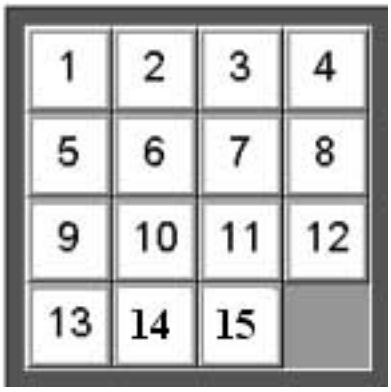
Homework 1
Intelligenza Artificiale

Ciavarro Cristina **253188**
Di Natale Marco **255660**



Abbiamo deciso di risolvere il gioco del 15 tramite un algoritmo di ricerca basato su una coda di priorità. Abbiamo rappresentato la griglia 4x4 contenente 16 caselle con un array avente 16 celle e, ai fini del gioco, le caselle sono etichettate con i numeri da 1 a 15, mentre la cella vuota è rappresentata con il numero "0". Con una singola mossa si può slittare una casella orizzontalmente (verso sinistra o destra) o verticalmente (su o giù), trasferendola nella cella vuota. Per risolvere tale problema abbiamo deciso di implementare come euristica quella basata sull'algoritmo *A** usando come funzione di priorità la *distanza di Manhattan*.

Best-first search e A*

A* è un algoritmo creato dall'unione degli algoritmi Depth-First Search (DFS) e Greedy. Infatti, come DFS, questo algoritmo costruisce un albero di tutti gli stati possibili che possono essere raggiunti dallo stato iniziale (cioè la board) e, da quegli stati, tutti gli stati figli, fino a raggiungere le foglie. Tuttavia, il numero di stati è esponenziale. Ciò implica che è anche esponenziale computazionalmente, cioè irrisolvibile in un tempo realistico. Qui è dove entra in gioco il Greedy. Invece di guardare tutti gli stati possibili ottenibili da un determinato stato, indovina quello più promettente. L'ipotesi è raggiunta con l'euristica. Ogni nodo n dell'albero è associato al valore: f(n) = g(n)+h(n) dove g (n) è il costo di raggiungimento dello stato n e h (n) è il valore stimato del raggiungimento della soluzione dal nodo n. L'euristica imposta il valore di h (n).

Infatti l'algoritmo implementato crea l'albero dove ogni nodo di ricerca è rappresentato da una configurazione della griglia ed il peso di ogni arco è il numero di mosse utilizzate per raggiungere tale configurazione.
All'inizio si inserisce nella coda di priorità il nodo iniziale (con la configurazione iniziale, 0 mosse effettuate, e un riferimento *null* come nodo predecessore).
Successivamente si rimuove dalla coda di priorità il nodo di ricerca a priorità minima, inserendo nella coda tutti i nodi adiacenti (ovvero quelli che possono essere raggiunti attraverso una singola mossa a partire dal nodo estratto dalla coda). Si ripete questa procedura fino a quando non si rimuove dalla coda il nodo corrispondente alla configurazione finale. Teniamo traccia delle visite effettuate nel grafo con un *orizzonte* che non è altro che l'insieme dei nodi che ci si è spinti a visitare fino a quel momento.
Tuttavia se si osserva la stampa a schermo del computer, si può notare che ai fini "fisici" del gioco certi passaggi non sono fattibili: perché in verità la stampa mostra il percorso che l'algoritmo sceglie nel grafo, tornando quindi anche indietro verso un altro punto del suo orizzonte se il percorso scelto fino a quel punto è divenuto più dispendioso con l'incontro di nuovi nodi. Questo è possibile perché abbiamo una visione del futuro parziale: possiamo prevedere tutte le mosse possibili da una data configurazione del puzzle, andando così avanti di una sola mossa che in quel momento è la più conveniente, ma non possiamo sapere da subito se quella scelta ci porterà in una strada che complessivamente peserà di meno di un'altra. Per questo capita spesso nell'algoritmo di fare un salto

indietro di alcune mosse per cambiare la scelta affrontata precedentemente.
Per decidere le priorità possiamo usare la funzione di priorità, *distanza di Manhattan*, che è data dalla somma delle distanze orizzontali o verticali (a seconda di quale spostamento è più conveniente) di ciascun blocco dalla sua posizione finale corretta. L'euristica scelta è la più usata per risolvere il gioco del quindici, ma se si desidera è possibile cambiarla togliendo cambiando la classe **Heuristic** implementata con qualsiasi altra funzione scelta. Il numero di mosse minimo da effettuare per risolvere il puzzle a partire da un dato nodo di ricerca (incluse quelle già effettuate) sarà maggiore o uguale alla sua priorità di Manhattan (dove la cella vuota non viene inclusa nel calcolo della distanza). Possiamo considerare il numero di mosse minimo che si ha con la configurazione iniziale del puzzle il lowerbound del problema. Il programma stamperà a schermo la stringa: *"Il numero minimo di passi per risolvere il problema è n "* e successivamente anche quante mosse l'algoritmo ha impiegato effettivamente per risolvere tale problema. Quindi, quando il nodo goal viene rimosso dalla coda di priorità, avremo il numero di mosse effettivo che l'algoritmo ha impiegato per raggiungerlo ed è dato esattamente dal suo valore di priorità.

Implementazione

```
class AStar:

    def __init__(self,heuristic,g):
        self.heuristic = heuristic
        self.list = LinkedList()
        self.game = g

    def solve(self,matrix):

        # Creo il nodo iniziale con la matrice nelle condizioni iniziali
        node = Node(matrix, [], self.heuristic)
        self.list.add(node)

        step=0
        # Scorro tutta la lista di nodi fino ad arrivare alla fine
        while not self.list.isEmpty():
            # Pop best node from priority queque
            currentNode = self.list.pop()

            # Stampiamo lo step
            self.game.getMatrix(step,currentNode.getMatrix())
            step = step+1

            # Verifico di essere arrivato sulla casella vuota
            if currentNode.getHScore() == 0:
                return currentNode.getMoves()
            # Compute child nodes and add them to the queue
            children = self.game.nextNode(currentNode)

            for child in children:
                self.list.add(child)

        return None
```

Classi implementate

- **AStar**: classe che implementa l'algoritmo A* come descritto sopra.
- **Heuristic**: classe dove risiede l'euristica del gioco che nel nostro caso è la distanza di Manhattan. È possibile tuttavia cambiare l'euristica del gioco con qualsiasi altra euristica a scelta.
- **LinkedList**: classe che crea la lista di nodi che comporranno il grafo, dove per nodi si intende una data configurazione del puzzle.
- **Game**: classe che gestisce la stampa di una matrice presente in un dato nodo.
- **Node**: classe che definisce la struttura di un nodo.

Stampa a schermo

Per quanto riguarda la stampa a schermo del programma, restituisce la matrice con tutti i movimenti del puzzle. Inoltre stampa il tempo impiegato per risolvere tale problema ed il minimo numero di mosse possibili per risolvere il puzzle dato. Ecco un esempio di 4 schermate di gioco: