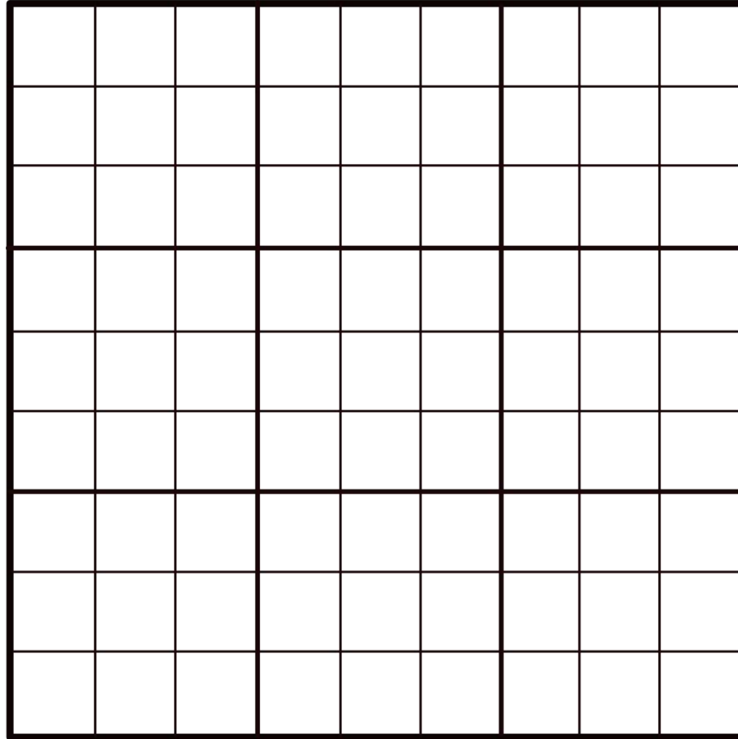
					3		8	5
		1		2				
			5		7			
		4				1		
	9							
5							7	3
		2		1				
				4				9

Graph Coloring

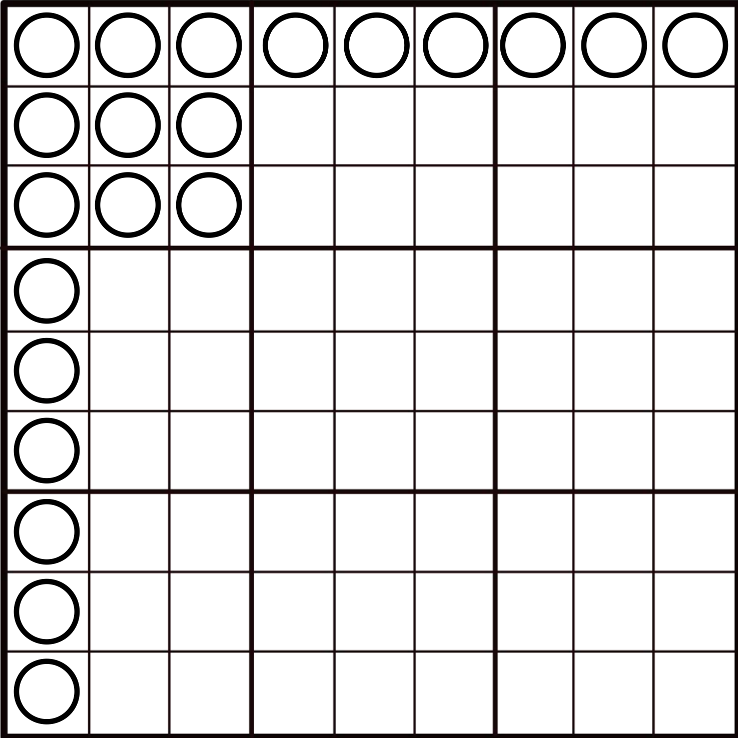
- Dato un grafo, si vuole individuare l'insieme minimo di colori che permette di colorare i nodi in maniera tale che due nodi adiacenti non abbiano lo stesso colore
- Formulazione alternativa (k-coloring): trovare una colorazione dati k colori



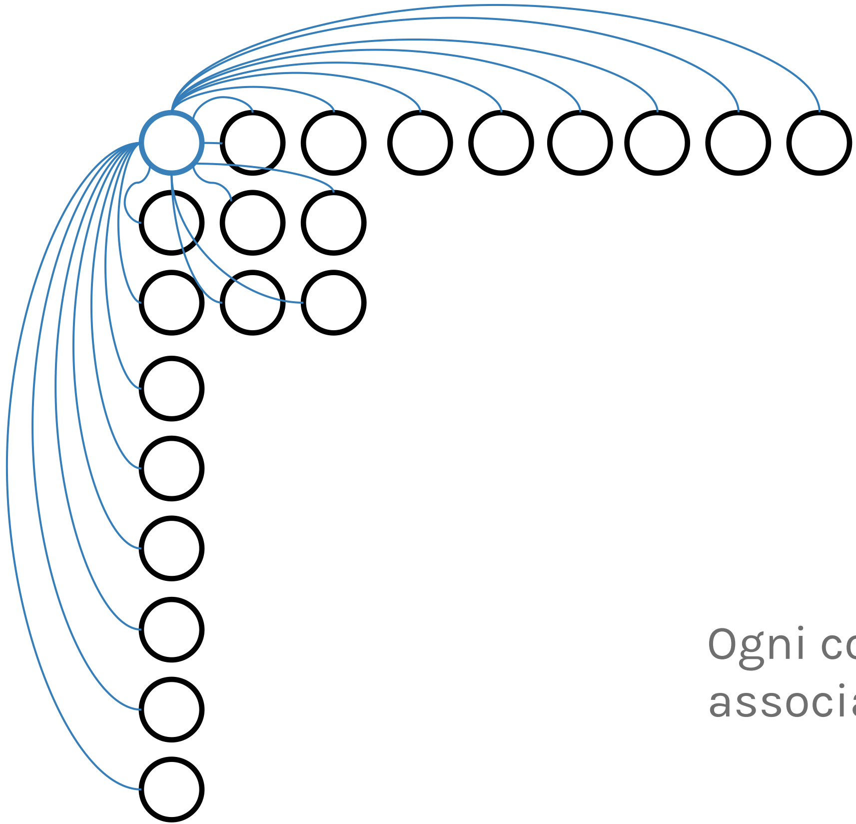
Graph Coloring vs Sudoku



Graph Coloring vs Sudoku



Graph Coloring vs Sudoku



Ogni colore viene associato ad un numero

Algoritmo di Welsh-Powell

```
WelshPowell(G):
    Sequence NodeList ← nodi non colorati in ordine di grado
    while NodeList is not empty:
        node ← NodeList.head()
        color ← trova il colore non utilizzato da alcun nodo adiacente
        node.color ← color
```

