INTRODUZIONE ALL'INTELLIGENZA ARTIFICIALE A.A. 2022/2023

Andrea Cappone

Matricola 869012

Università degli Studi di Milano Bicocca

PROBLEMA

Consideriamo il problema dello spostamento di k cavalieri degli scacchi da k caselle di partenza $s_1, \dots, s_k$ a k caselle obiettivo $g_1, \dots, g_k$ su una scacchiera NxN, soggetta alla regola che due cavalli non possono occupare contemporaneamente la stessa casa. Ogni azione permette di muovere fino a m cavalieri contemporaneamente. L'obiettivo è completare la manovra nel minor numero di azioni.

DIMENSIONE DELLO SPAZIO DEGLI STATI IN FUNZIONE DI N E M

Il problema del trasferimento di k cavalieri da k posizioni di partenza a k posizioni obiettivo su una scacchiera NxN può essere formulato come un problema di ricerca nello spazio degli stati. La dimensione dello spazio degli stati dipende dal numero di possibili posizioni di partenza dei cavalieri e dal numero di possibili azioni per ogni stato. In particolare, ci sono $\binom{N^2}{k}$ possibili posizioni di partenza per k cavalieri su una scacchiera NxN e ogni stato può generare fino a m^k nuovi stati.

Quindi, la dimensione totale dello spazio degli stati è pari a: $\binom{N^2}{k} * m^k$

MASSIMO FATTORE DI RAMIFICAZIONE, ESPRESSO IN FUNZIONE DI K

In ogni stato, ogni cavaliere ha al massimo 8 possibili mosse.

Quindi il massimo fattore di ramificazione in questo spazio degli stati è pari a 8^k

DEFINIRE DELLE EURISTICHE AMMISSIBILI

1

DISTANZA DI MANHATTAN

Stima la distanza tra la posizione attuale di ogni cavallo e la sua posizione obiettivo come la somma delle distanze orizzontali e verticali tra le due posizioni

2

DISTANZA DI CHEBYSHEV

Stima la distanza tra la posizione attuale di ogni cavallo e la sua posizione obiettivo come la massima delle distanze orizzontali e verticali tra le due posizioni

3

EURISTICA DI CLUSTERING

Raggruppa i cavalieri in cluster in modo che ogni cluster abbia almeno un cavallo che è già nella posizione obiettivo. Quindi, l'euristica stima il numero di cluster che devono ancora essere spostati

ALGORITMI IMPLEMENTATI

Per problemi legati alla gestione dello spazio e della memoria gli algoritmi implementati non tengono conto del numero m di cavalli spostati contemporaneamente, ma considerano ogni cavallo singolarmente

BFS | RICERCA IN AMPIEZZA

Algoritmo di ricerca che esplora tutti i nodi adiacenti di un nodo prima di muoversi in profondità. In altre parole, esplora i nodi livello per livello e garantisce che la soluzione ottimale sia trovata solo quando la soluzione corrente ha il minor numero di mosse possibili.

Tuttavia, BFS può richiedere molta memoria per tenere traccia dei nodi visitati in quanto deve mantenere una coda dei nodi visitati.

SPIEGAZIONE

Prende in input i seguenti parametri:

$start$: tupla contenente le posizioni di partenza dei cavalieri nella forma $((r_1, c_1), (r_2, c_2), \dots, (r_k, c_k))$

$goal$: tupla contenente le posizioni di arrivo dei cavalieri nella forma $((r_1, c_1), (r_2, c_2), \dots, (r_k, c_k))$

k: numero di cavalieri da spostare

N: dimensione della scacchiera

INPUT

Restituisce il percorso ottimo per spostare i cavalieri dalla posizione di partenza alla posizione di arrivo, rappresentato come una lista di mosse.

Ogni mossa è una tupla nella forma $(i, [(r_1, c_1), (r_2, c_2), \dots, (r_z, c_z)])$ dove i indica l'indice del cavallo che viene spostato e (r, c) indica la nuova posizione del cavallo.

