

## Algoritmo di ricerca

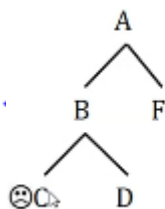
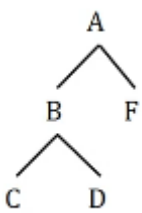
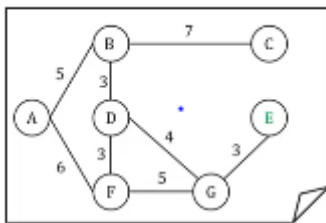
Hanno lo stesso **scopo: cercare una strada da uno stato iniziale ad uno stato di goal, col minore costo possibile** (se è un problema di ottimizzazione e non di semplice fattibilità).

La differenza fra i vari algoritmi di ricerca sta nel modo in cui essi costruiscono il percorso verso il goal. Vi sono diversi metodi verso il goal, con diverse prestazioni.

### - Ricerca in profondità

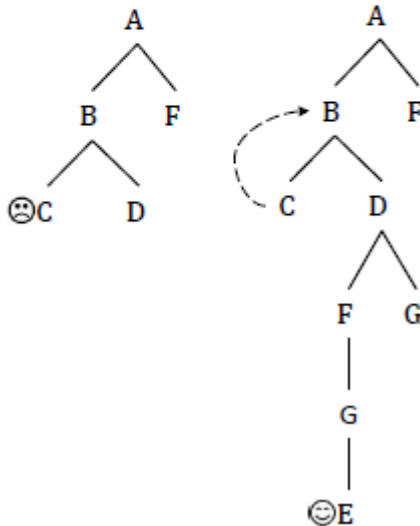
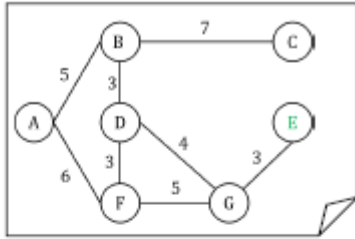
#### Depth First Search DFS.

L'algoritmo, partendo dal nodo iniziale esplora il nodo raggiungibile che sta nella profondità maggiore.



### Operazione di backtracking

Riconsiderare la scelta è l'operazione di **backtracking**.



La soluzione è quindi:  $A \rightarrow B \rightarrow D \rightarrow F \rightarrow G \rightarrow E$

## Depth-First Search (DFS)

- Una ricerca in profondità (DFS) sceglie il nodo più profondo (più in basso) non ancora esplorato nell'albero (grafo) di ricerca
- Se ci sono nodi alla stessa profondità adottiamo l'ordine lessicografico (qualsiasi altro criterio va bene)
- Evitiamo i loop sullo stesso ramo! (sarebbe ridondante, stiamo già applicando buone norme di efficienza)
- Un vicolo cieco ha interrotto la ricerca, DFS non è completa! Possiamo rimediare?
- Diamo alla DFS la capacità di fare **backtracking**: riconsiderare decisioni valutate in precedenza e ripartire su un percorso alternativo
- Soluzione:  $A \rightarrow B \rightarrow D \rightarrow F \rightarrow G \rightarrow E$

Il backtracking è importante perché rende la ricerca completa, se esiste una soluzione la trova.

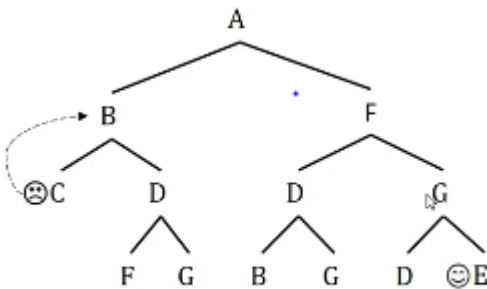
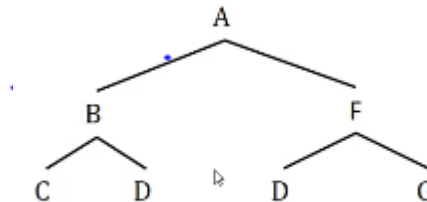
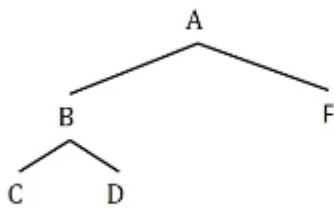
### Proprietà del Depth First Search DFS

# Depth-First Search (DFS)

- La ricerca in profondità con rimozione dei loop e backtracking è **corretta e completa**
- Chiamiamo  $b$  il massimo branching factor, cioè il massimo numero di azioni disponibili in uno stato (*out degree* del nodo, numero di archi uscenti)
- Chiamiamo  $d$  la profondità massima di una soluzione, cioè il massimo numero di azioni in un percorso dallo stato iniziale al goal
- Complessità spaziale:  $O(d)$
- Complessità temporale:  $1 + b + b^2 + b^3 + \dots + b^d = O(b^d)$

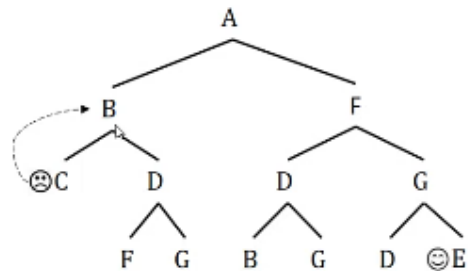
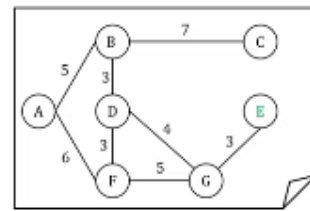
## Breadth First Searchg BFS - Ricerca in ampiezza

L'algoritmo, partendo dal nodo iniziale esplora il nodo raggiungibile che sta nella profondità minore tra quelli non ancora esplorati.



