A.D. 1308
unipg
DIPARTIMENTO
DI INGEGNERIA

Tesina Finale di

Algoritmi e strutture dati

Corso di Laurea in Ingegneria Informatica ed Elettronica – A.A. 2023-2024

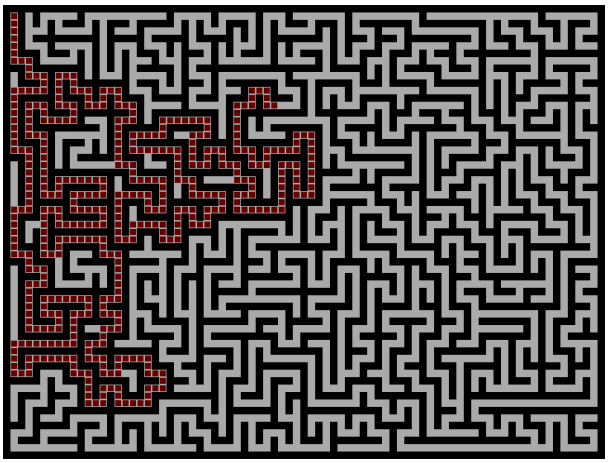
Dipartimento di Ingegneria

docente

Prof. Emilio Di Giacomo

Astar

algoritmo di ricerca



344955 **Aiman Rattab** aiman.rattab@studenti.unipg.it

Data ultimo aggiornamento: 14 luglio 2024

0. Indice

1. Introduzione

Questo progetto si pone l'obiettivo di illustrare il funzionamento dell'algoritmo *Astar (A*)* di Peter Hart, Nils Nilsson e Bertram Raphael, e di realizzare una sua implementazione attraverso il linguaggio di programmazione Java. L' algoritmo nasce con l'idea di risolvere un qualsiasi grafo pesato, in maniera completa (trova sempre la soluzione se esiste ed e' raggiungibile in un numero finito di passi), ottimale (trova la soluzione piu' efficiente, ovvero il percorso piu' breve per raggiungere il target) e con complessita' temporale minore di *Dijkstra*, l'algoritmo di ricerca senza euristica che trova moltissime applicazioni nel mondo informatico (usato ad esempio dai router per il pathfinding di un sito web nascosto dentro una rete di routers, o anche da Google Maps per trovare la strada piu' veloce da citta' A fino a B).

Di seguito sarà data una breve descrizione del funzionamento di *Dijkstra* e *Astar*.

2. L'algoritmo di Dijkstra

L'algoritmo di Dijkstra, proprio come A*, ha lo scopo di trovare il percorso piu' veloce in un grafo con archi pesati; per farlo in un modo efficiente, si applica una coda di priorita' per scegliere i nodi da esplorare, andando a prioritizzare i percorsi piu' brevi trovati finora (dunque l'ordinamento sara' dato dal pathcost fino a quel nodo). Esplorando prima i percorsi piu' brevi, e' garantita l'ottimalita' della soluzione (se ci fosse stato un percorso piu' breve, sarebbe stato esplorato prima della soluz. ottimale). Sappiamo inoltre che e' completo (ovvero trova sempre la soluzione, se esiste), poiche' l'algoritmo a fine esecuzione avra' esplorato tutti i nodi (l' algoritmo si ferma quando la frontiera e' vuota).

2.1 Pseudocodice Dijkstra

```
f = path_cost() //path_cost() sara' una funzione contenente il costo di tutto il percorso fino al nodo

frontiera = CodaPriorita('crescente', f) //frontie sara' una lista ordinata

                        //seguendo la funzione f definita qua sopra

frontiera.add(nodo_iniziale) //aggiungo il nodo di partenza del nostro problema

esplorati = Set() //esplorati andra' a contenere tutti i nodi che ho gia' espanso, e che quindi non

                        //dovro' piu' visitare, altrimenti il costo sarebbe per forza maggiore

While frontiera Not Empty:

    nodo = frontiera.pop() //prendo il primo elemento presente nella frontiera (e dunque il meno costoso)

    if nodo.stato == nodo_finale.stato: return nodo //se il nodo corrisponde allo stato finale (il

                        //nostro obiettivo), allora ho finito l'esecuzione

    esplorati.add(nodo.stato) //questo nodo verra' espanso, dunque lo aggiungo nella lista degli esplorati

    for child_node in nodo.adiacenti: //ciclo su tutti i nodi raggiungibili da quello corrente

        if child_node.stato Not in esplorati and child_node Not in frontiera://il nodo figlio non e'
```

```
    nodo = frontiera.pop() //prendo il primo elemento presente nella frontiera (e dunque il meno costoso)

    if nodo.stato == nodo_finale.stato: return nodo //se il nodo corrisponde allo stato finale (il

                        //nostro obiettivo), allora ho finito l'esecuzione

    esplorati.add(nodo.stato) //questo nodo verra' espanso, dunque lo aggiungo nella lista degli esplorati

    for child_node in nodo.adiacenti: //ciclo su tutti i nodi raggiungibili da quello corrente

        if child_node.stato Not in esplorati and child_node Not in frontiera://il nodo figlio non e'

                        //mai stato incontrato

            frontiera.add(child_node)

        else if child_node in frontiera://il nodo figlio e' raggiungibile anche da un altro nodo,

                        //controllo quale dei 2 percorsi e' il piu' breve

            if f(child_node) < frontiera[child_node]://aggiorno frontiera in caso il pathcost del nuovo
                        //nodo e' piu' breve

                delete frontiera[child_node]
```

