Documentazione Fia-Man

Basile Annamaria Siepe Fabio

19 febbraio 2022

LAUREA TRIENNALE IN INFORMATICA UNIVERSITÀ DI SALERNO CORSO DI FONDAMENTI DI INTELLIGENZA ARTIFICIALE PROFESSORE FABIO PALOMBA

Repository Github: https://github.com/FabioSiepe/FIA-PAC-MAN

Basile Annamaria, Siepe Fabio

2022

Indice

CAPITOLO 1

Definizione del problema

1.1 Introduzione

Il progetto FIA-MAN nasce dalla curiosità di voler osservare come algoritmi studiati a lezione funzionassero quando applicati a videogiochi. Per semplicità sono stati selezionati giochi con un limitato numero di input possibili dall'utente. Tra i vari videogiochi presi in considerazione, dopo un ricevimento con il professore F. Palomba, è stato scelto PAC-MAN. Per la realizzazione di questo problema è stato utilizzato un framework trovato su github chiamato jpacman-framework.

1.2 Obiettivi

Lo scopo di questo progetto è stato quello di creare un IA capace di "giocare" a PAC-MAN. Questo gioco consiste in pac-man che è una sorta di pallina gialla, capace di muoversi in un matrice quadrata e che ha lo scopo di mangiare tutte palline che sono posizionate all'interno della matrice stessa.

1.3 Specifica PEAS

- **Performance:** sono le misure di prestazione adottate per valutare l'operato di un agente, in questo caso valutiamo se mangia tutte le palline o meno.
- **Environment:** Descrizione degli elementi che formano l'ambiente (descritto sotto).
- **Actuators:** Gli attuatori disponibili dell'agente per intraprendere le azioni. In questo caso i nostri attuatori saranno le quattro dierzioni: nord, sud, est, ovest.
- **Sensors:** I sensori attraverso i quali riceve gli input percettivi.

1.3.1 Caratteristiche dell'ambiente

- **Singolo agente:** sul tavolo da gioco esiste un solo agente in cui lo scopo è quello di mangiare tutte le palline .
- **Parzialmente osservabile:** l'agente non conosce la posizione delle palline a priori, ma può sapere se un determinato quadrato contiene una pallina o meno.
- **Stocastico:** l'agente conosce lo stato successivo dell'ambiente perché è determinato dallo stato corrente dell'azione eseguita.

- **Statico:** durante l'esecuzione dell'agente l'ambiente non muta mentre pensa alla prossima azione.
- **Episodico:** Ogni mossa che compie un'azione la prossima dipenderà direttamente dalla mossa stessa.

1.4 Analisi del problema

Il problema può essere formalizzato descrivendolo:

- **Stato iniziale:** lo stato iniziale è definito tramite un file.txt chiamato board.txt. Ad ogni singolo valore contenuto in questo file corrisponde un valore della board:
  - **# :** corrisponde un muro
  - **. :** corrisponde il cibo
  - **G :** corrisponde un fantasma (ghost)
  - **P :** corrisponde un player
- **Descrizione delle azioni possibili:**Pac-man ha la possibilità di effettuare 4 possibili azioni: muoversi in nord, sud, ovest ed est: move (player, Direction).
- **Modello di transizione:** Ad ogni azione si andrà a controllare se nella lista occupans del singolo Square è contenuto un pallino.Sia nel caso in cui non contiene il pallino o che lo contiene, il player si sposterà su quel determinato Square, se possibile(pac man non può camminare sui muri). Nel caso in cui continene il pallino lo si andrà ad eliminare aumentando lo score di pacman.
- **Test obbiettivo:** l'obiettivo di pac-man è quello di mangiare tutte le palline presenti. Il gioco termina quando le mangia tutte.
- **Costo del cammino:** ogni azione che esegue pac-man ha lo stesso costo.

Questo problema è stato affrontato guardandolo da diverse prospettive. Nel particolare sono stati utilizzati algoritmi di ricerca non informata(ricerca in ampiezza, ricerca in profondità), e algoritmi di ricerca informata (A*). Come si può intuire leggendo la definizione del problema si è deciso di togliere i nemici (fantasmi) per avere un ambiente più semplice da attraversare. Per cercare di rimanere fedeli al gioco pac-man si è inoltre utilizzato un algoritmo di ricerca con avversari in particolare il min/max per quest'ultima strategia è stato modificato l'ambiente, ma ne parleremo più avanti.

