Documentazione di progetto Sistemi Intelligenti

Dipartimento di ingegneria dell'informazione,
Corso di laurea triennale: Ingegneria delle Tecnologie per l'impresa digitale,
Corso: Sistemi Intelligenti, User Experience e Robotica

Causetti Thomas	Gandinelli Paolo	Rossi Mirko
m. 731077	m. 732803	m. 731078

A.A. 2022/2023

Introduction

L'obiettivo di questo progetto è la risoluzione del problema della Torre di Hanoi nella variante bicolore. Vengono utilizzate due metodologie per la risoluzione: algoritmi di search specifici per il problema e la realizzazione di un modello del problema scritto in PDDL.

Questo progetto mira a esplorare e confrontare questi due metodi/approcci di risoluzione del problema della Torre di Hanoi, fornendo un'analisi approfondita dei vantaggi e delle limitazioni di ciascun approccio.

1 Search

1.1 Rappresentazione delle mosse

Le mosse effettuabili allo stato i-esimo vengono rappresentate mediante una matrice quadrata ($num_{rods} \times num_{rods}$), dove l'entrata (i, j) è uguale a 1 se è possibile muovere un disco dal paletto i al paletto j, altrimenti è posta a zero. (i, j) viene posto a zero se:

- $i = j$: per evitare mosse nulle.
- il primo disco del rod i ha una dimensione inferiore a quello del rod j.
- se *rodi* e *rodj* hanno entrambi un unico disco ed entrambi questi dischi sono quelli di dimensione massima: impedisce le mosse che portano i due dischi di dimensione massima ad essere uno sopra l'altro, essendo sicuramente non ottime.

Il calcolo della matrice per ogni nodo viene ottenuta copiando e modificando una matrice precalcolata in World contenente tutti 1 tranne sulla diagonale principale.

1.1.1 Calcolo della matrice

La matrice viene calcolata ogni volta che un nodo deve essere espanso, per capire quali mosse possono essere effettuate partendo dallo stato associato a tale nodo. Tale calcolo è implementato nei metodi "ap" e "apRulesCalculation" contenuti nella classe State, invocati nel metodo "expand" (in Solver) che viene richiamato dal metodo "solve" di Solver (metodo che implementa l'algoritmo di search per l'esplorazione dell'albero). La matrice non viene visitata nella sua interezza, si è optato per scorrere solo la matrice triangolare superiore in modo da incrementare l'efficienza. Infine il metodo "ap" è stato implementato in Cython nel file State_C1, metodo "aP_C" richiamato in state con "aP_C_Caller".

1.2 Classi principali

Rod: classe le cui istanze rappresentano la configurazione di uno dei paletti all'interno del gioco. Tale configurazione è modellata con una lista di tuple. Ogni tupla ([dimensione, colore]) rappresenta univocamente uno dei dischi utilizzati. Per le dimensione dei dischi si usano numeri incrementali da 1 a numero dischi per rod.

La classe State rappresenta gli stati del gioco. Ha come attributi: una lista di istanze delle classe Rod, e g numero di mosse(costo) per arrivare a tale stato, partendo dallo stato iniziale.

Node: classe usata per modellare i nodi dell'albero di search. Ha come attributi: un oggetto azione che indica la mossa effettuata per arrivare a tale stato, uno stato del gioco, il numero di mosse (g) effettuate partendo dallo stato iniziale, il valore di euristica (e) per lo stato associato al nodo e il riferimento al nodo padre.

World: classe utilizzata per rappresentare il contesto del gioco. Ogni istanza contiene stato iniziale e finale del problema.

Execution: classe in cui viene avviata la risoluzione ed estratta la soluzione dal nodo goal.

MainWindow e MainWindowController: la prima classe gestisce la presetazione a video della GUI e la seconda gli eventi della prima.

1.3 Algoritmi di esplorazione

Come algoritmi di esplorazione dell'albero sono stati implementati Astar in quanto grantisce l'ottimo con euristica ammissibile e Bfs perché essendo un problema con costo unitario anche Bfs trova l'ottimo e quindi è possibile mostrare l'efficacia dell'euristica.

Breadth first search

L'algoritmo esplora l'albero di ricerca in ampiezza, espandendo tutti i nodi di un livello prima di passare al successivo. Vengono utilizzate tre collezioni di elementi per implementarlo:

- frontier: coda FIFO che mantiene memorizzati i nodi generati e ancora da espandere.
- reached: set che mantiene memorizzati tutti gli stati che sono già stati raggiunti, ossia stati associati a nodi già espansi (senza ripetizione).
- nods_in_frontier: set aggiornato parallelamente alla frontier ed utilizzato per controllare in modo più efficiente se uno nodo è contenuto in frontier.