Di seguito riporto lo pseudocodice dell'algoritmo con i vari commenti

3. Differenza tra Dijkstra e Astar

Astar, come funzionamento e' molto simile a Dijkstra, con una sola piccola modifica. In Astar viene introdotto il concetto di euristica; dato un problema (come ad esempio trovare la strada piu' breve che connette due citta'), si cerca di arrivare alla soluzione non solo dando priorita' (e quindi visitando prima) ai nodi con pathcost minore, ma a questo pathcost si somma anche un nuovo numero che indica la distanza "teorica" che quel nodo ha dalla soluzione. Se l'euristica risulta essere minore o uguale alla vera distanza dalla soluzione, per ogni nodo di partenza scelto, allora si tratta di un'euristica ammissibile (condizione sufficiente a garantire una soluzione ottimale). Un esempio di euristica e' la distanza in linea d'aria dal nodo da analizzare al goalstate.

3.1 Pseudocodice Astar

```
f = h() + path_cost //h() sara' una funzione contenente l'euristica del problema

                    //nel caso di pathfinding di una citta', h() restituisce distanza

                    //in linea d'aria dal nodo fino al nodo_finale. path_cost e' il costo

                    //per raggiungere quel determinato nodo

frontiera = CodaPriorita('crescente', f) //frontiera sara' una lista ordinata

                    //seguendo la funzione f definita qua sopra

frontiera.add(nodo_iniziale) //aggiungo il nodo di partenza del nostro problema

esplorati = Set() //esplorati andra' a contenere tutti i nodi che ho gia' espanso, e che quindi non

                    //dovro' piu' visitare, altrimenti il costo sarebbe per forza maggiore

While frontiera Not Empty:

    nodo = frontiera.pop() //prendo il primo elemento presente nella frontiera (e dunque il meno costoso)

    if nodo.stato == nodo_finale.stato: return nodo //se il nodo corrisponde allo stato finale (il

                    //nostro obiettivo), allora ho finito l'esecuzione
```

```

    nodo = frontiera.pop() //prendo il primo elemento presente nella frontiera (e dunque il meno costoso)

    if nodo.stato == nodo_finale.stato: return nodo //se il nodo corrisponde allo stato finale (il

                    //nostro obiettivo), allora ho finito l'esecuzione

    esplorati.add(nodo.stato) //questo nodo verra' espanso, dunque lo aggiungo nella lista degli esplorati

    for child_node in nodo.adiacenti: //ciclo su tutti i nodi raggiungibili da quello corrente

        if child_node.stato Not in esplorati and child_node Not in frontiera://il nodo figlio non e'

                    //mai stato incontrato

            frontiera.add(child_node)

        else if child_node in frontiera://il nodo figlio e' raggiungibile anche da un altro nodo,

                    //controllo quale dei 2 percorsi e' il piu' breve

            if f(child_node) < frontiera[child_node]://aggiorno frontiera in caso il pathcost del nuovo
                    //nodo e' piu' breve

                delete frontiera[child_node]

```

Operzione	Costo
TREE-SEARCH	$O(h)$
TREE-MINIMUM	$O(h)$
TREE-MAXIMUM	$O(h)$
TREE-SUCCESSOR	$O(h)$
TREE-PREDECESSOR	$O(h)$
TREE-INSERT	$O(h)$
TREE-DELETE	$O(h)$

Figura 3.1: In entrambi i casi, per l'implementazione della frontiera, ovvero la struttura dati che conterra' in ordine tutti i nodi da esplorare, ho scelto di utilizzare un albero rosso-nero, in maniera da garantire profondita' (h) massima pari a log(n), dove n sono il numero di nodi; questo ci consentira' di avere operazioni di comparazione o ricerca piu' veloci:

4. OneGame e LabirinthGame

Per testare la complessita' temporale dei 2 algoritmi, ho deciso di implementare due problemi la quale struttura e' riconducibile ad un grafo pesato. Il primo e' LabirinthGame, che come suggerisce il nome, consiste nel trovare l'uscita (considerando che sia nota la posizione "spaziale" dell'uscita) nel minor tempo possibile. Per ricondurre il problema ad una struttura a nodi ed archi pesati, ho implementato una classe stato che potesse descrivere il problema, dove ogni stato indica un nodo, ed avra' la stessa matricie iniziale che rappresenta la struttura del labirinto, un vettore che indica la posizione attuale e il vettore goalstate. ogni stato e' collegato al massimo ad altri 4 (a seconda di presenza di muri o meno), e gli "archi" in questo caso indicano proprio lo spostamento da compiere per raggiungere il nuovo stato (nel nostro caso avranno sempre costo 1). Per testare la velocita' dei 2 algoritmi, ho implementato anche un maze generator, in maniera da avere un "gioco" procedurale, ed evitando cosi' di doverli scrivere a mano.

Il secondo gioco, OneGame, e' un semplice gioco che ho deciso di testare assieme a labirinthgame, poiche' nel caso di onegame l'euristica e' molto piu' vicina al caso reale di quanto lo sia l'euristica per la risoluzione di un labirinto (nel nostro caso la distanza in linea d'aria). Si tratta di una matrice che indica una stanza NxN con 4 direzioni in cui andare. L'obiettivo consiste nel cancellare tutti gli "1" nella matrice nel minor numero di passi possibile. L'euristica che ho implementato e' il conteggio degli "1" mancanti nella matrice (dovranno per forza essere cancellati, dunque almeno stesso numero di passi, euristica ammissibile). Vedremo poi dai dati sperimentali che anche solo un euristica cosi' banale aiuta molto l'algoritmo a trovare molto prima la soluzione ottimale.

5. Costo dei 2 algoritmi

5.1 Complessita' temporale Dijkstra

Supponendo di applicare Dijkstra ad un grafo pesato, avremo che ad ogni vertice del grafo possiamo prendere "b" strade diverse (supponiamo numero di archi per ogni nodo e' b). Data questa struttura ci e' piu' semplice trasformare il problema in una struttura ad albero, dove il padre di ogni nodo corrisponde al nodo dal quale siamo arrivati. Sapendo che Dijkstra prioritizza sempre i cammini piu' brevi possibili (la lunghezza del cammino e' data dalla profondita' in cui ci troviamo nell'albero di ricerca), allora avremo che gli unici cammini provati saranno quelli piu' piccoli della migliore soluzione di costo C*. Supponendo che in media ogni "passo" ha costo ϵ (dunque la sol ottimale e' raggiungibile in $\frac{C^*}{\epsilon}$ passi), allora i nodi nell' albero di profondita' minore saranno esattamente $b^{\frac{C^*}{\epsilon}} - 1$. Quindi la complessita' temporale sara' O($b^{\frac{C^*}{\epsilon}}$)

5.2 Complessita' temporale Astar

La complessita' temporale di Astar (in media) e' uguale a quella di Dijkstra, poiche' hanno appunto una struttura molto simile; ma dal momento che Astar possiede l'algoritmo di euristica che aiuta a direzionare la ricerca e saltare inutili ramificazioni, quest'ultimo sara' sempre piu' veloce (se di poco o di molto dipende dal tipo di euristica, e di conseguenza dal tipo di problema da risolvere) assumendo che si sia scelta un euristica ammissibile e "intelligente".

5.3 Complessita' spaziale

La complessita' spaziale, proprio come per quella tempolare, e' la stessa sia nel caso di A* che Dijkstra, poiche' in entrambi casi dobbiamo portarci dietro tutti i nodi esplorati (supponendo che la profondita' in cui si trova la soluzione sia d), e quindi la complessita' sara' O(b^d).