CAPITOLO 2

Algoritmi di ricerca

2.1 Ricerca non informata

Gli algoritmi scelti per la prima implementazione sono: Ricerca in Ampiezza e Ricerca in Profondità. Entrambi fanno parte degli algoritmi di ricerca non informata. Per algoritmi di ricerca non informata ci riferiamo al fatto che le strategie non dispongono di infromazioni aggiuntive sugli stati.

2.1.1 Ricerca in ampiezza

La ricerca in ampiezza è una strategia sistematica di ricerca, ovvero è in grado di trovare sempre una soluzione, se esiste, nel quale si estende prima il nodo radice e poi i loro successori e cosi via. In particolare nel nostro caso i nodi vengono espansi attraverso un determinato ordine che sarebbe nord, sud, ovest ed est. Dato quest'ordine possiamo dire che il primo nodo espanso ad una certa profodità d sarà sempre il nodo con direzione nord e l'ultimo nodo espanso sarà quello con direzione est. Detto ciò, dato che la ricerca in ampiezza ritorna sempre la prima soluzione trovata, possiamo aspettarci quale sarà il cammino per quest'algoritmo.

La ricerca in ampiezza espanderà tutti i nodi che precedono il nodo obbiettivo fino a raggiungerlo. Come conseguenza del fatto che espande tutti i nodi precedenti al nodo obbiettivo conosciamo che la sua complessità temporale è $O(b^d)$ che nel nostro problema si traduce con un fattore di ramificazione b di 4 (le possibili azioni che pac-man può effettuare) e d, che indica la profondità del nodo obbiettivo più vicino allo stato iniziale. Mentre dal punto di vista della memoria occupata, dato che ogni nodo generato rimane in memoria, ci saranno $O(b^{d-1})$ nodi nell'insieme esplorato e $O(b^d)$ nodi nella frontiera, perciò la complessità complessiva è $O(b^d)$.

```java
public final class Ampiezza {

    Ampiezza() {
    }

    public static List<Direction> nonInformata(Square attuale, Unit giocatore) {
        List<Nodo> frontiera = new ArrayList<>();
        frontiera.add(new Nodo(null, attuale, null));
        Set<Square> esplorati = new HashSet<>();

        while (!frontiera.isEmpty()) {

            Nodo node = frontiera.remove(0);
            Square padre = node.getSquare();
            List<Unit> occupants = padre.getOccupants();

            for (Unit u : occupants) {
                if (u instanceof Pellet) {
                    return node.getPath();
                }
            }
            esplorati.add(padre);

            for (Direction dir : Direction.values()) {
                Nodo figlio = new Nodo(dir, padre.getSquareAt(dir), node);

                if (!esplorati.contains(figlio.getSquare()) && (!frontiera.contains(figlio))
                        && (figlio.getSquare().isAccessibleTo(giocatore)))
                    frontiera.add(figlio);
            }

        }
        return null;
    }
```

In questo algoritmo ciò che andiamo a fare è inizializzare la frontiera al nodo attuale, ovvero quello in cui siamo, e attraverso un ciclo controlliamo tutti i nodi vicini (esplorando l'albero di ricerca) fino a quando non troviamo un nodo obiettivo.

2.1.2 Ricerca in profondità

Nella ricerca in profondità si espande sempre per primo il nodo più profondo nella frontiera fino a raggiungere le foglie dell'albero di ricerca. Anche in questo caso i nodi vengono espansi attraverso un determinato ordine che sarebbe nord, sud, ovest ed est. Dato quest'ordine possiamo dire che inizia con l'esplandere l'ultima direzione disponibile fino a quando non trova una soluzione. Infatti abbiamo notato che molto spesso torna sui suoi passi per mangiare una pallina distante rispetto a dove si trova in quel momento.

La ricerca in profondità, essendo stata realizzata su un grafo finito, che evita stati ripetuti e cammini ridondanti, è completa, se la soluzione esiste, perché alla fine espanderà tutti i nodi, ma non sarà mai ottima perché sul suo cammino può trovare un cammino sub-ottimo. Per quanto riguarda la complessità temporale è limitata dalla dimensione dello spazio degli stati e genera tutti gli $O(b^m)$ nodi, dove m è la profondità massima di un nodo, bisogna precisare che m può essere molto più grande di d. Mentre dal punto di vista della memoria occupata, essendo l'implementazione uguale a quella della ricerca in ampiezza sarà $O(b^m)$.

```
public final class Profondita {

    Profondita() {
    }

    public static List<Direction> nonInformata(Square attuale, Unit giocatore) {
        LinkedList<Nodo> frontiera = new LinkedList<>();
        frontiera.add(new Nodo(null, attuale, null));
        Set<Square> esplorati = new HashSet<>();

        while (!frontiera.isEmpty()) {

            Nodo node = frontiera.remove(0);
            Square padre = node.getSquare();
            List<Unit> occupants = padre.getOccupants();

            for (Unit u : occupants) {
                if (u instanceof Pellet) {
                    return node.getPath();
                }
            }
            esplorati.add(padre);

            for (Direction dir : Direction.values()) {
                Nodo figlio = new Nodo(dir, padre.getSquareAt(dir), node);

                if (!esplorati.contains(figlio.getSquare()) && (!frontiera.contains(figlio))
                        && (figlio.getSquare().isAccessibleTo(giocatore)))
                    frontiera.addFirst(figlio);
            }

        }
        return null;
    }
```

