Francesco Gradi

07/05/2018

Implementazione del Dominio 2 (NailBox)

1. Descrizione del problema

Nel dominio in questione sono presenti un numero indefinito di oggetti di tipo scatola, coperchio, chiodo disposti in pile qualsiasi in un minimo di sei postazioni a disposizione. Per poterli spostare si ha a disposizione un braccio meccanico che può eseguire delle operazioni su di essi, senza spostarsi:

- ***get***: il braccio prende in mano l'oggetto (ammesso che sia libero e nella postazione in questione ci sia effettivamente un oggetto).
- ***put***: il braccio deposita l'oggetto (ammesso che ne abbia uno in mano) in cima ad una postazione o pila di oggetti.
- ***hammer***: il braccio unisce tre oggetti insieme e da questo momento in poi li tratta come un oggetto unico (ammesso che il braccio sia libero e che gli oggetti siano nella sequenza dall'alto chiodo/coperchio/scatola).

Ognuna delle quali ha costo di esecuzione pari a 1.

L'obiettivo è, partendo da una disposizione qualsiasi di oggetti in numero qualsiasi, ottenere un'altra disposizione qualsiasi degli stessi oggetti.

2. Implementazione

È stato necessario usufruire di particolari strutture dati quali le tuple, nel mio caso si vedrà un'implementazione che sfrutta l'algortimo A*. Con questo tipo di struttura dati è possibile tuttavia usufruire anche di BFS o di DLS.
Il problema è stato così rappresentato:

```
state = ((0, 0), (1,0), (2, 'a'), (3, 'b'), (4, 'c'), (5, 0), (6, 0))
```

Dove ogni tupla (eccetto la prima) corrisponde ad una postazione e ogni primo elemento al numero della postazione stessa, che chiameremo *indice della postazione*. Per elemento 'a' si intende l'oggetto di tipo scatola, per 'b' l'oggetto di tipo coperchio e per 'c' l'oggetto di tipo chiodo. L'elemento 0 sta invece a significare che la postazione è vuota.
L'ordine rappresentato degli oggetti in ogni pila (da sinistra verso destra) corrisponde all'ordine effettivo dall'alto verso il basso.
La prima tupla inoltre è riservata allo stato del braccio che, non dovendo spostarsi, non necessita di memorizzare la postazione sopra la quale si trova. Di conseguenza il primo elemento della prima tupla sarà sempre 0, rappresentando così l'*indice del braccio*. Il suo secondo elemento invece sarà rispettivamente 0 se il braccio è libero, oppure sarà settato con l'oggetto che ha preso in mano (quindi 'a', 'b', 'c', oppure 'cba' nel caso particolare in cui sia stata effettuata un'operazione di tipo *hammer*).
Il codice è così composto (*NailBox.py*):

```
def init():
    statoIniziale = ((0, 0), (1, 0), (2, 'a'), (3, 'b'), (4, 'c'), (5, 0), (6, 0))
    problema = NailBox(statoIniziale)
    alg = astar_search(problema)
    pp = alg.path()
    soluzione = alg.solution()
    for i in range(1, len(pp)):
        stato = pp[i].state
        costo = pp[i].path_cost
        azione = soluzione[i-1]
        print(str(azione) + "\t\t\t" + str(stato) + "\t\t\t" + str(costo))

init()
```

Le funzioni principali in *search.py*, quali **actions()**, **result()** e **h()** sono state così implementate (pagina seguente):

La funzione **actions()**:

```
def actions(self, state):
    possible_actions = ['Get', 'Put', 'Hammer']

    vett = []
    for i in state:
        vett.append(list(i))

    for posiz in range(1, len(vett)):
        if(vett[0][1] == 0):
            #se il braccio è libero:
            possible_actions.remove('Put')
            for posiz2 in range(1, len(vett[posiz])):
                if(posiz2>=3):
                    #se nella posizione corrente ci sono 3 o piu elementi:
                    if(vett[posiz][1] != c or vett[posiz][2] != 'b' or vett[posiz][3] != 'a'):
                        #se i 3 elementi non sono in sequenza chiodo, coperchio, scatola:
                        possible_actions.remove('Hammer')
                else:
                    possible_actions.remove('Hammer')
        else:
            #se il braccio è occupato:
            possible_actions.remove('Get')
            possible_actions.remove('Hammer')

    return possible_actions
```

Inizialmente è stato necessario convertire le tuple in una lista di liste, per poterle meglio gestire. Successivamente verranno ri-convertite in tuple per potervi applicare l'algoritmo.
In primis si controlla se il braccio sia libero o meno e, in caso sia libero, viene rimossa l'azione 'Put' dall'elenco di azioni possibili. Nel caso poi in cui la postazione non ha più di 3 elementi impilati sopra di essa oppure avendoli non si presentano nell'ordine chiodo/coperchio/scatola, viene rimossa l'azione 'Hammer'.
Altrimenti se il braccio è occupato vengono direttamente rimosse le azioni 'Get' e 'Hammer'.

La funzione **result()**, a pagina seguente:

```
def result(self, state, action):

    nuovoStato = []
    for i in range(1, len(state)):
        nuovoStato.append((list(i)))

    vett = []
    for j in state:
        vett.append(list(j))


    if(action == 'Get'):
        for posiz in range(1, len(vett)):
            if(vett[posiz][1] != 0):
                vett[0][1] = vett[posiz][1] #setta la posizione 1 del braccio con l'oggetto che ha gettato
                self.lastPos = posiz
                del vett[posiz][1]
                if(len(vett[posiz]) == 1):
                    #se ha gettato l'unico oggetto nella postazione:
                    vett[posiz].append(0)
                break


    if(action == 'Put'):
        vett[self.lastPos + 1].insert(1, vett[0][1]) #fa la put dell'oggetto nella posizione successiva in cui l'ha gettato

    if(action == 'Hammer'):
        for posiz in range(1, len(vett)):
            for posiz2 in range(1, len(vett[posiz])):
                if(posiz2 >= 3):
                    if(vett[posiz][1] == 'c' and vett[posiz][2] == 'b' and vett[posiz][3] == 'a'):
                        del vett[posiz][1]
                        del vett[posiz][2]
                        del vett[posiz][3]
                        vett[posiz].insert(1, 'cba')


    nuovaTupla = []

    for index in vett:
        nuovaTupla.append(tuple(index))

    return tuple(nuovaTupla)
```

Nel caso in cui venga eseguita l'azione 'Get', il secondo elemento della prima tupla (N.B: la prima tupla rappresenta sempre lo stato del braccio) viene riempito con l'oggetto che è stato preso dal braccio meccanico, viene rimosso l'oggetto in questione dalla tupla della postazione e, nel caso in cui sia stato l'ultimo oggetto presente sulla postazione, viene aggiunto uno 0 al suo posto (ad indicare che la postazione è rimasta vuota).
Nel caso in cui invece venga eseguita le 'Put', l'oggetto che il braccio attualmente ha viene inserito nella postazione successiva da cui era stato preso. Il secondo elemento della prima tupla viene poi settato nuovamente a 0, ad indicare che il braccio è di nuovo libero.
Nel caso infine in cui venga eseguita la 'Hammer' (quindi nel caso in cui ci siano 3 oggetti o più nella postazione nell'ordine chiodo/coperchio/scatola), vengono rimossi i primi 3 oggetti dalla tupla e vengono sostituiti con un nuovo oggetto ('cba') che sta a rappresentare i 3 oggetti in questione uniti tra di loro.

La funzione **h():**

```
def h(self, node):

    oraS = node.state
    goalG = self.goal
    dissimilarity = 0

    vett = []
    for i in oraS:
        vett.append(list(i))

    vett2 = []
    for j in goalG:
        vett2.append(list(j))

    for posiz in range(0, len(oraS)-1):
        if (oraS[posiz] != goalG[posiz]):
            dissimilarity += 1

    return dissimilarity
```

Questa funzione sta invece a rappresentare l'euristica usata: in essa vengono confrontati lo stato goal con lo stato iniziale e viene elaborato il loro coefficiente di disuguaglianza.