6. Dati sperimentali

Di seguito riporto i dati sperimentali dell'esecuzioni di Dijkstra, Astar e un algoritmo di bruteforce (implementati in Java) sui 2 problemi visti prima (labirinthgame e onegame):

6.1 Labirinti di dimensione ≤ 20

```
--------------------------------------------------------------------------------
Dimensione attuale labirinto: 12; numero di stati computati: 252, di cui 29 esiste una sol
Astar, media di esecuzione : 4.365079365079365E-5 secondi
Dijkstra, media di esecuzione : 2.777777777777778E-4 secondi
Bruteforce, media di esecuzione : 9.655172413793104E-4 secondi, con 0 problemi non risolti depth=64
--------------------------------------------------------------------------------



--------------------------------------------------------------------------------
Dimensione attuale labirinto: 14; numero di stati computati: 645, di cui 74 esiste una sol
Astar, media di esecuzione : 4.186046511627907E-5 secondi
Dijkstra, media di esecuzione : 2.2635658914728682E-4 secondi
Bruteforce, media di esecuzione : 0.0036216216216216216 secondi, con 0 problemi non risolti depth=64
--------------------------------------------------------------------------------



--------------------------------------------------------------------------------
Dimensione attuale labirinto: 20; numero di stati computati: 1648, di cui 145 esiste una sol
Astar, media di esecuzione : 0.0011589805825242719 secondi
Dijkstra, media di esecuzione : 0.0034520631067961167 secondi
Bruteforce, media di esecuzione : 0.026013793103448275 secondi, con 0 problemi non risolti depth=64
```

6.2 Labirinti di dimensione ≤ 40

```
--------------------------------------------------------------------------------
Dimensione attuale labirinto: 31; numero di stati computati: 1506, di cui 39 esiste una sol
Astar, media di esecuzione : 0.03295683930942895 secondi
Dijkstra, media di esecuzione : 0.04639176626826029 secondi
Bruteforce, media di esecuzione : 0.9933076923076923 secondi, con 0 problemi non risolti depth=64
--------------------------------------------------------------------------------



--------------------------------------------------------------------------------
Dimensione attuale labirinto: 37; numero di stati computati: 2778, di cui 52 esiste una sol
Astar, media di esecuzione : 0.0507080633549316 secondi
Dijkstra, media di esecuzione : 0.0676331893448524 secondi
Bruteforce, media di esecuzione : 1.0909423076923077 secondi, con 0 problemi non risolti depth=64
--------------------------------------------------------------------------------



--------------------------------------------------------------------------------
Dimensione attuale labirinto: 41; numero di stati computati: 4617, di cui 72 esiste una sol
Astar, media di esecuzione : 0.08835131037470219 secondi
Dijkstra, media di esecuzione : 0.11043079922027291 secondi
Bruteforce, media di esecuzione : 2.628861111111111 secondi, con 0 problemi non risolti depth=64
```

6.3 Onegame, dimensione ≤ 5

```
--------------------------------------------------------------------------------
Dimensione attuale labirinto: 3; numero di stati computati: 29, di cui 29 esiste una sol
Astar, media di esecuzione : 5.862068965517242E-4 secondi
Dijkstra, media di esecuzione : 8.275862068965517E-4 secondi
Bruteforce, media di esecuzione : 0.0 secondi, con 0 problemi non risolti depth=64
--------------------------------------------------------------------------------




--------------------------------------------------------------------------------
Dimensione attuale labirinto: 5; numero di stati computati: 47, di cui 46 esiste una sol
Astar, media di esecuzione : 0.03331914893617021 secondi
Dijkstra, media di esecuzione : 0.22993617021276597 secondi
Bruteforce, media di esecuzione : 0.0 secondi, con 0 problemi non risolti depth=64
--------------------------------------------------------------------------------




--------------------------------------------------------------------------------
Dimensione attuale labirinto: 5; numero di stati computati: 52, di cui 51 esiste una sol
Astar, media di esecuzione : 0.04121153846153846 secondi
Dijkstra, media di esecuzione : 0.4301346153846154 secondi
Bruteforce, media di esecuzione : 0.0 secondi, con 0 problemi non risolti depth=64
```

Per il gioco Onegame ho omesso la soluzione bruteforce poiche' impiegherebbe troppo tempo (nell'ordine di minuti o anche ore).

6.4 Onegame, dimensione 6×6

```
--------------------------------------------------------------------------------
Dimensione attuale labirinto: 6; numero di stati computati: 8, di cui 8 esiste una sol
Astar, media di esecuzione : 28.80675 secondi
Dijkstra, media di esecuzione : 0.0 secondi
Bruteforce, media di esecuzione : 0.0 secondi, con 0 problemi non risolti depth=64
--------------------------------------------------------------------------------
```

Dai dati sperimentali e' immediato notare come Astar sia l'algoritmo piu' veloce in ogni esecuzione. Inoltre va notato che nel caso di Onegame, la differenza di tempo tra A* e Dijkstra e' molto maggiore rispetto a Labirinthgame, e questo e' dovuto all'euristica del problema (l'euristica di onegame permette di saltare tutte

quelle esplorazioni che non portano ad una cancellazione, prioritizzando appunto la cancellazione di "1").