Exact Cover

- Input: matrice binaria
- Output: sottoinsieme delle righe
- Condizione: la somma delle righe è il vettore unitario

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}$$

- Si possono vedere le colonne come elementi di un universo
- Vogliamo “coprire” l'intero universo con insiemi disgiunti

Exact Cover

- Input: matrice binaria
- Output: sottoinsieme delle righe
- Condizione: la somma delle righe è il vettore unitario

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}$$

- Si possono vedere le colonne come elementi di un universo
- Vogliamo “coprire” l'intero universo con insiemi disgiunti

Sudoku vs Exact Cover

- Le righe della matrice sono per simbolo, per posizione nella griglia $(n^2 \times (n^2 \times n^2) = n^6)$
- Le colonne sono le condizioni: $(3n^4$ in totale):
  - Per simbolo, per riga della griglia: simbolo nella riga $(n^2 \times n^2)$
  - Per simbolo, per colonna della griglia: simbolo nella colonna $(n^2 \times n^2)$
  - Per simbolo, per regione: simbolo nella regione $(n^2 \times n^2)$

- Si inserisce un 1 nella matrice solo laddove la riga della matrice indica che il posizionamento del simbolo soddisfa le condizioni della matrice

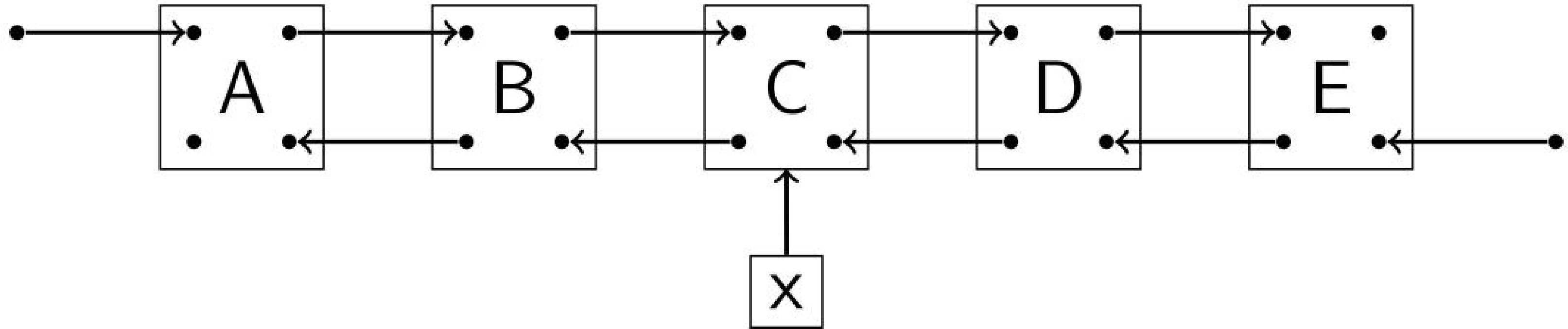
Sudoku vs Exact Cover

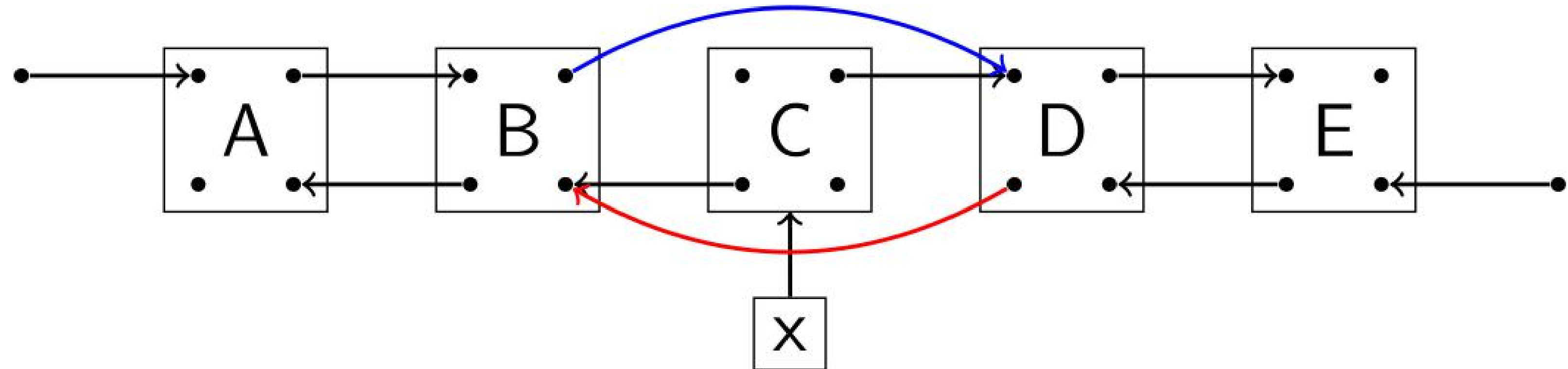
- Esempio: inseriamo un 7 nella griglia alla riga 4, colonna 9

- Si costruisce una riga tale che:
  - Si ha un 1 nella colonna “7 nella griglia alla riga 4”
  - Si ha un 1 nella colonna “7 nella griglia alla colonna 9”
  - Si ha un 1 nella colonna “7 nella griglia nella regione 6”
  - 0 in tutte le altre posizioni

- Un puzzle è un insieme di righe “preselezionate”
- Si possono eliminare queste righe, e le “colonne coperte”
- Le righe selezionate descrivono la soluzione al problema

Dancing Links





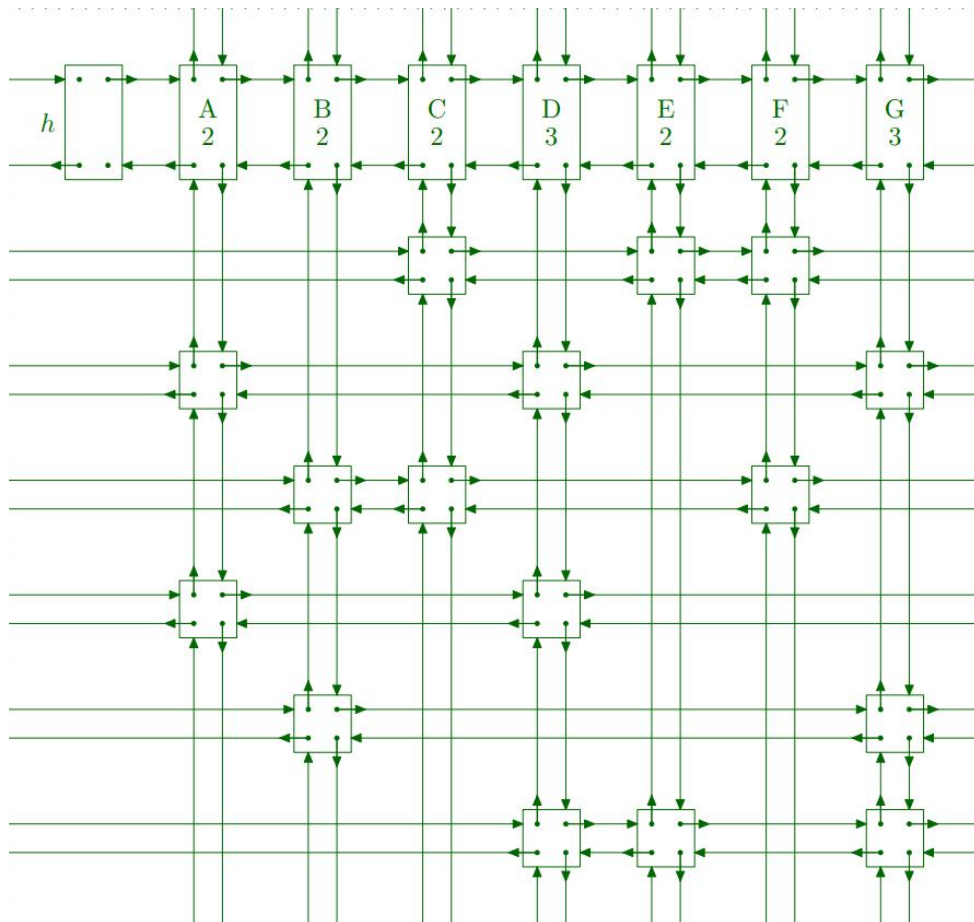
Algorithm X—Knuth, 2000

- Algoritmo DLX (**D**ancing **L**inks - Algorithm **X**)
  - Backtrack sulle colonne
  - Si sceglie una colonna da coprire, questa indicherà la selezione delle righe
  - Si itera sulle righe, per ciascuna riga si rimuovono le colonne coperte
  - Si analizzano sottomatrici in maniera ricorsiva
  - Se non si trova una soluzione valida, si ripristinano le colonne nel passo di backtrack

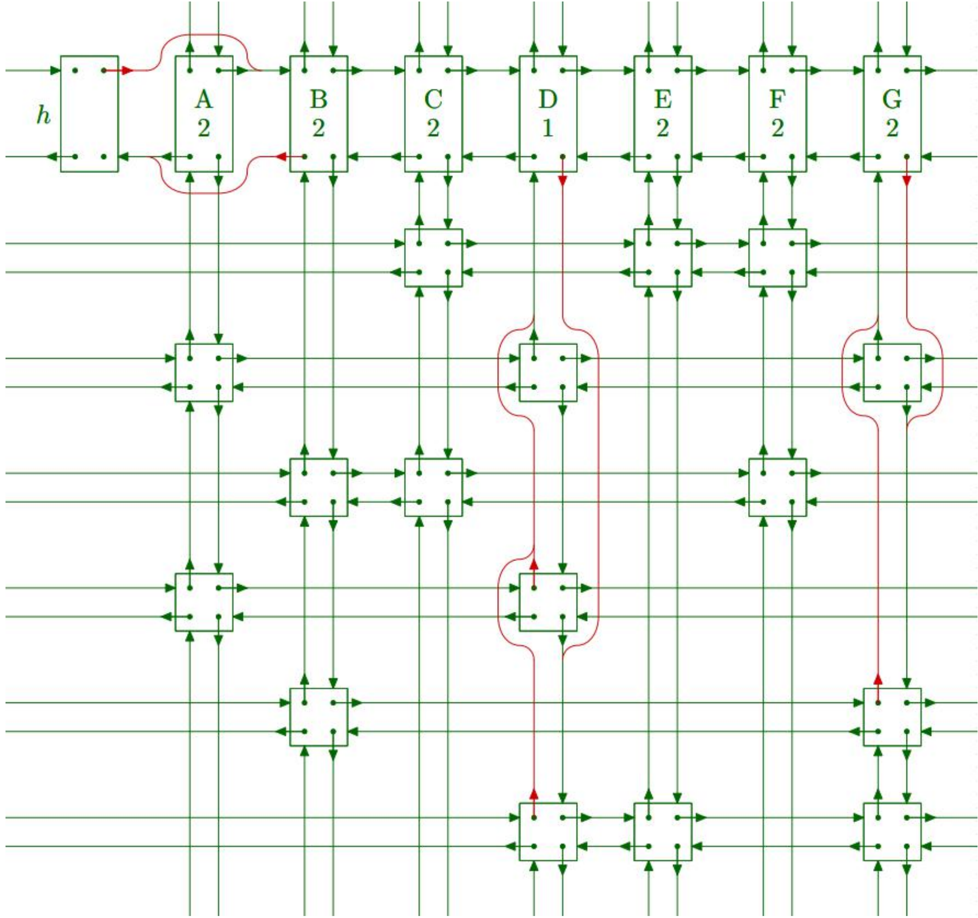
Algorithm X—Knuth, 2000

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}$$

Algorithm X—Knuth, 2000



Algorithm X—Knuth, 2000



Query Bidimensionali

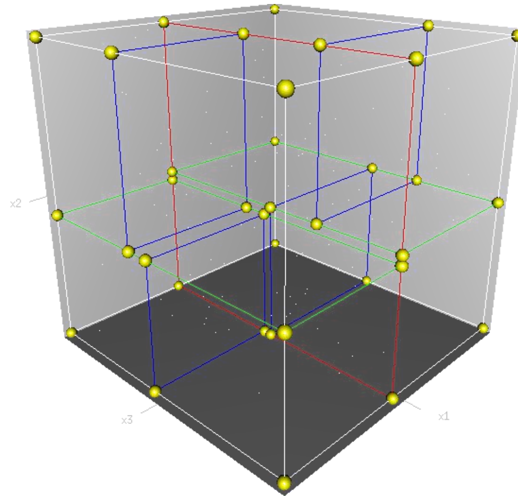
Approccio naïve

- Lista di punti, con scansione e contatore
- Ogni volta che viene effettuata una query si scandisce la lista
- Si accumula in una variabile il numero di punti che cadono nella query
  - Si tengono in considerazione anche le occorrenze ripetute, memorizzate nella lista

```
class Punto:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.count = 1
```

k-d Trees

- Si tratta di una struttura dati utilizzata per partizionare lo spazio
- Molto utilizzate nelle ricerche multidimensionali