# Breadth-First Search (BFS)

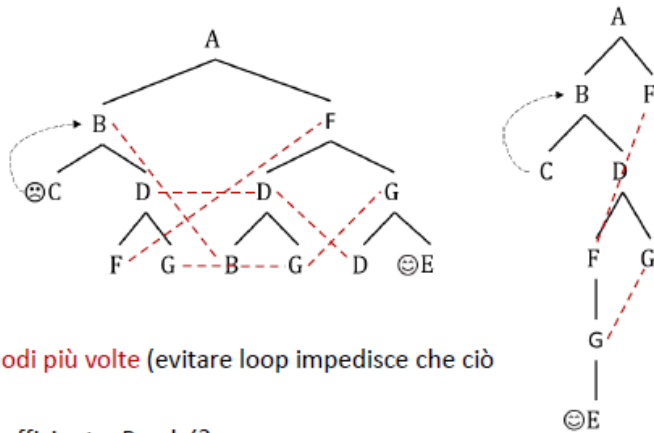
- La ricerca in ampiezza, Breadth-First Search (BFS) il nodo meno profondo (più in alto) non ancora esplorato, procede quindi per livelli di profondità
- Soluzione:  $A \rightarrow F \rightarrow G \rightarrow E$
- Ha un comportamento più conservativo rispetto a DFS e non ha necessità di riconsiderare decisioni
- Chiamiamo  $q$  la profondità minima a cui sta una soluzione (in generale  $q \leq d$ )
- Complessità spaziale:  $O(b^q)$
- Complessità temporale:  $O(b^q)$



Questo modo di procedere è più conservativo nel senso che, prima di andare in profondità, l'algoritmo si assicura di aver esplorato tutto ciò che è possibile trovare alla profondità corrente, sviluppando così la ricerca in orizzontale e non in verticale. **La ricerca in orizzontale è tipicamente la ricerca sempre corretta e completa e che garantisce la soluzione ottima.** Ha una complessità spaziale e temporale leggermente peggiore di quella della ricerca in profondità.

# Nodi ridondanti

- Nell'ottica di migliorare l'efficienza dei precedenti algoritmi, analizziamo gli alberi di ricerca prodotti da DFS e BFS
- Essi rappresentano lo storico complessivo del procedimento che ha portato alla risoluzione del problema
- Sia DFS che BFS hanno visitato alcuni nodi più volte (evitare loop impedisce che ciò accada solo lungo lo stesso ramo)
- In generale, questo non sembra molto efficiente. Perché?
- Idea: scartare un nodo appena generato se già presente da qualche parte nell'albero, possiamo farlo con una lista: **la lista degli accodamenti** (Enqueued list, EQL)

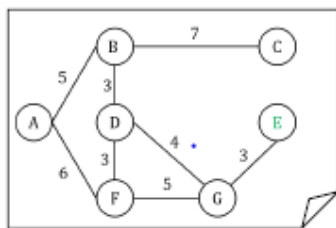


## Lista degli accodamenti (EQL = Enqueued List)

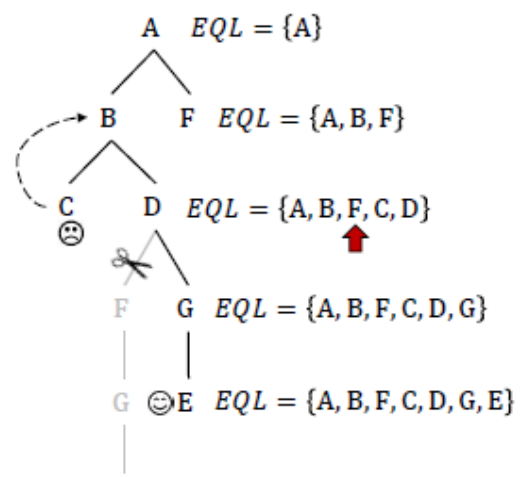
La presenza dei ridondanti non è efficiente e non è interessante considerare più volte il passaggio dallo stesso nodo anche se è su un altro percorso.

Si adotta quindi una lista degli accodamenti (**EQL = Enqueued List**). Ogni volta che l'algoritmo espande un nodo lo aggiunge alla lista EQL.

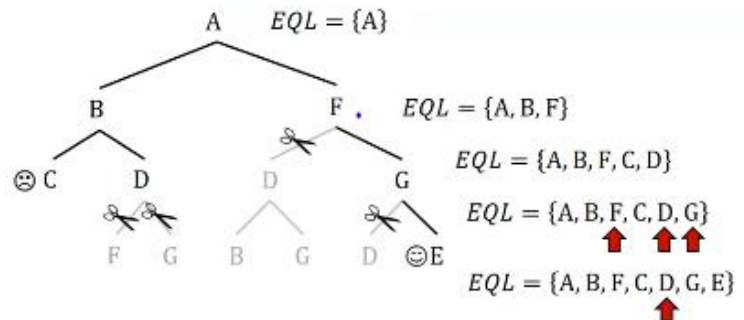
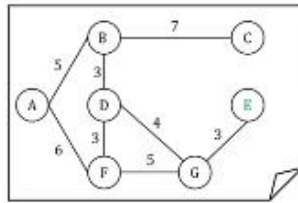
## DFS con EQL



- Il nodo F è già in lista! Lo scarto potando un ramo dell'albero (operazione di **pruning**)
- Risparmio di tempo e spazio: evito di generare una parte dell'albero di ricerca



## BFS con EQL



- Anche in questo caso una parte dell'albero non viene generata con conseguente risparmio di tempo e spazio
- Correttezza e completezza sono preservate
- Se non esistesse il pruning DFS e BFS sarebbero raramente applicabili

↳

## Uniform Cost Search UCS - ricerca a costo uniforme

La **ricerca a costo uniforme: UCS (Uniform Cost Search)** è un altro tipo di algoritmo di ricerca. Si tratta di un'estensione della ricerca in ampiezza. Qui sono interessato al percorso che minimizza il costo totale, il percorso ottimo, per minimizzare il costo totale.