L'algoritmo si compone di un ciclo che ha come condizione di terminazione lo svuotamento di frontier o il raggiungimento del nodo goal. All'interno di tale ciclo ci sono due sottoparti principali:

- espansione: viene estratto da frontier un nodo n con logica FIFO, si aggiunge n a *reached* e vengono generati nuovi nodi foglia con n come nodo padre, mediante la chiamata alla funzione "expand".
- generazione: dei nodi generati si scartano quelli con associato uno stato già contenuto in *reached* o in nods_in_frontier (nella frontiera). Per i rimanenti si controlla se abbiano associato lo stato goal: se sì viene ritornato tale nodo, se no si aggiunge il nodo a frontier e a nods_in_frontier.

A star

Utilizzate di due collezioni di elementi: reached (uguale a BFS) e frontier. In questo caso frontier opera come una priority queue che ordina i nodi in modo crescente in base al valore di: $g + euristica$. La priority queue è implementata dalla classe OpenList le cui istanze contengono: una lista ordinata di tuple $[node.g + node.e, node]$ in modo crescente in base al valore node.g + node.e, implementata con il modulo heap di python; un hashtable con chiavi gli oggetti nodi e valori tuple $[node.g + node.e, node]$.
Avendo come hash degli oggetti node l'hash di una string associata in modo univoco allo stato realativo a tale nodo, e non potendoci essere due nodi con lo stesso stato nella frontiera, l'hashtable permette di verificare se uno stato è contenuto nella frontiera con complessità O(1), rispetto al worst case O(n) che si avrebbe dovendo scorrere tutta la

lista ordinata di nodi.
Le tuple conenute nella lista e nel dizionario sono riferimenti allo stesso oggetto, questo permette di efficientare a livello di tempo la procedura di rimozione: si esegue il pop della tupla da dizionario, si modifica l'oggetto node della tupla con un segna posto, in modo che successivamente quella tupla sarà sempre l'ultima della lista e verrà sempre ignorata nei confronti per l'ordinamento. Non rimuovere fisicamente l'oggetto dalla lista fa in modo che non sia necessario eseguire uno spostamento di oggetti all'interno di quest'ultima.

Prima Euristica (Heuristic3)

Per ogni rod, se l'i-esimo disco non coincide con quello nello stato goal, si aggiunge al totale un numero di mosse pari a: dimensione del disco se questa è maggiore (numero di rod - 1), altrimenti 1 mossa.
L'euristica è ammissibile perchè: se un disco di dimensione m è in posizione sbagliata per raggiungere il goal servirà almeno una mosse per portarlo nella posizione corretta e altre m-1 mosse per i dischi sopra di esso, supponendo di poter sempre spostare un disco alla posizione goal con una mossa diretta e senza considerare che vanno rispettate le combinazioni di colori tra i dischi.
Inoltre l'euristica considera le configurazioni dei paletti in cui i dischi di dimensione minore a (numero rod - 1) possano arrivare alla posizione goal in una sola mossa.

Seconda Euristica (Heuristic4)

Per il primo sottoproblema della risoluzione in multiprocessing, sia 2 sia 3 processi, viene utilizzata una seconda euristica. Per ogni disco sul terzo rod si aggiunge una mossa, per il primo e secondo rod: trovato il primo disco errato partendo dal fondo, si aggiunge un numero di mosse pari a: (1 + num di dischi sul disco errato).
I dischi sul terzo rod richiedono sicuramente almeno una mossa per raggiungere la posizione goal, e dato un disco errato sui primi due rod bisognerà almeno spostare i dischi al di sopra e sopstare il disco considerato per portarlo in posizione goal.

All'interno del programma, nel calcolo dell'euristica, si moltiplica la quantità "num di dischi sul disco errato" per due, in quanto mantiene l'ottimalità riducendo di alcune miglialia il numero di nodi espansi. Non si ha però una prova teorica dell'ammissibilità dell'euristica con quest'ultima aggiunta se non il fatto che A-star trovi la soluzione ottima.

2 PDDL

Il Planning Domain Definition Language (PDDL) è il linguaggio standard per la rappresentazione di AI planning problems, tale approccio si differenzia dalle tecniche di search perchè descrive in modo formale e dichiarativo, mediante la creazione di un modello, il problema da risolvere. Grazie a tale descrizione è possibile ottenere una soluzione per il problema utilizzando un solver generico che non dipende da una conoscenza pregresse sul problema specifico.