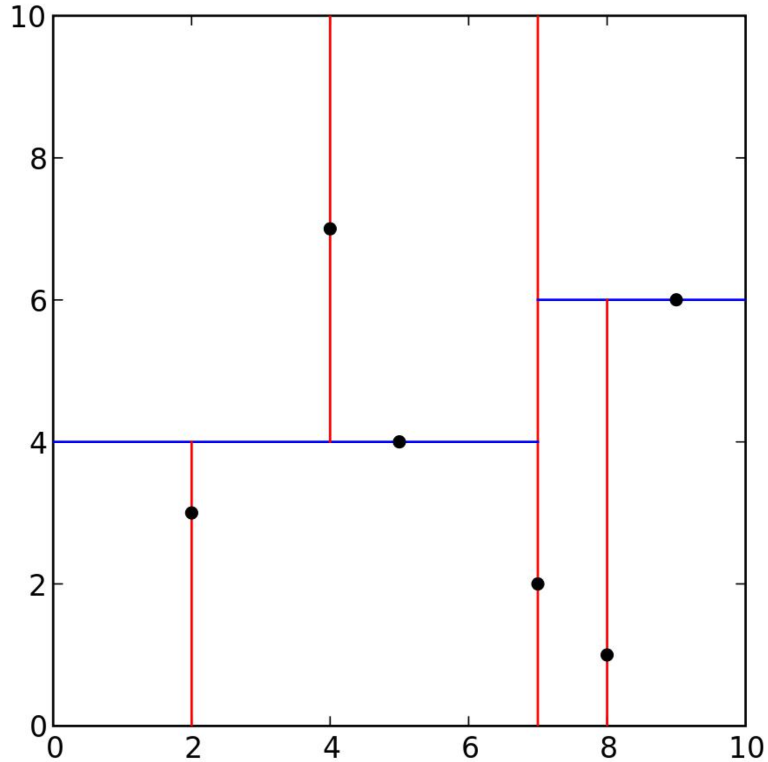


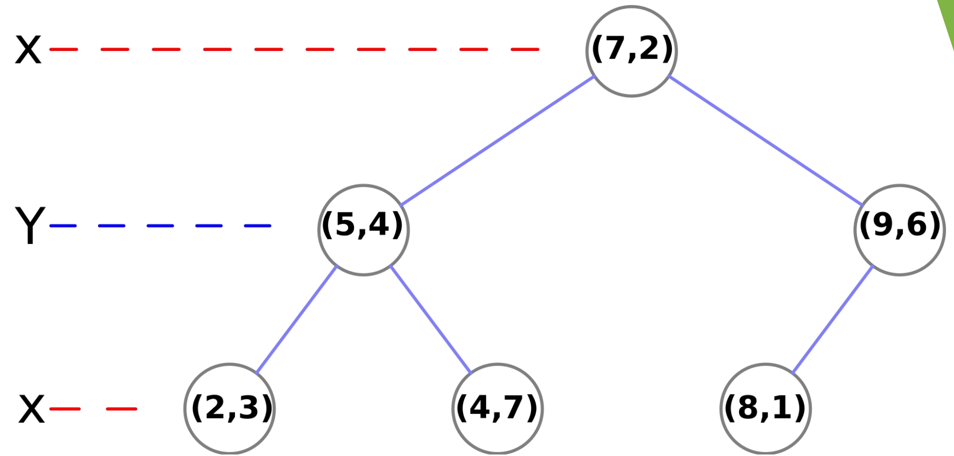
k-d Trees

- Ogni foglia dell'albero è un punto nello spazio
- Ogni nodo interno “genera” un iperpiano che divide lo spazio in due semispazi
- I punti a sinistra del semispazio sono nel sottoalbero sinistro
- I punti a destra del semispazio sono nel sottoalbero destro

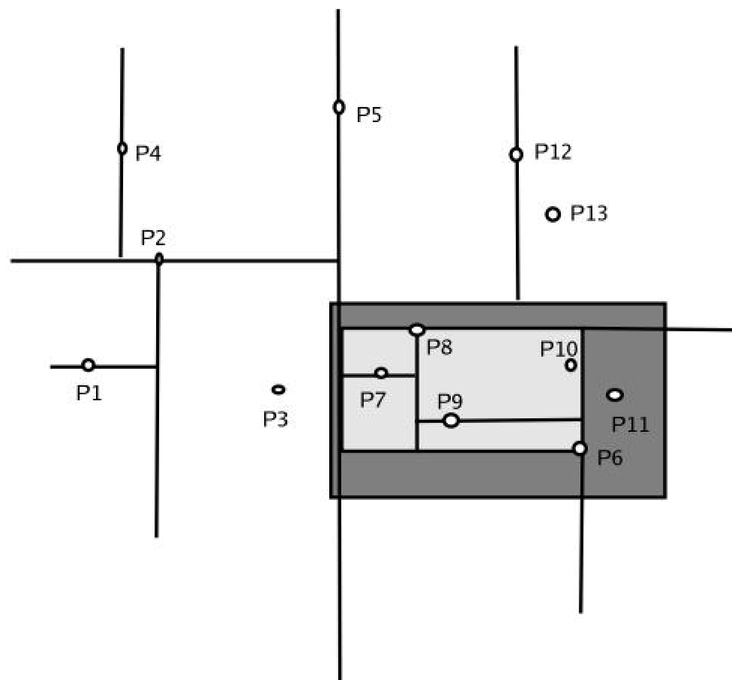
- Scendendo di un livello, si genera una partizione nella direzione “successiva”
  - Il primo livello partiziona in x
  - Il secondo livello partiziona in y
  - Il terzo livello partiziona in z
  - …

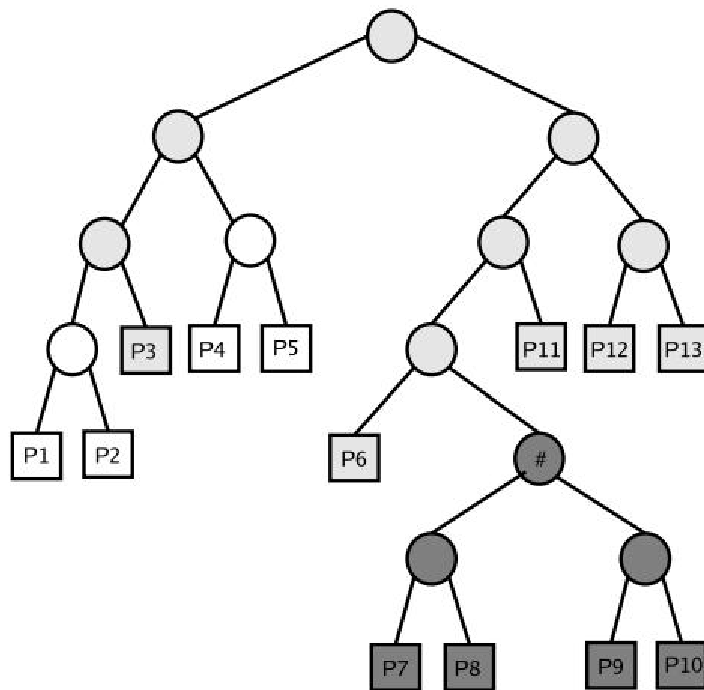
k-d Trees



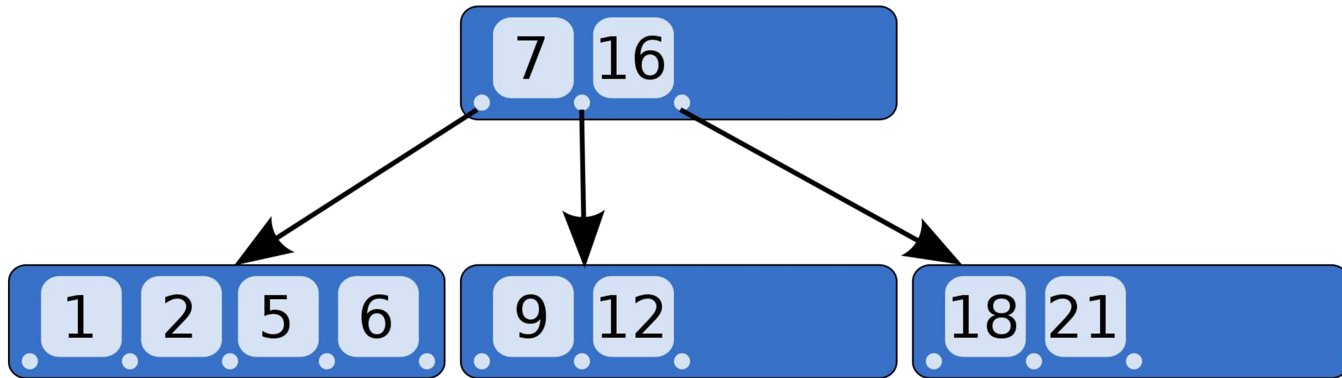


Range Query





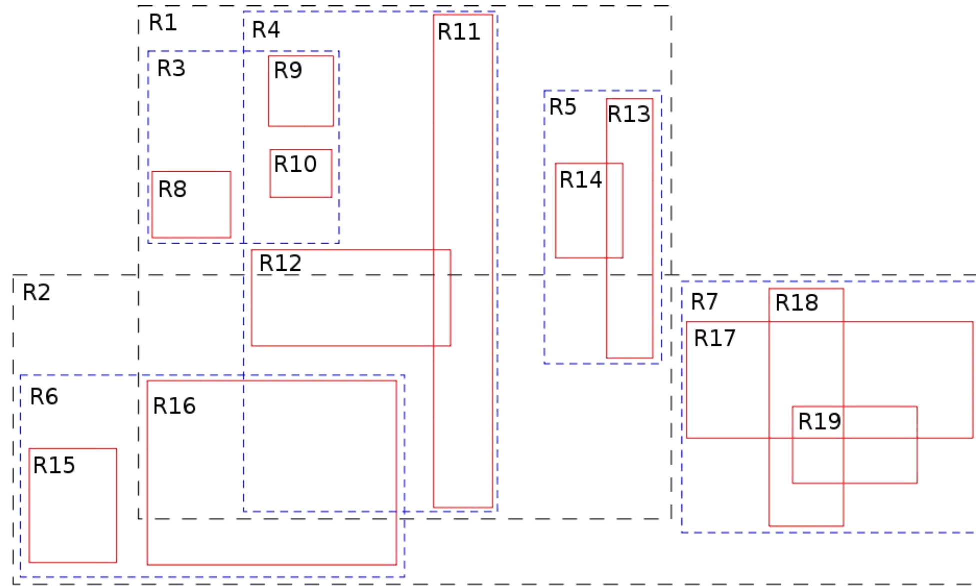
B-Tree

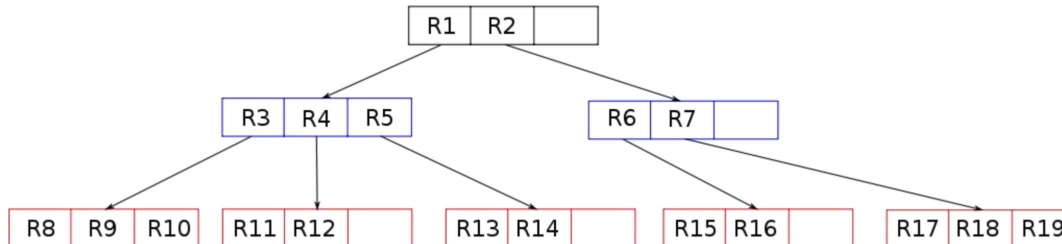


R-Tree

- Una forma particolare di B-Tree che raggruppa oggetti vicini all'interno di *bounding rectangles*
- Le foglie descrivono un singolo elemento
- Ai livelli superiori, si individuano aggregazioni superiori
- I rettangoli possono essere divisi quando il numero di punti supera una certa soglia

R-Tree





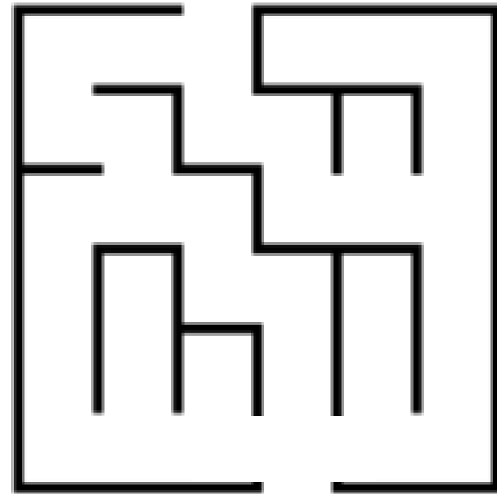
Labirinti

Backtracking

```
FIND-PATH(x, y):
    if (x,y outside maze) then: return false
    if (x,y is goal) then: return true
    if (x,y not open) then: return false

    mark x,y as part of solution path

    if (FIND-PATH(North of x,y) == true) then: return true
    if (FIND-PATH(East of x,y) == true) then: return true
    if (FIND-PATH(South of x,y) == true) then: return true
    if (FIND-PATH(West of x,y) == true) then: return true

    unmark x,y as part of solution path
    return false
```

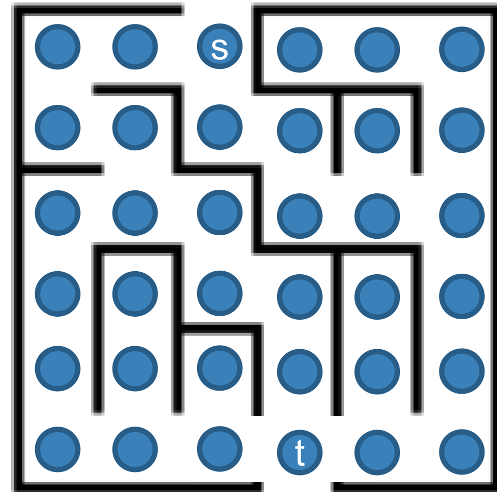
SSSP

```
SSSP(G, r):
    r.distance ← 0
    MinHeap PQ ← Ø
    foreach v in G:
        if v ≠ r then:
            v.distance ← ∞
        v.parent ← nil
        PQ.enqueue(v)
    while PQ is not empty:
        u ← PQ.getMin()
        foreach v in G.adj(u):
            if v is in PQ then:
                newDist ← u.distance + w(u, v)
                if newDist < v.distance then:
                    v.distance ← newDist
                    v.parent ← u
                    PQ.decreasePrio(v, newDist)
```