## Ricerca della soluzione ottima

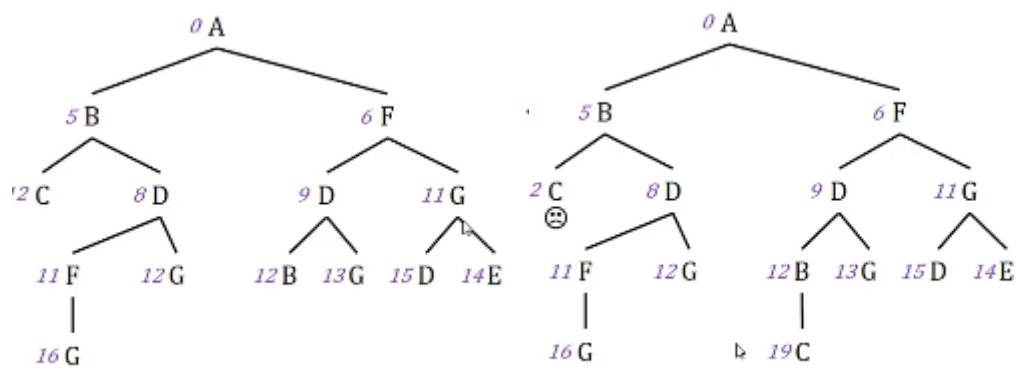
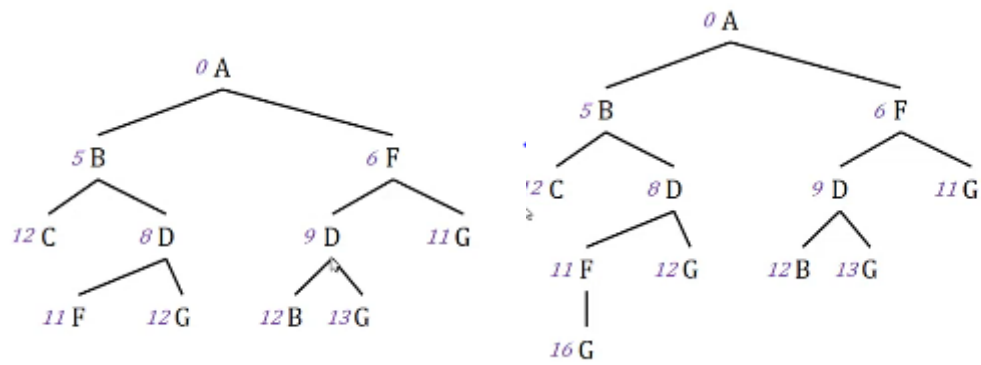
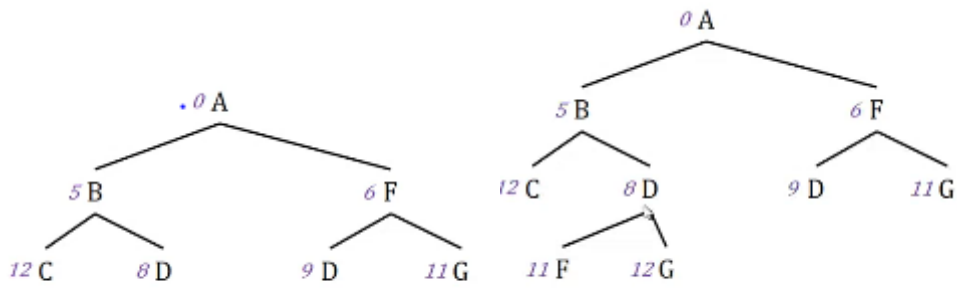
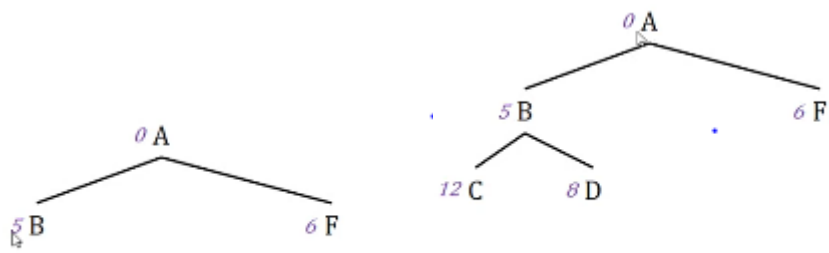
- Ora supponiamo di essere interessati alla soluzione ottima, quella con il costo minimo: non un percorso qualsiasi verso il goal, ma il più economico possibile

**Richiamo**

- Le transizioni hanno un costo additivo (che non abbiamo considerato fino ad ora)
- Ad esempio il path  $A \rightarrow B \rightarrow D \rightarrow G$  richiede di pagare un costo pari a  $5 + 3 + 4 = 12$

