Documentazione AI - 2 assignment

Piccirilli, Addario

3 novembre 2023

1 Problem solving as search

"Consider the problem of finding a path in the grid shown in the figure below from the position s to the position g. A piece can move on the grid horizontally or vertically, one square at a time. No step may be made into a forbidden black areas"

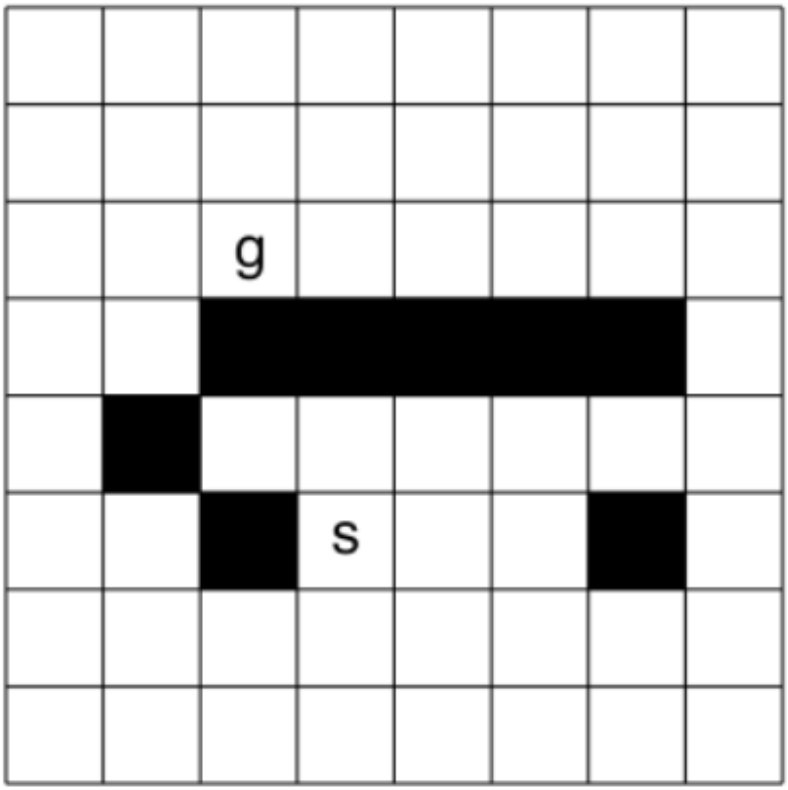

Figura 1: Immagine della griglia da esplorare

"On the grid shown above, number the nodes expanded (in order) for a depth-first search from s to g, given that the order of the operators is up, left, right, and down. Assume there is cycle pruning. What is the first path found"

Tra le opzioni presenti abbiamo scelto la prima, ossia una **visita in profondità** che presentasse un criterio di scelta per l'esplorazione diverso da quello comune del peso degli archi (dal momento che gli archi del grafo hanno tutti uguale peso). Infatti, come da consegna, l'esplorazione dei vicini deve avvenire prima per il vicino di sopra, se questo non fosse stato presente allora bisognava procedere con quello a sinistra, poi quello a destra e così anche per quello in basso.

2 Divide et impera

Dal momento che bisognava mostrare la visita sia sulla griglia sopra, sia su un grafo costruito ad hoc, abbiamo scelto di dividere i due concetti dedicandoci prima alla costruzione della griglia. Abbiamo scelto di usare il modulo **pyplot** di matplotlib. Inoltre, per rendere ancora più visibili le celle di partenza e di arrivo, queste sono colorate, rispettivamente, in verde e in blu; lo stesso è presente sul grafo.

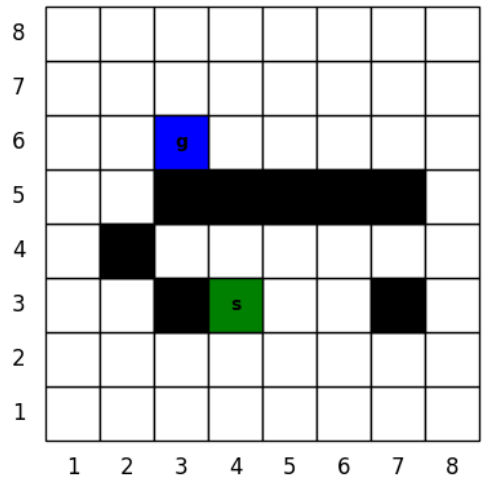

Figura 2: Immagine della griglia costruita da noi

Per la costruzione del grafo abbiamo scelto di usare la libreria **networkx** e questo è stato il risultato finale:

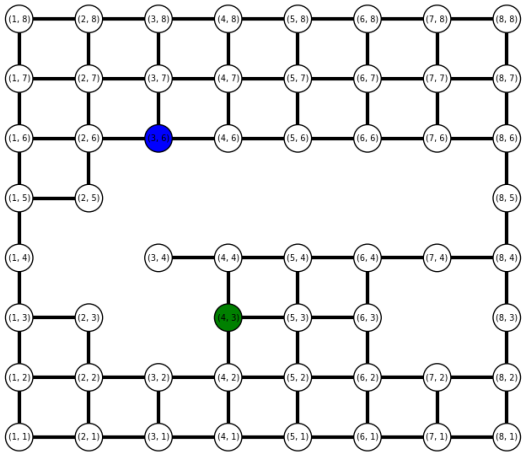

Figura 3: Immagine del grafo costruito da noi

Si noti che, per una gestione più facile dei nodi e dei suoi vicini, si è scelto di dare come nomi dei nodi la loro posizione nella forma **(x, y)**

2.1 Implementazione DFS

Per la visita in profondità abbiamo scelto di reimplementare il codice, sotto una sezione dello stesso:

```
    def dfs(self):

        visited = set()
        self.path.append(str(self.start))
        self.stack.put(str(self.start))
        print("Stack: ", self.stack.queue)
        print("Explored: ", self.path)

        node = self.stack.get()

        while node != str(self.goal):

            a = list(node)
            current_node_x, current_node_y = int(a[1]), int(a[4])

            neighbors = list(self.g.neighbors(f"({current_node_x}, {current_node_y})"))

            if node not in visited:
                visited.add(node)

            # Aggiungo allo stack i nodi mettendo in senso contrario all'esplorazione
            if f"({current_node_x}, {current_node_y - 1})" in neighbors:
                if f"({current_node_x}, {current_node_y - 1})" not in visited:
                    self.stack.put(f"({current_node_x}, {current_node_y - 1})")
            if f"({current_node_x + 1}, {current_node_y})" in neighbors:
                if f"({current_node_x + 1}, {current_node_y})" not in visited:
                    self.stack.put(f"({current_node_x + 1}, {current_node_y})")
            if f"({current_node_x - 1}, {current_node_y})" in neighbors:
                if f"({current_node_x - 1}, {current_node_y})" not in visited:
                    self.stack.put(f"({current_node_x - 1}, {current_node_y})")
            if f"({current_node_x}, {current_node_y + 1})" in neighbors:
                if f"({current_node_x}, {current_node_y + 1})" not in visited:
                    self.stack.put(f"({current_node_x}, {current_node_y + 1})")

            print("Stack: ", self.stack.queue)

            if all(map(lambda x: x in visited, list(self.g.neighbors(f"({current_node_x}, {current_node_y})")))):
                for node in list(self.g.neighbors(f"({current_node_x}, {current_node_y})")):
```

```
                    self.path.append(node)

            node = self.stack.get()
            self.path.append(node)
            print("Explored: ", self.path)
```

La visita è gestita esplicitamente con uno stack (**stack**) da cui si fa il pop ad ogni iterazione. Inoltre, per tenere traccia dei nodi visitati, utilizziamo un set chiamato **visited**. Il ciclo while va avanti finchè non arriva al nodo goal e procede a mettere nello stack i nodi da vistare in ordine contrario a quello stabilito, cosicchè, in cima allo stack ci sia, se esista, il nodo al di sopra di quello esplorato, poi sotto quello a sinistra e così via. Infine, sono presente alcuni controlli per evitare che looppi in determinate condizioni. Il percorso ottenuto viene salvato nella variabile di classe **path** e richiamato, attraverso un generatore, un nodo per volta come nel codice che segue:

```
    def get_next_node(self):
        for node in self.path:
            yield node, list(self.g.neighbors(node))
```

Si noti, in questo caso, che non solo torna il nodo da esplorare, ma anche i suoi vicini per andare a costruire la frontiera un passo per volta.

Nel **main.py** è presente la dichiarazione della griglia e del grafo e attraverso un ciclo, che termina quando il generatore non ha più elementi da restituire, vengono colorati passo per passo i nodi esplorati e la frontiera.