OUTPUT

BFS | RICERCA IN AMPIEZZA

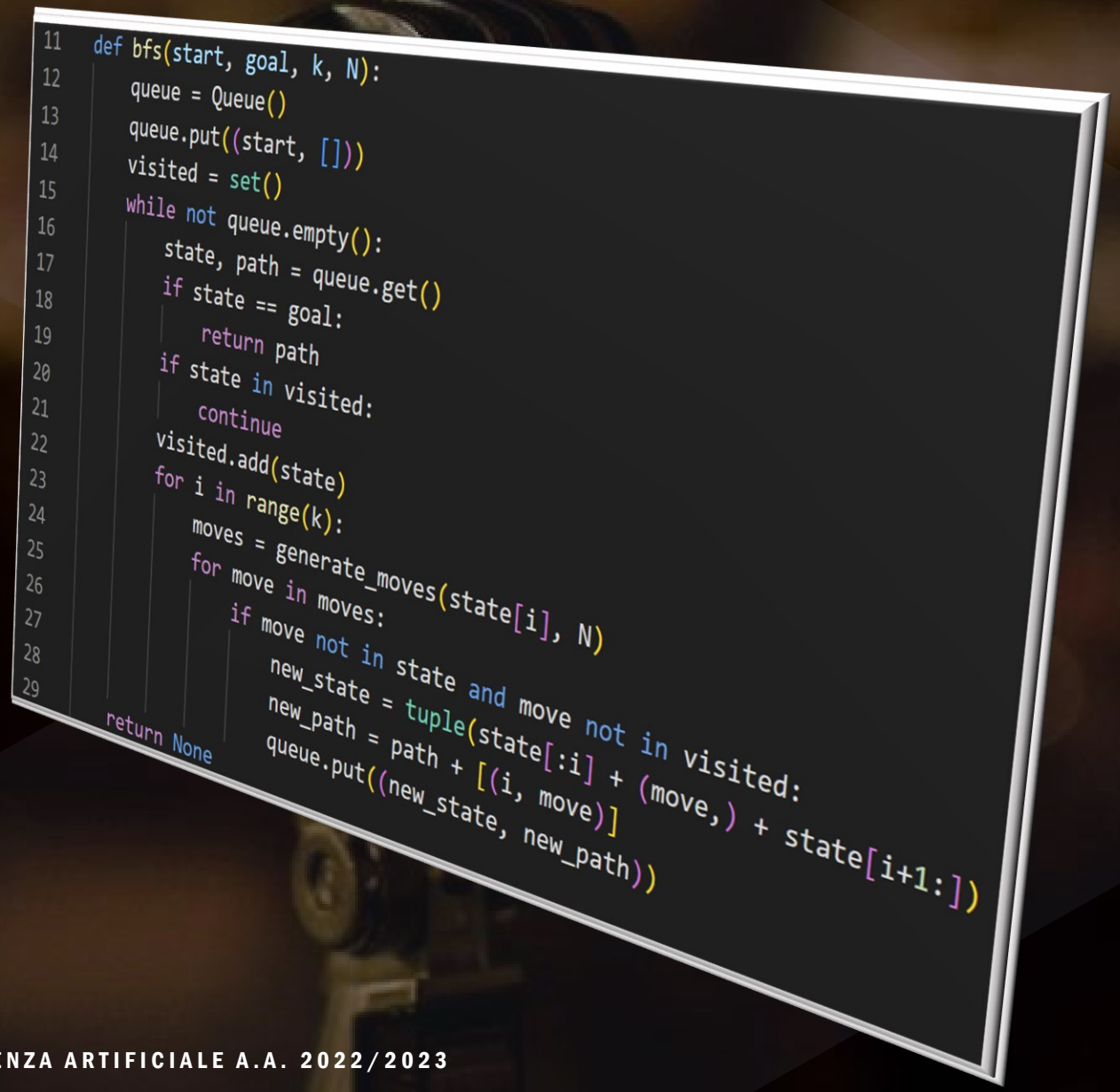
L'algoritmo utilizza una coda per memorizzare i successivi stati da esplorare.

Inizialmente, il punto di partenza viene inserito nella coda insieme al percorso vuoto.

Ad ogni iterazione, l'algoritmo estrae lo stato dalla testa della coda e lo espande generando tutti i possibili nuovi stati che possono essere raggiunti.

Per ogni nuovo stato generato, l'algoritmo controlla se è lo stato obiettivo e, in caso contrario, lo inserisce nella coda insieme al percorso fino a quel punto.

In questo modo, l'algoritmo esplora tutti gli stati possibili nello spazio degli stati in modo sistematico, partendo dallo stato iniziale e muovendosi verso gli stati successivi in ordine di distanza dalla radice.



```
def bfs(start, goal, k, N):
    queue = Queue()
    queue.put((start, []))
    visited = set()
    while not queue.empty():
        state, path = queue.get()
        if state == goal:
            return path
        if state in visited:
            continue
        visited.add(state)
        for i in range(k):
            moves = generate_moves(state[i], N)
            for move in moves:
                if move not in state and move not in visited:
                    new_state = tuple(state[:i] + (move,) + state[i+1:])
                    new_path = path + [(i, move)]
                    queue.put((new_state, new_path))
    return None
```

ALGORITMO A*

Algoritmo di ricerca informata che combina l'approccio di BFS con una funzione euristica per migliorarne l'efficienza.

Calcola un'euristica del costo dallo stato corrente allo stato finale e usa questa informazione per guidare l'algoritmo nella direzione giusta.