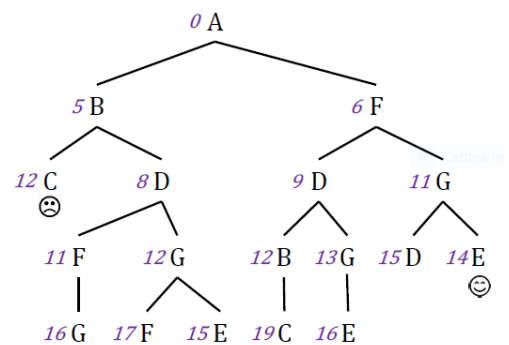
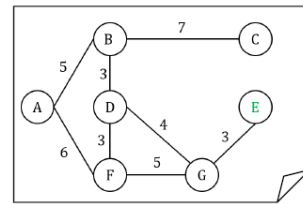
- Per ideare un algoritmo di ricerca ottimo prendiamo le mosse da **BFS**. Perché sembra ragionevole farlo?
- Generalizziamo l'idea di BFS a quella di **Uniform Cost Search (UCS)**, la ricerca a costo uniforme
- BFS procede per livelli di profondità, **UCS procede per livelli di costo** (di conseguenza, se i costi sono tutti uguali a una costante BFS e UCS coincidono!)

La ricerca a costo uniforme UCS funziona esattamente come nella ricerca in ampiezza (BFS), solo che usa il costo di raggiungimento del nodo per valutare la profondità dei nodi da esplorare.



# Uniform Cost Search (UCS)

- Nell'albero di ricerca, teniamo traccia del costo accumulato sul percorso dal nodo iniziale a ogni nodo  $V$ :  $g(V)$
- Non consideriamo EQL
- UCS: selezione (espansione) del nodo con  $g$  minore ancora da esplorare (sulla frontiera)
- Goal check: se il **nodo selezionato per l'espansione** è un goal, mi fermo e restituisco la soluzione
- Abbiamo trovato il percorso ottimale per l'obiettivo? In questo specifico caso, possiamo rispondere sì ispezionando il grafico
- E con istanze più grandi? Possiamo dimostrare che troverà sempre l'ottimo?
- In realtà, possiamo fare un'affermazione più forte:  
**ogni volta che UCS seleziona per la prima volta un nodo per l'espansione, il percorso che, sull'albero di ricerca, porta a quel nodo ha un costo minimo**



**Questo metodo permette di essere certi di trovare una strada per il goal e a costo minore.**

Quando trovo un nodo per la prima volta l'algoritmo scopre il percorso ottimale per quel nodo perché sta procedendo per linee di costo. Quindi, con questo metodo, l'algoritmo quando scopre per la prima volta il nodo di goal, scopre contemporaneamente anche il suo percorso ottimo.



## Ricerca informata: l'algoritmo A\*

L'algoritmo della ricerca a costo uniforme appartiene ad una classe di algoritmi che si dice **non informata**.

## Ricerca non informata e informata

- Gli algoritmi di ricerca decidono quale nodo espandere attraverso delle regole che applicano in funzione della conoscenza del problema e del processo di ricerca svolto fino al tempo presente
- Una ricerca è **non informata** se utilizza solo la conoscenza del problema che è specificata nella sua definizione: il grafo, le sue connessioni, e il criterio con cui scegliere il prossimo nodo non considera la bontà del nodo stesso
- Una ricerca **informata** va oltre la definizione del problema sfruttando della conoscenza aggiuntiva: ciò che quel grafo, quelle connessioni e quei costi rappresentano nel mondo reale, oltre il formalismo agnostico che li esprime
- Dato un generico stato  $S$ , usando questa conoscenza, un algoritmo informato **stima** la bontà di  $S$  attraverso una funzione  $f(S)$  e guida la ricerca usando  $f$
- Approccio **best-first**: **espandere prima gli stati che hanno una  $f$  migliore**
- Esistono diversi algoritmi di ricerca best-first, la differenza la fa il **come  $f$  è definita**

La ricerca non informata utilizza solo la conoscenza del problema che è specificata nella sua definizione. Usa semplicemente la mappa, il grafo e le sue connessioni.

Una conoscenza informata, invece, cerca di andare oltre questa definizione del problema. Invece di usare una funzione  $g$  del costo usa una **funzione  $f$  che stima la bontà di un certo nodo**. Questo tipo di approccio si chiama **best-first** e **ottimizza una funzione che non è semplicemente il costo di raggiungimento del goal**, ma qualcosa di più sofisticato. A seconda di come viene specificata la funzione  $f$ , avremo diversi algoritmi di tipo best-first.

## A\*

- L'idea alla base di A\* è semplice: eseguire un UCS, ma invece di considerare soltanto i costi accumulati  $g$ , considerare la funzione  $f(s) = g(s) + h(s)$
- Con  $g(s)$  indichiamo, come prima, il costo accumulato lungo in path che arriva nello stato  $s$
- Con  $h(s)$  indichiamo una stima del costo ancora da spendere per arrivare al goal lungo il path ottimo, è l'**euristica**
- L'algoritmo di ricerca seleziona per l'espansione i nodi sulla frontiera che minimizzano  $f$

Chi calcola  $h$ ? E come?

## Euristica

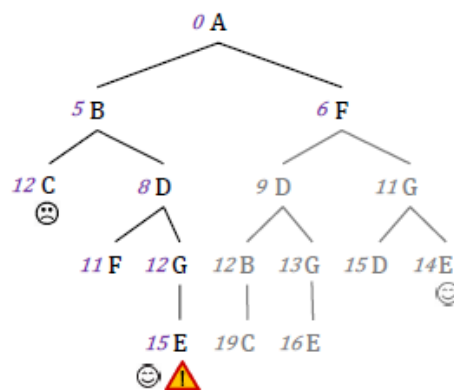
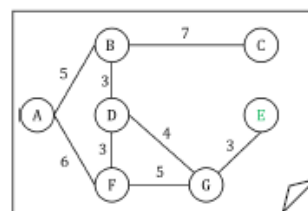
Con l'euristica stimiamo quanto uno stato è lontano dallo stato di goal sul percorso ottimo.

È molto importante che l'euristica sia ammissibile. Ammissibile vuol dire che non deve mai sovrastimare il costo, la stima deve essere sempre per difetto, mai per eccesso. La sovrastima può confondere l'algoritmo e far sì che esso espanda nodi che in realtà si allontanano dal goal aumentandone il costo computazionale invece che ridurlo.

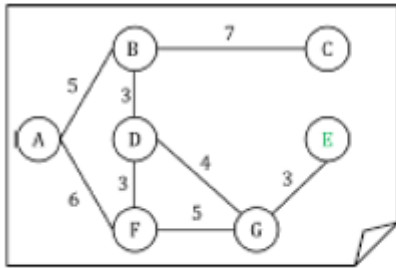
### La proprietà dell'ammissibilità nell'euristica

## Euristica (ammissibilità)

- Una proprietà fondamentale che una buona euristica deve avere è l'**ammissibilità**.
- Un'euristica  $h$  è ammissibile se per ogni possibile stato  $S$   $h(S)$  non sovrastima il costo del path minimo da  $s$  al goal
- In pratica: la stima deve essere ottimista!
- Se questa proprietà non vale, l'algoritmo di ricerca potrebbe non riconoscere il percorso ottimo!
- **Esempio:** supponiamo che l'euristica sia sempre uguale a 0 (caso degenero sempre ammissibile) tranne che per lo stato F:  $h(F) = 100$

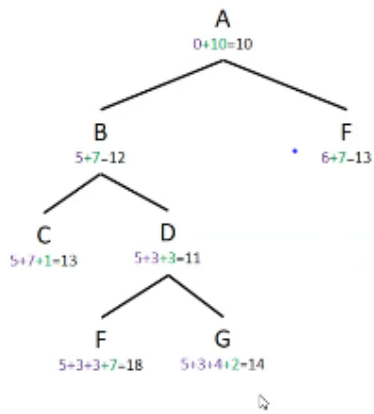


# A\* (con EXL)

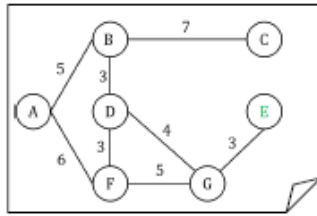


stato	Euristica $h(v)$
A	10
B	7
C	1
D	3
E	0
F	7
G	2

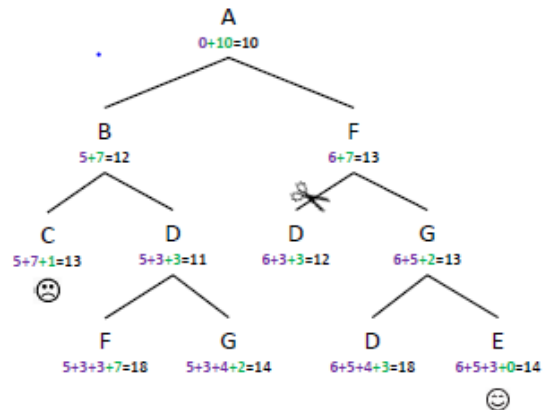
Cattura rettangoli



# A\* (con EXL)



stato	Euristica $h(v)$
A	10
B	7
C	1
D	3
E	0
F	7
G	2

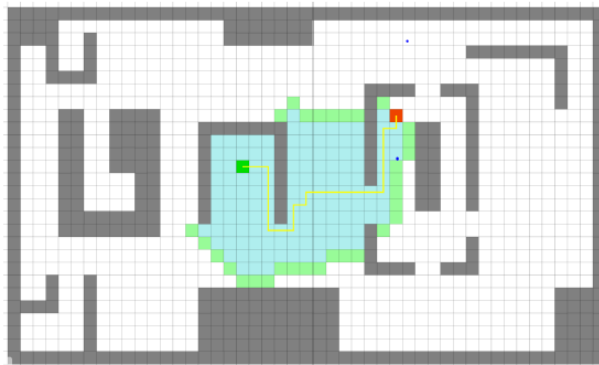


- Sembrerebbe che grazie all'ammissibilità manteniamo la stessa proprietà di ottimalità che abbiamo dimostrato per UCS: se non sovrastimiamo non possiamo scartare il path ottimo!
- Non abbiamo fatto i conti con la EXL

Prendiamo un altro esempio. Qui abbiamo una mappa in cui la parte bianca è esplorabile, la

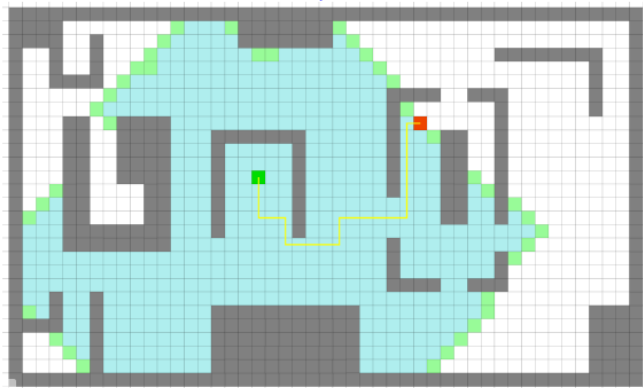
## A\* in azione a confronto con UCS (ricerca a costo uniforme)

A\*



<https://qiao.github.io/PathFinding.js/visual/>

## Ricerca a costo uniforme (UCS)



<https://qiao.github.io/PathFinding.js/visual/>

## Utilizzo dell'algoritmo A\* per risolvere il gioco del 15

Qua di seguito verrà descritto il codice in allegato a questo documento di testo, un'applicazione dell'algoritmo A\* utilizzato per risolvere il problema del gioco del 15;

Il gioco del 15 può essere formalizzato come una matrice 4x4 dove all'interno della matrice, nelle posizioni I e J, vi possono essere presenti una e una sola volta i numeri da 1 a 15 in ordine sparso. Vi è inoltre un elemento della matrice che non rappresenta questi numeri che rappresentiamo con lo zero.

Obiettivo del gioco è quello di ordinare i numeri da 1 a 15 potendo spostare solo lo zero, passo per passo, cambiando il suo indice di riga o di colonna di 1 e invertendone la posizione con il numero che esso andrebbe a sovrascrivere.



Goal dell'algoritmo

Partendo da uno qualsiasi stato ( nodo ) iniziale dove i numeri sono persi in modo arbitrario l'azione di spostare lo zero esplorerà un nuovo stato (nodo) e il costo per il passaggio da uno stato precedente ad un altro spostando un indice (di riga o di colonna) dello zero di uno è unitario. La funzione  $F(s)$  che utilizzerà l'algoritmo A\* per uno generico stato  $s$  esplorato sarà quindi la somma del costo totale dei passi compiuti per raggiungere lo stato  $s$  a partire dallo stato iniziale e di un euristica che rappresenta una stima del costo che si dovrà ancora spendere per raggiungere , dallo stato  $s$ , lo stato finale di goal.

```
# -*- coding: UTF-8 -*-
from Node import Node

class ManhattanDistance:
    """
    Implementation of Manhattan distance heuristic
    for 15-puzzle positions
    """

    def __init__(self):
        self._goal = Node([
            [ 1,  2,  3,  4],
            [ 5,  6,  7,  8],
            [ 9, 10, 11, 12],
            [13, 14, 15,  0]
        ], [], self)

    def compute(self, node):
        """
        Computes Manhattan distance of the given Node
        """
        score = 0
        for value in range(1, 16):
            iGoal, jGoal = self._goal.getCoordByValue(value)
            iActual, jActual = node.getCoordByValue(value)
            score += abs(iGoal - iActual) + abs(jGoal - jActual)
        return score
```

L'euristica utilizzata dal codice prende in analisi la posizione di ogni numero dello stato corrente e calcola quanto dista la sua posizione attuale da quella finale di goal; Dopo aver calcolato le distanze per ciascun numero il valore finale dell'euristica sarà la somma di tutti i contributi di ogni numero.

Ad esempio nello stato iniziale:

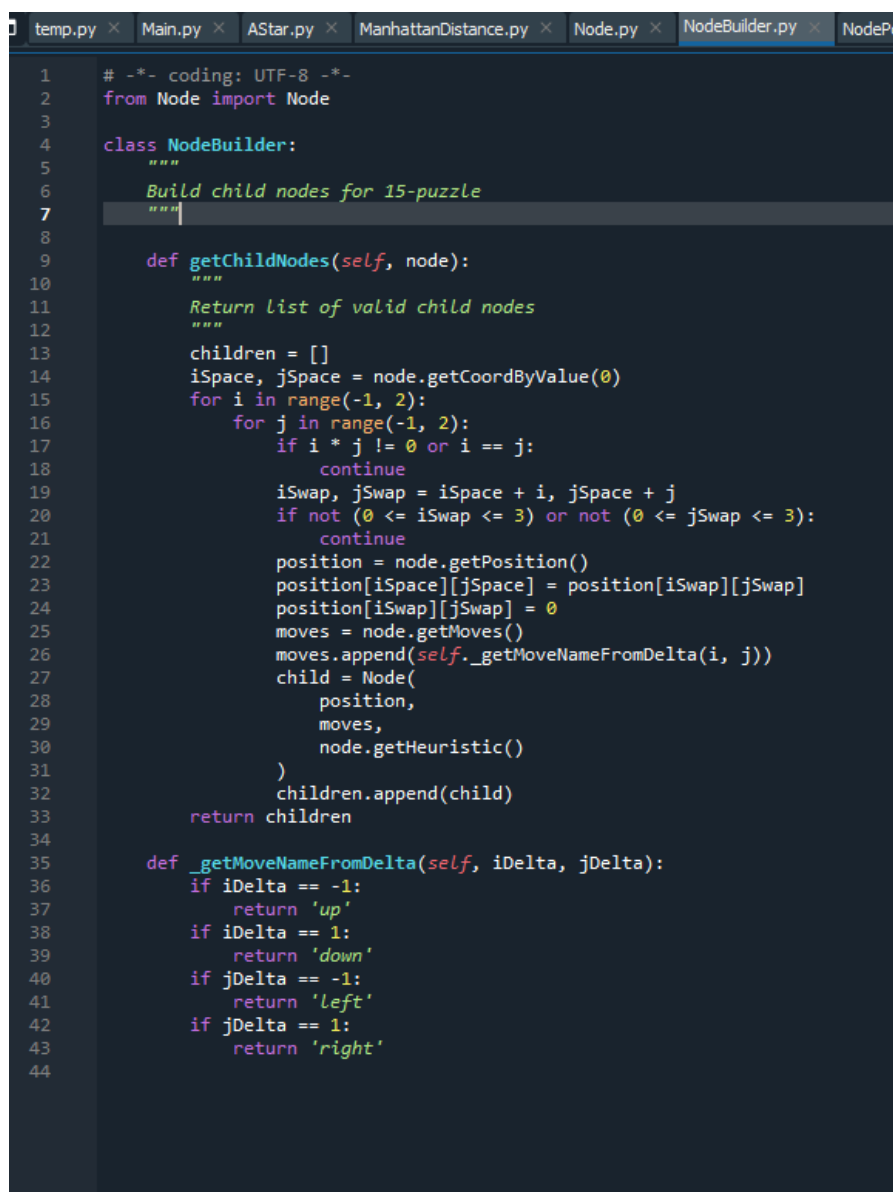
```
start = [
    [ 1,  6,  7,  5],
    [ 9,  3, 10,  2],
    [13,  8,  4, 12],
    [14, 11, 15,  0]
]
```