La sintassi di PDDL, ispirata a LISP, favorisce la separazione degli operatori dagli oggetti, condizioni iniziali e obiettivi. L'istanziazione comporta la sostituzione degli oggetti specifici del problema alle variabili nei modelli di azioni e fatti, compresa la tipizzazione delle variabili. I fatti istanziati sono proposizioni logiche costruite con predicati e termini, mentre precondizioni ed effetti delle azioni istanziate sono formule logiche combinate con connettivi logici.
Questa programmazione, a differenza dell'approccio basato sulla metodologia di ricerca, si concentra sulla struttura intrinseca del problema piuttosto che sugli algoritmi utilizzati per risolverlo.

Per il progetto si è realizzato uno script python (pythonWriterPLUS/WritePDDL.py) che genera in modo automatico il file problema PDDL noto il numero di dischi e dei rod: partendo da un template di base lo adatta al numero di dischi e rod dato in input. Il file problema PDDL generato viene risolto all'ottimo mediante invocazione tramite Python dell'apposito comando per il solver enhsp-20.

2.1 File di dominio

Il modello del problema della Torre di Hanoi in PDDL coinvolge tre tipi di oggetti:

- Le "Rod" (Aste o Pali): Questi rappresentano le strutture verticali su cui i dischi vengono posizionati. Nel contesto del problema, ci sono tre aste come caso di default, ma possono essere aggiunte per complessificare il problema.

- I "Disk" (Dischi): Questi rappresentano i dischi di diverse dimensioni che devono essere spostati da un palo all'altro.
- Un Oggetto "Hand" (Mano o Meccanismo): Questo oggetto rappresenta il mezzo attraverso il quale i dischi vengono spostati da un palo all'altro. La "mano" è responsabile di garantire che i dischi siano spostati correttamente e rispettando le regole del problema della Torre di Hanoi Bicolore.

Vengono definiti i seguenti predicati:

- on(disco1, disco2): il disco1 è sopra a disco2
- smaller(d1, d2)/ equal(d1, d2): il disco d1 è rispettivamente più piccolo e uguale a d2
- clear(d1): il disco d1 non ha nessun disco sopra di esso
- located(d1, r): il disco d1 è posizionato sul rod r
- handfull(h): vero se la "mano" è occupata, ossia è appena dopo aver preso un disco da un rod
- toAllocate(d1) : vero se il disco d1 è stato parametro dell'azione "take block conditional"

Vengono definite tre azioni chiave:

- Azione "take block conditional": Questa azione rappresenta il prendere in mano un disco dalla cima di uno dei rod non vuoti.
- Azione "put block on block": Questa azione permette di posizione un disco precedentemente preso su uno degli altri rod, assicurandosi che un disco più piccolo non venga mai posizionato sopra uno più grande.
- Azione "put block on empty": Questa azione consente di porre un disco, precedentemente preso, su uno dei rod vuoti

2.2 File problema

Vengono inizializzati gli oggetti necessari a modellare il gioco: n oggetti disk per ognuno dei due colori, m rod e un oggetto di tipo hand. Ogni oggetto disco ha un nome univoco formato dalla concatenzazione di "nome colore + dimensione".

Successivamente si definiscono stato iniziale e goal:

- Stato Iniziale:

  I primi due paletti ciascuno con una torre da n dischi, uno per dimensione, disposti a colore alternato e gli altri rod vuoti.

- Stato Obiettivo (Goal):

  Due torri monocromatiche sugli stessi paletti di partenza, gli altri rod vuoti. I dischi alla base delle due torri nello stato iniziale sono invertiti di posizione.

3 Esplorazione dell'albero con multiprocessing

Nell'analisi del problema sono stati individuati due subgoal intermedi comuni alle risoluzioni del gioco per ogni numero di dischi. Tali subgoal vengono utilizzati sia nella Search sia in PDDL (solo il primo in questo caso) per costruire dei sottoproblemi che risolti in parallelo permettono di ottenere una soluzione ottima con complessivamente minor tempo e numero di nodi esplorati; in quanto ogni sottoproblema comporta l'esplorazione di un albero di dimensioni inferiori rispetto a quello del problema originale.

Per l'implementazione in Python si è utilizzato il modulo "multiprocessing" che permette l'elaborazione parallela su più core logici, invece di creare molteplici thread su un singolo core logico.
Come nominato prima, i sottoproblemi utilizzano euristiche differenti, infatti è possibile passare come parametro al metodo "solve" di Solver.py il valore '4' per utilizzare l'euristica descritta nella sezione "Seconda euristica" in 1.3

Bibliografia

[1] P. N. e Stuart J. Russell, *Artificial intelligence a modern approach*. Pearson, 2016.

Sitografia

[2] «Enhsp-20.» (), indirizzo: `https://gitlab.com/enricos83/ENHSP-Public/-/tree/enhsp-20`.

[3] «Indexed priority queue.» (), indirizzo: `https://www.geeksforgeeks.org/indexed-priority-queue-with-implementation/`.