A* Con Disjoint Pattern Databases

Femia Davide

Indice

1 Introduzione

In questo esercizio si parte da un'implementazione dell'algoritmo A* per poi implementare dunque le euristiche descritte in (Korf and Felner, 2002) tentando di riprodurre i risultati riportati in Table 1 di tale articolo.

2 Descrizione Esperimenti

2.1 Dati e Misure

Gli stati iniziali da cui iniziare la ricerca verso il goal con le varie euristiche vengono generati casualmente attraverso il metodo randomTable della classe TilesProblem che a sua volta utilizza la funzione sample, della libreria random di python, sull'insieme delle azioni possibili su uno stato per scegliere quale azione applicare; vengono effettuate 100 azioni di default a partire dallo stato goal(il parametro 100 può essere modificato attraverso il costruttore della classe TilesProblem).

Le misure vengono effettuate all'interno del metodo AStar della classe RandomAStarProblem e vengono memorizzate in alcuni suoi attributi. Il metodo getResults della classe RandomAStarProblem mostra a video le misure effettuate.

I secondi vengono misurati grazie alla funzione default_timer della libreria timeit. Il numero di nodi generati vengono misurati invece grazie alle dimensioni delle strutture dati che vengono utilizzate nel metodo AStar della classe RandomAStarProblem. Tali strutture dati sono un dizionario python per i nodi incontrati durante la ricerca(explored) e un oggetto AStarFrontier la cui dichiarazione della relativa classe si trova nel modulo Frontier.py

2.2 Specifiche Della Piattaforma di Test

La piattaforma di test è un HP Notebook con Sistema Operativo Windows 10 Home, 12 GB di RAM e con il seguente processore:
Intel Core i5-7200U CPU @ 2.50 GHz 2.70 GHz

3 Documentazione Codice

3.1 Interazione fra Moduli

Nel progetto sono disponibili 2 eseguibili. uno si trova nel modulo DBLoader.py che crea i Pattern Database, li serializza e li scrive su dei file di testo('DB1.txt','DB2.txt','DB3.txt','DB4.txt','DB5.txt','DB6.txt') attraverso la libreria pickle di python; l'altro si trova nel modulo Main.py ed effettua la ricerca su 500 istanze random del gioco del 15, generate come descritto in precedenza, attraverso le euristiche descritte in (Korf and Felner, 2002).

3.2 Scelte Implementative

Per quanto riguarda l'implementazione delle euristiche disjointDatabase e disjointAndReflected(implementate nella classe TilesProblem)ho scelto di partizionare le tessere del gioco del 15 seguendo una partizione (5,5,5) e non (7,8) che sarebbe la partizione ottimale come descritto in (Korf and Felner 2002); questo perchè nella creazione dei database con partizione (7,8) ho riscontrato problemi di memoria(RAM e file di paging pieno).

Rispetto alle misure effettuate sull'articolo(Korf and Felner 2002) ho scelto di non misurare il numero di nodi generati per trovare tutte le soluzioni ottime(il campo All solutions di Table1 di tale articolo) perchè richiedeva troppo tempo e memoria dato che utilizzo A* e non IDA* a differenza di come descritto in (Korf and Felner 2002).

4 Presentazione Risultati

Heuristic Function	Value	Nodes	Nodes/Sec	Seconds
Manhattan distance	19.99	56857	16503	3.45 s
Linear Conflicts	21.12	14727	3757	3.92 s
Disjoint Database	23.08	3427	10997	0.31 s
Disjoint + reflected	23.73	1762	7001	0.25 s

Tabella 1: Risultati sperimentali sul gioco del 15

Nella Tabella 1 possiamo vedere i risultati sperimentali delle diverse euristiche a confronto su 500 istanze random del gioco del 15.
Il campo Value indica il valor medio della relativa euristica sulle 500 istanze generate casualmente.

Il campo Nodes indica il numero medio di nodi generati per trovare il primo stato goal ottimo.
il campo Nodes/Sec indica la frequenza di nodi generati al secondo per avere un'idea della velocità delle varie euristiche.
Il campo Seconds indica il tempo impiegato dall'algoritmo per trovare il primo stato goal ottimo.

5 Analisi Risultati

Come ci si aspettava dalla teoria i database di pattern si rivelano euristiche molto efficaci malgrado il tempo necessario alla creazione dei database stessi, infatti confrontati alle altre euristiche in termini di tempo risultano molto più rapidi, anche in termini di nodi generati per trovare gli stati goal si rivelano molto efficaci. I risultati invece un pò inattesi sono la lentezza dell'euristica Linear Conflicts rispetto all'euristica Manhattan malgrado la prima generi meno nodi rispetto alla seconda per trovare le soluzioni ottime, la seconda invece batte la prima in termini di tempo di esecuzione, tutto ciò probabilmente è dovuto all'implementazione della funzione euristica Linear conflicts che richiede più tempo rispetto alla funzione Manhattan distance(entrambe le funzioni si trovano nel modulo TilesProblem.py) e molto spesso è inconcludente rispetto alla prima.
Un'analoga analisi la possiamo fare sulle euristiche Disjoint Databases e Disjoint+Reflected.