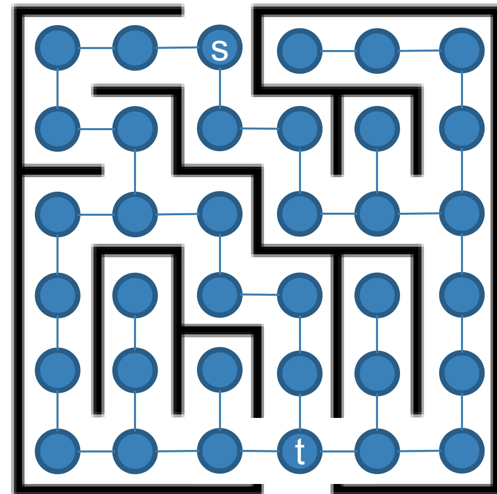
SSSP

```
SSSP(G, r):
    r.distance ← 0
    MinHeap PQ ← Ø
    foreach v in G:
        if v ≠ r then:
            v.distance ← ∞
        v.parent ← nil
        PQ.enqueue(v)
    while PQ is not empty:
        u ← PQ.getMin()
        foreach v in G.adj(u):
            if v is in PQ then:
                newDist ← u.distance + w(u, v)
                if newDist < v.distance then:
                    v.distance ← newDist
                    v.parent ← u
                    PQ.decreasePrio(v, newDist)
```



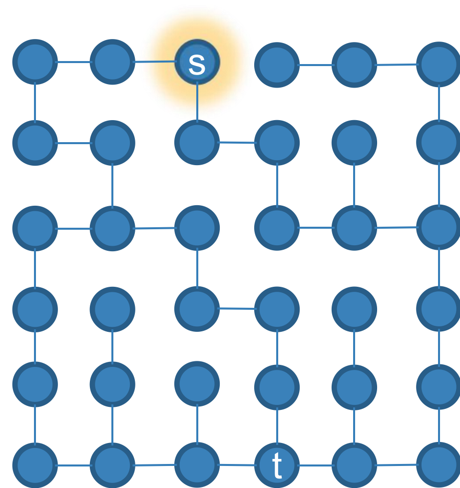
SSSP

```
SSSP(G, r):
    r.distance ← 0
    MinHeap PQ ← Ø
    foreach v in G:
        if v ≠ r then:
            v.distance ← ∞
        v.parent ← nil
        PQ.enqueue(v)
    while PQ is not empty:
        u ← PQ.getMin()
        foreach v in G.adj(u):
            if v is in PQ then:
                newDist ← u.distance + w(u, v)
                if newDist < v.distance then:
                    v.distance ← newDist
                    v.parent ← u
                    PQ.decreasePrio(v, newDist)
```



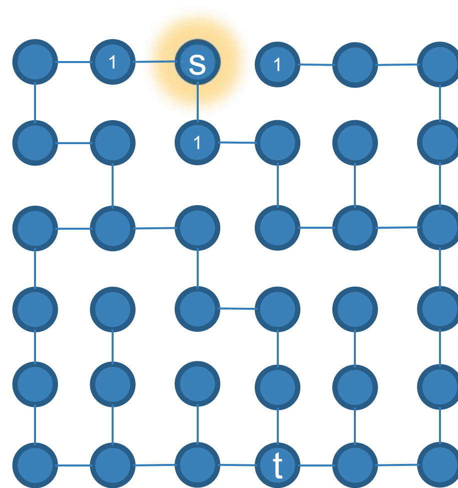
SSSP

```
SSSP(G, r):
   r.distance ← 0
   MinHeap PQ ← Ø
   foreach v in G:
      if v ≠ r then:
         v.distance ← ∞
      v.parent ← nil
      PQ.enqueue(v)
   while PQ is not empty:
      u ← PQ.getMin()
      foreach v in G.adj(u):
         if v is in PQ then:
            newDist ← u.distance + w(u, v)
            if newDist < v.distance then:
               v.distance ← newDist
               v.parent ← u
               PQ.decreasePrio(v, newDist)
```



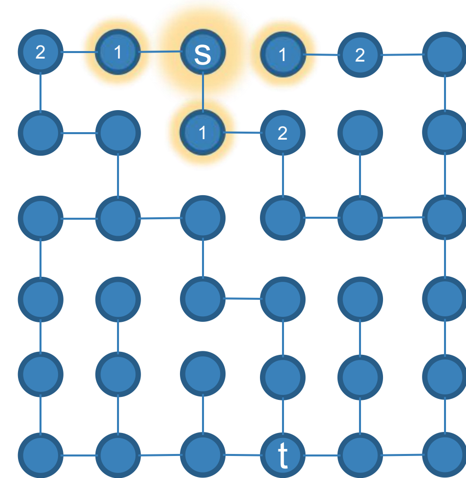
SSSP

```
SSSP(G, r):
    r.distance ← 0
    MinHeap PQ ← Ø
    foreach v in G:
        if v ≠ r then:
            v.distance ← ∞
        v.parent ← nil
        PQ.enqueue(v)
    while PQ is not empty:
        u ← PQ.getMin()
        foreach v in G.adj(u):
            if v is in PQ then:
                newDist ← u.distance + w(u, v)
                if newDist < v.distance then:
                    v.distance ← newDist
                    v.parent ← u
                    PQ.decreasePrio(v, newDist)
```



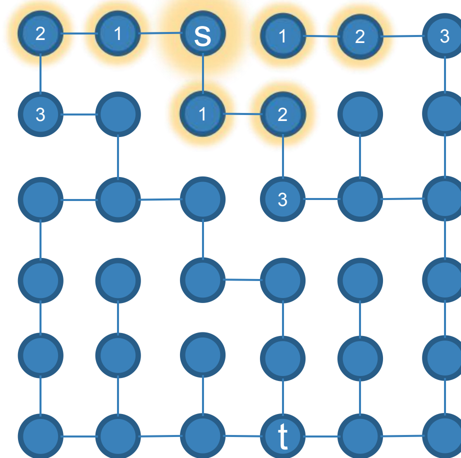
SSSP

```
SSSP(G, r):
  r.distance ← 0
  MinHeap PQ ← Ø
  foreach v in G:
    if v ≠ r then:
      v.distance ← ∞
    v.parent ← nil
    PQ.enqueue(v)
  while PQ is not empty:
    u ← PQ.getMin()
    foreach v in G.adj(u):
      if v is in PQ then:
        newDist ← u.distance + w(u, v)
        if newDist < v.distance then:
          v.distance ← newDist
          v.parent ← u
          PQ.decreasePrio(v, newDist)
```