A* garantisce di trovare la soluzione ottimale se la funzione euristica è ammissibile e consistentemente stimabile.

A differenza di BFS e DFS, A* può essere più efficiente in termini di tempo e spazio.

SPIEGAZIONE

Prende in input i seguenti parametri:

$start$: tupla contenente le posizioni di partenza dei cavalieri nella forma $((r_1, c_1), (r_2, c_2), \dots, (r_k, c_k))$

$goal$: tupla contenente le posizioni di arrivo dei cavalieri nella forma $((r_1, c_1), (r_2, c_2), \dots, (r_k, c_k))$

k: numero di cavalieri da spostare

N: dimensione della scacchiera

INPUT

Utilizza una coda di priorità per tenere traccia degli stati che devono essere esplorati, e la funzione $heuristic$ per calcolare un'euristica ammissibile.

Come funzione $heuristic$ sono state definite:

- Distanza di Manhattan
- Distanza di Chebyshev

OUTPUT

ALGORITMO A*

Il ciclo principale dell'algoritmo estrae lo stato con il percorso $path$ minore dalla coda di priorità, lo esamina e genera tutti i successori.

Per ogni successore, l'algoritmo calcola il percorso new_path e lo inserisce nella coda di priorità. Se uno stato è già presente nella coda di priorità con un percorso inferiore, non viene inserito di nuovo.

Infine, se la coda di priorità si svuota senza trovare una soluzione, la funzione restituisce None, che identifica un percorso vuoto.

```
11  def a_star(start, goal, k, N, heuristic=manhattan_distance):
12      open_set = [(heuristic(start, goal, k), start, [])]
13      closed_set = set()
14      while open_set:
15          f, state, path = heapq.heappop(open_set)
16          if state in closed_set:
17              continue
18          if state == goal:
19              return path
20          closed_set.add(state)
21          for i in range(k):
22              moves = generate_moves(state[i], N)
23              for move in moves:
24                  if move in state:
25                      continue
26                  new_state = tuple(state[:i] + (move,) + state[i+1:])
27                  new_path = path + [(i, move)]
28                  new_f = len(new_path) + heuristic(new_state, goal, k)
29                  heapq.heappush(open_set, (new_f, new_state, new_path))
30      return None
```

DFS | RICERCA IN PROFONDITÀ

Algoritmo di ricerca che esplora un ramo del grafo il più possibile prima di tornare indietro e esplorare il prossimo ramo. In altre parole, esplora il grafo in profondità prima di tornare indietro.

A differenza di BFS, DFS non garantisce di trovare la soluzione ottimale, ma può essere più efficiente in termini di memoria.

SPIEGAZIONE

```
def dfs(k, positions, targets, N, visited):
    if k == len(positions):
        return []
    moves = []
    for i, pos in enumerate(positions):
        if i not in visited:
            visited.add(i)
            for move in generate_moves(pos, N):
                if move == targets[k]:
                    new_positions = positions[:i] + (move,) + positions[i+1:]
                    res = dfs(k+1, new_positions, targets, N, visited)
                    if res is not None:
                        moves.append((i, move))
                        moves.extend(res)
            visited.remove(i)
    return moves
```

Ho avuto diversi problemi di implementazione. Sul file dfs.py si trova una versione non completamente funzionante.

Nello specifico ritorna solo percorsi parziali, e spesso il valore di ritorno è "None", come se non esistesse una soluzione al problema.

NOTE

FUNZIONE GENERATE_MOVES

Prende in input i seguenti parametri:

pos: una tupla rappresentante la posizione attuale del cavallo nella forma (row, col)

N: dimensione della scacchiera

INPUT

```
11  def generate_moves(pos, N):
12      row, col = pos
13      moves = set()
14      for dr, dc in [(2, 1), (1, 2), (-1, 2), (-2, 1),
15                     (-2, -1), (-1, -2), (1, -2), (2, -1)]:
16          r, c = row + dr, col + dc
17          if 0 <= r < N and 0 <= c < N:
18              moves.add((r, c))
19      return moves
```

La funzione itera su tutti i possibili movimenti del cavallo. Per ogni movimento, viene calcolata la posizione della casella di destinazione e, se questa si trova all'interno della scacchiera, la casella di destinazione viene aggiunta all'insieme $moves$.

La funzione restituisce un insieme di tutte le caselle raggiungibili dal cavallo in una scacchiera NxN partendo dalla posizione pos.

RIEPILOGO

Diversi algoritmi forniscono diversi output

Tempi di esecuzione per BFS molto più elevati di A*

I percorsi generati sono diversi, ma tutti arrivano alla soluzione con lo stesso numero di mosse. Questo è dovuto all'ordine di esplorazione di tutte le possibili mosse di ogni cavallo per ogni step