Il numero 1 è nella sua posizione finale, quindi darà contributo zero all'euristica,  
Il numero 6 dista 1 dalla sua posizione finale ( basta cambiare di 1 il suo indice di riga),  
il numero 10 invece dista 2 in quanto bisogna cambiare di 1 sia il suo indice di riga che di colonna.  
In questo caso l'euristica totale ha come valore 24.

Si noti che questa euristica sicuramente sottostima il numero di passi totali da compiere in quanto valuta il numero minimo di spostamenti che dovrebbero fare i valori come se potessero muoversi liberamente, nella realtà per spostarli occorrerà sicuramente un numero superiore di mosse.

Andando ad analizzare le varie funzioni del codice:

NODEBUILDER.py



```
1  # -*- coding: UTF-8 -*-
2  from Node import Node
3
4  class NodeBuilder:
5      """
6      Build child nodes for 15-puzzle
7      """
8
9  def getChildNodes(self, node):
10     """
11     Return list of valid child nodes
12     """
13     children = []
14     iSpace, jSpace = node.getCoordByValue(0)
15     for i in range(-1, 2):
16         for j in range(-1, 2):
17             if i * j != 0 or i == j:
18                 continue
19             iSwap, jSwap = iSpace + i, jSpace + j
20             if not (0 <= iSwap <= 3) or not (0 <= jSwap <= 3):
21                 continue
22             position = node.getPosition()
23             position[iSpace][jSpace] = position[iSwap][jSwap]
24             position[iSwap][jSwap] = 0
25             moves = node.getMoves()
26             moves.append(self._getMoveNameFromDelta(i, j))
27             child = Node(
28                 position,
29                 moves,
30                 node.getHeuristic()
31             )
32             children.append(child)
33     return children
34
35 def _getMoveNameFromDelta(self, iDelta, jDelta):
36     if iDelta == -1:
37         return 'up'
38     if iDelta == 1:
39         return 'down'
40     if jDelta == -1:
41         return 'left'
42     if jDelta == 1:
43         return 'right'
44
```

Dato un nodo iniziale crea e appende nella lista dei nodi esplorabili i nodi figli ottenibili spostando lo zero per riga o per colonna

## NODE.py

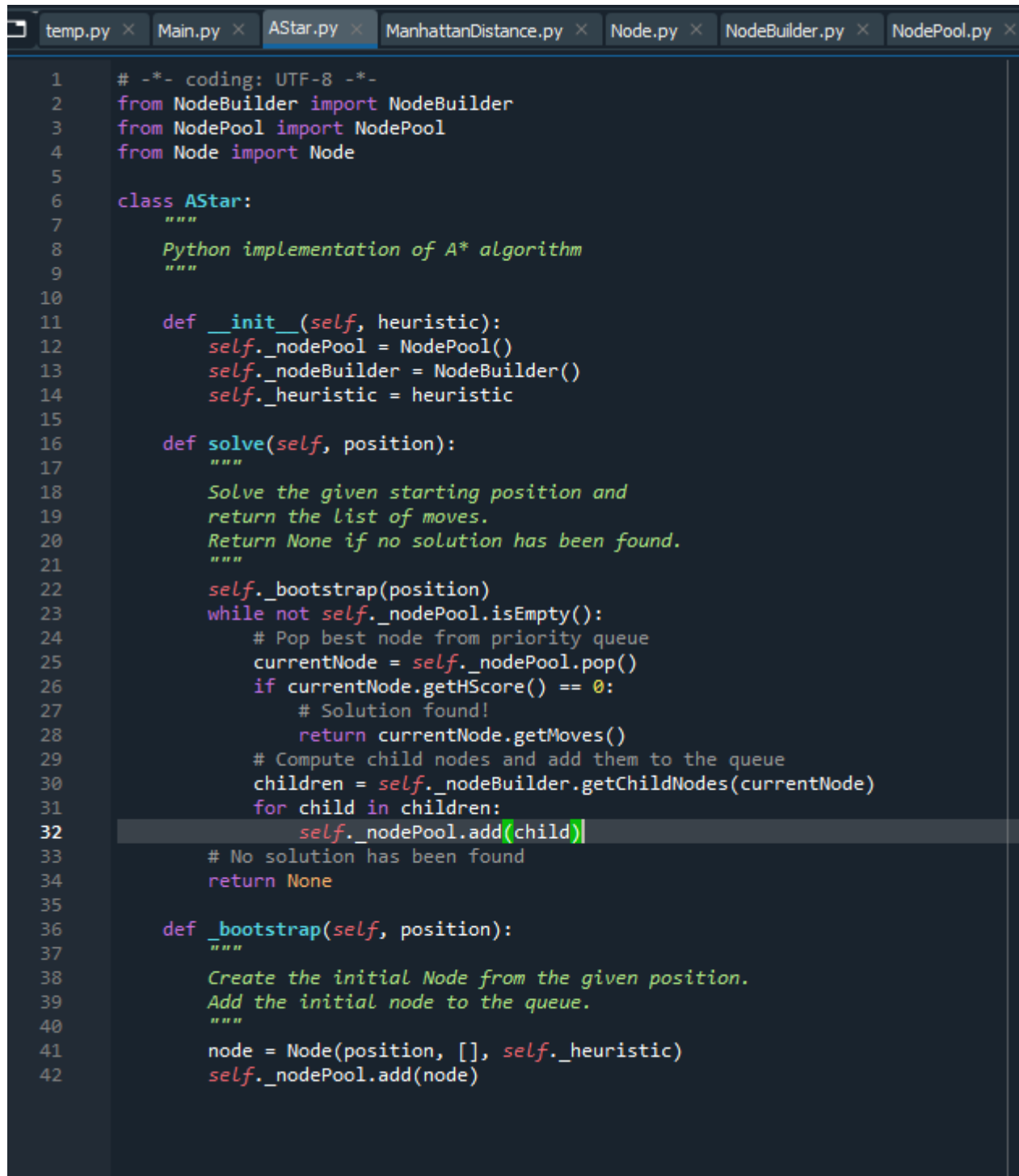
```
1  # -*- coding: UTF-8 -*-
2  import copy
3
4  class Node:
5      """
6      A node is a position (immutable) in the puzzle
7      It also contains the gScore, hScore, fScore and
8      the List of moves which bring us to this position
9      """
10
11     def __init__(self, position, moves, heuristic):
12         """
13         Builds a new node
14         """
15         self._position = position
16         self._moves = moves
17         self._heuristic = heuristic
18         self._hScore = None
19
20     def getPosition(self):
21         """
22         Get position of the Node (nested List of integers)
23         """
24         return copy.deepcopy(self._position)
25
26     def getGScore(self):
27         """
28         Get gScore of the Node
29         """
30         return len(self._moves)
31
32     def getHScore(self):
33         """
34         Get hScore of this Node. Heuristic passed in
35         the constructor will be used for computation
36         """
37         if self._hScore is None:
38             self._hScore = self._heuristic.compute(self)
39         return self._hScore
40
41     def getFScore(self):
42         """
43         Get fScore of the Node
44         fScore = gScore + hScore
45         """
46         return self.getGScore() + self.getHScore()
47
48     def getMoves(self):
49         """
50         Return List of moves which bring us to
51         this position
52         """
53         return copy.copy(self._moves)
54
55     def getHeuristic(self):
56         """
57         Return heuristic used to compute hScore for this node
58         """
59         return self._heuristic
60
61     def getCoordByValue(self, value):
62         """
63         Get i and j coord of the given value
64         """
65         i = 0
66         for row in self._position:
67             j = 0
68             for cell in row:
69                 if cell == value:
70                     return [i, j]
71                 j += 1
72             i += 1
```

È la classe utilizzata per descrivere il nodo ( o lo stato) del gioco del 15  
si nota come la funzione di costo (che verrà utilizzata dall'algoritmo A\* per discriminare quale



deve essere il nodo successivo da esplorare)  $F$  è la somma della funzione  $G$ , rappresentante i passi compiuti per arrivare a quel nodo e della funzione euristica  $H$ .

Astar.py



```
1  # -*- coding: UTF-8 -*-
2  from NodeBuilder import NodeBuilder
3  from NodePool import NodePool
4  from Node import Node
5
6  class AStar:
7      """
8      Python implementation of A* algorithm
9      """
10
11     def __init__(self, heuristic):
12         self._nodePool = NodePool()
13         self._nodeBuilder = NodeBuilder()
14         self._heuristic = heuristic
15
16     def solve(self, position):
17         """
18         Solve the given starting position and
19         return the list of moves.
20         Return None if no solution has been found.
21         """
22         self._bootstrap(position)
23         while not self._nodePool.isEmpty():
24             # Pop best node from priority queue
25             currentNode = self._nodePool.pop()
26             if currentNode.getHScore() == 0:
27                 # Solution found!
28                 return currentNode.getMoves()
29             # Compute child nodes and add them to the queue
30             children = self._nodeBuilder.getChildNodes(currentNode)
31             for child in children:
32                 self._nodePool.add(child)
33         # No solution has been found
34         return None
35
36     def _bootstrap(self, position):
37         """
38         Create the initial Node from the given position.
39         Add the initial node to the queue.
40         """
41         node = Node(position, [], self._heuristic)
42         self._nodePool.add(node)
```

E' il cuore dell'algoritmo che realizza, tramite la sua funzione solve, la computazione. Si noti come esso termina quando, analizzando un nodo, la sua euristica è pari a zero, caso possibile solo se tutti i numeri sono nella posizione corretta ed è quindi stato raggiunto il goal.