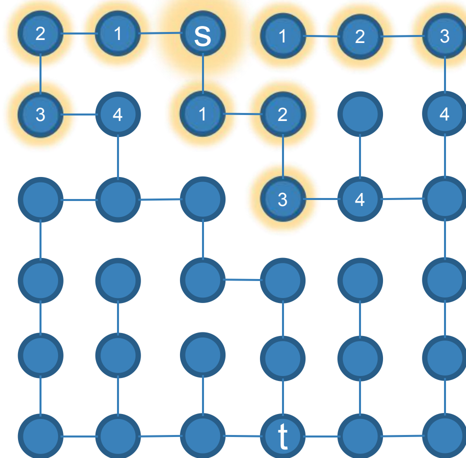
SSSP

```
SSSP(G, r):
    r.distance ← 0
    MinHeap PQ ← Ø
    foreach v in G:
        if v ≠ r then:
            v.distance ← ∞
        v.parent ← nil
        PQ.enqueue(v)
    while PQ is not empty:
        u ← PQ.getMin()
        foreach v in G.adj(u):
            if v is in PQ then:
                newDist ← u.distance + w(u, v)
                if newDist < v.distance then:
                    v.distance ← newDist
                    v.parent ← u
                    PQ.decreasePrio(v, newDist)
```



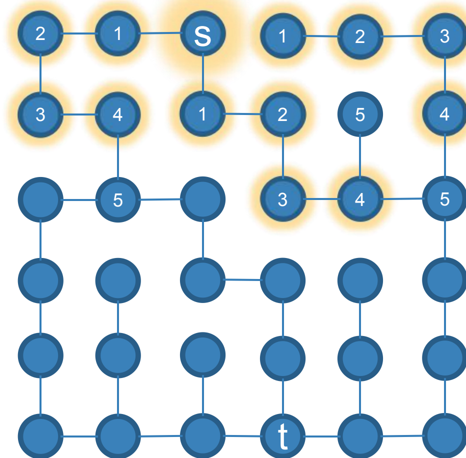
SSSP

```
SSSP(G, r):
   r.distance ← 0
   MinHeap PQ ← Ø
   foreach v in G:
      if v ≠ r then:
         v.distance ← ∞
      v.parent ← nil
      PQ.enqueue(v)
   while PQ is not empty:
      u ← PQ.getMin()
      foreach v in G.adj(u):
         if v is in PQ then:
            newDist ← u.distance + w(u, v)
            if newDist < v.distance then:
               v.distance ← newDist
               v.parent ← u
               PQ.decreasePrio(v, newDist)
```



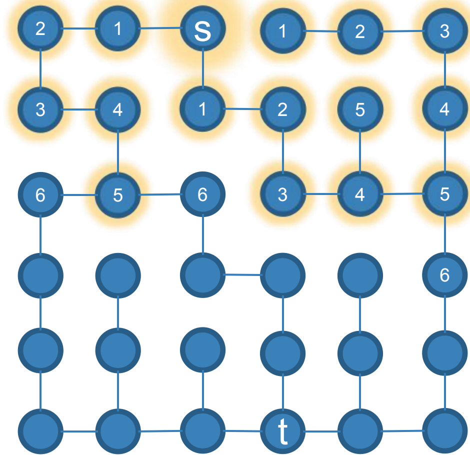
SSSP

```
SSSP(G, r):
    r.distance ← 0
    MinHeap PQ ← Ø
    foreach v in G:
        if v ≠ r then:
            v.distance ← ∞
        v.parent ← nil
        PQ.enqueue(v)
    while PQ is not empty:
        u ← PQ.getMin()
        foreach v in G.adj(u):
            if v is in PQ then:
                newDist ← u.distance + w(u, v)
                if newDist < v.distance then:
                    v.distance ← newDist
                    v.parent ← u
                    PQ.decreasePrio(v, newDist)
```



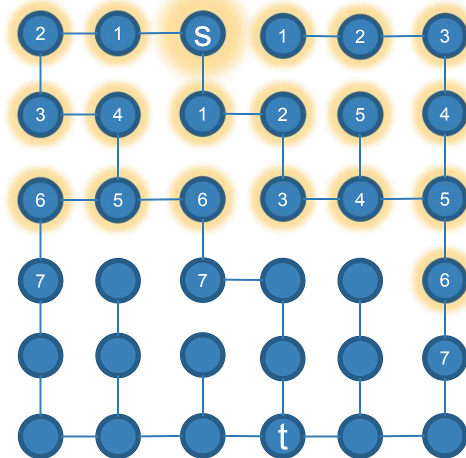
SSSP

```
SSSP(G, r):
    r.distance ← 0
    MinHeap PQ ← Ø
    foreach v in G:
        if v ≠ r then:
            v.distance ← ∞
        v.parent ← nil
        PQ.enqueue(v)
    while PQ is not empty:
        u ← PQ.getMin()
        foreach v in G.adj(u):
            if v is in PQ then:
                newDist ← u.distance + w(u, v)
                if newDist < v.distance then:
                    v.distance ← newDist
                    v.parent ← u
                    PQ.decreasePrio(v, newDist)
```



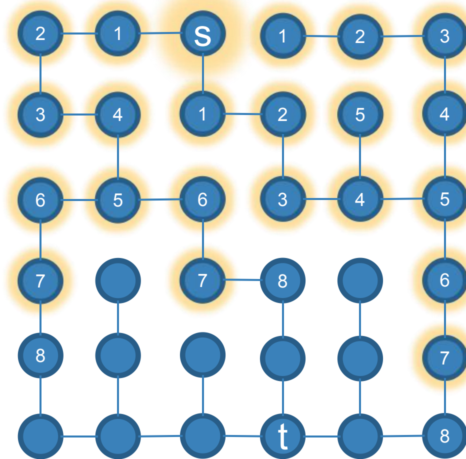
SSSP

```
SSSP(G, r):
   r.distance ← 0
   MinHeap PQ ← Ø
   foreach v in G:
      if v ≠ r then:
         v.distance ← ∞
      v.parent ← nil
      PQ.enqueue(v)
   while PQ is not empty:
      u ← PQ.getMin()
      foreach v in G.adj(u):
         if v is in PQ then:
            newDist ← u.distance + w(u, v)
            if newDist < v.distance then:
               v.distance ← newDist
               v.parent ← u
               PQ.decreasePrio(v, newDist)
```



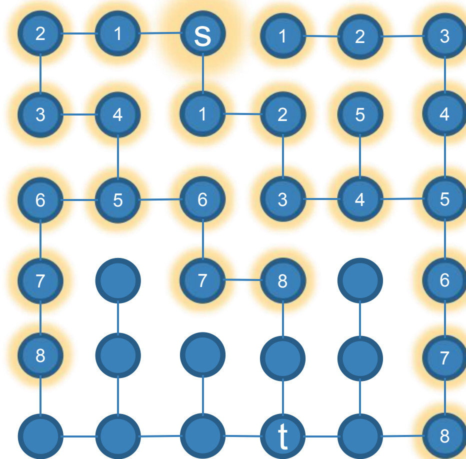
SSSP

```
SSSP(G, r):
    r.distance ← 0
    MinHeap PQ ← Ø
    foreach v in G:
        if v ≠ r then:
            v.distance ← ∞
        v.parent ← nil
        PQ.enqueue(v)
    while PQ is not empty:
        u ← PQ.getMin()
        foreach v in G.adj(u):
            if v is in PQ then:
                newDist ← u.distance + w(u, v)
                if newDist < v.distance then:
                    v.distance ← newDist
                    v.parent ← u
                    PQ.decreasePrio(v, newDist)
```



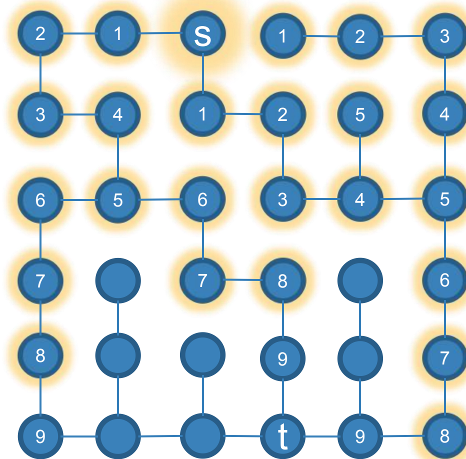
SSSP

```
SSSP(G, r):
    r.distance ← 0
    MinHeap PQ ← Ø
    foreach v in G:
        if v ≠ r then:
            v.distance ← ∞
        v.parent ← nil
        PQ.enqueue(v)
    while PQ is not empty:
        u ← PQ.getMin()
        foreach v in G.adj(u):
            if v is in PQ then:
                newDist ← u.distance + w(u, v)
                if newDist < v.distance then:
                    v.distance ← newDist
                    v.parent ← u
                    PQ.decreasePrio(v, newDist)
```



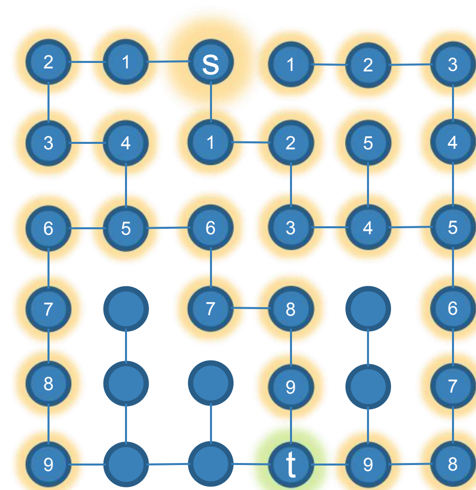
SSSP

```
SSSP(G, r):
    r.distance ← 0
    MinHeap PQ ← Ø
    foreach v in G:
        if v ≠ r then:
            v.distance ← ∞
        v.parent ← nil
        PQ.enqueue(v)
    while PQ is not empty:
        u ← PQ.getMin()
        foreach v in G.adj(u):
            if v is in PQ then:
                newDist ← u.distance + w(u, v)
                if newDist < v.distance then:
                    v.distance ← newDist
                    v.parent ← u
                    PQ.decreasePrio(v, newDist)
```



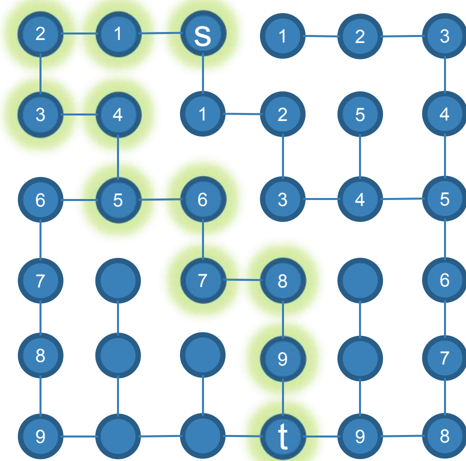
SSSP

```
SSSP(G, r):
    r.distance ← 0
    MinHeap PQ ← Ø
    foreach v in G:
        if v ≠ r then:
            v.distance ← ∞
        v.parent ← nil
        PQ.enqueue(v)
    while PQ is not empty:
        u ← PQ.getMin()
        foreach v in G.adj(u):
            if v is in PQ then:
                newDist ← u.distance + w(u, v)
                if newDist < v.distance then:
                    v.distance ← newDist
                    v.parent ← u
                    PQ.decreasePrio(v, newDist)
```



SSSP

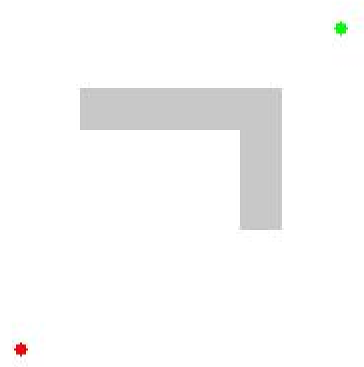
```
SSSP(G, r):
    r.distance ← 0
    MinHeap PQ ← Ø
    foreach v in G:
        if v ≠ r then:
            v.distance ← ∞
        v.parent ← nil
        PQ.enqueue(v)
    while PQ is not empty:
        u ← PQ.getMin()
        foreach v in G.adj(u):
            if v is in PQ then:
                newDist ← u.distance + w(u, v)
                if newDist < v.distance then:
                    v.distance ← newDist
                    v.parent ← u
                    PQ.decreasePrio(v, newDist)
```



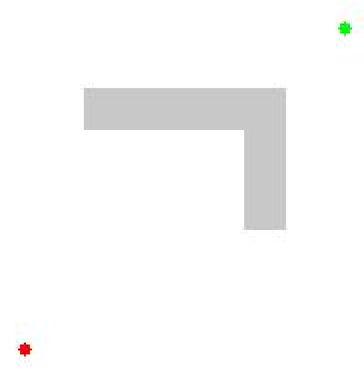
A*

- L'algoritmo di Dijkstra esplora "a macchia d'olio"
  - Non ha informazioni su dove si trova il punto di arrivo
- Si può fare di meglio se abbiamo questa informazione?
  - Possiamo fare delle scelte greedy che ci "avvicinano" alla destinazione
- A* utilizza un'euristica che "sottostima" la distanza
- Vengono mantenuti tre insiemi:
  - Nodi da esplorare
  - Nodi esplorati
  - Nodi di frontiera
- Si sceglie il nodo adiacente alla frontiera che avvicina di più all'obiettivo

SSSP vs A*



A*



Dijkstra