Questo algoritmo risulta molto simile all'algoritmo di ricerca in ampiezza tranne per la gestione della frontiera. Qui utilizziamo una LinkedList alla quale andremo ad aggiungere il nodo all'inizio della frontiera, a differenza della ricerca in ampiezza dove la frontiera era gestita da una semplice lista.

2.2 Ricerca informata

Una strategia di ricerca informata può trovare soluzioni in modo più efficiente di una strategia non informata perché oltre a conoscere gli stati del problema conosce informazioni aggiuntive sul problema dette funzioni euristiche. La funzione euristica più diffusa, utilizzata anche in questo progetto, è $h(n)$ ovvero il costo stimato del cammino più conveniente dallo stato del nodo n a uno stato obbiettivo, che nel nostro problema si traduce nella distanza tra la posizione di pac-man e la posizione delle palline che deve mangiare. Avendo modellato il problema su una griglia è stato deciso di utilizzare come metodo per calcolare la funzione $h(n)$ la distanza tra due punti. Le coordinate ci sono state fornite dalla classe board.java ed è per questo che alla funzione viene passata la board a differenza degli altri due gli altri due algoritmi presentati in precedenza(ampiezza e profondità).

2.2.1 Ricerca A*

Come terzo algoritmo è stata analizzata la ricerca A* che appartiene agli algoritmi di ricerca informata. L'implementazione della ricerca A* si basa sull'algoritmo di ricerca a costo uniforme a cui però viene modificato il test di valutazione dei nodi che combina $g(n)$, il costo per raggiungere il nodo, e $h(n)$, il costo per andare da li all'obbiettivo, formando $f(n)$, ovvero, il costo stimato della soluzione più conveniente che passa per n.

```
public class Astar {

    Astar() {
    }

    public static List<Direction> informata(Square attuale, Unit giocatore, Board board) {
        PriorityQueue<Nodo> frontiera = new PriorityQueue<>();
        frontiera.add(new Nodo(null, attuale, null, 0, board));
        Set<Square> esplorati = new HashSet<>();

        while (!frontiera.isEmpty()) {

            Nodo node = frontiera.poll();
            Square padre = node.getSquare();
            List<Unit> occupants = padre.getOccupants();

            for (Unit u : occupants) {
                if (u instanceof Pellet) {
                    return node.getPath();
                }

            }
            esplorati.add(padre);

            for (Direction dir : Direction.values()) {
                Nodo figlio = new Nodo(dir, padre.getSquareAt(dir), node,
                        costo(attuale, padre.getSquareAt(dir), node.getCosto() + 1, board), board);

                if (!esplorati.contains(figlio.getSquare()) && (!frontiera.contains(figlio))
                        && (figlio.getSquare().isAccessibleTo(giocatore)))
                    frontiera.add(figlio);
                else {
                    for (Nodo f : frontiera) {
                        if ((f.uguali(figlio.getSquare())) && (f.compareTo(figlio) > 0)) {
                            frontiera.remove(f);
                            frontiera.add(figlio);
                        }
                    }
                }
            }
        }
        return null;

    }
```

La prima parte dell'implementazione è praticamente analoga agli algoritmi visti in precedenza tranne per la gestione della frontiera che viene fatta attraverso una coda a priorità. Per la seconda parte (quella che cerca i prossimi nodi da aggiungere alla frontiera) nel momento in cui ci troviamo a considerare un nodo figlio già esplorato o contenuto in frontiera, invece di scartarlo, confrontiamo il costo del nodo figlio preso in considerazione con quello già incontrato. Se il costo del nodo figlio è minore di quello incontrato allora andremo a rimuovere quello già incontrato e inseriamo quello con costo minore all'interno della frontiera.

L'algoritmo A* per sua natura è completo, ottimo e ottimamente efficiente. Completo perché esiste un numero finito di nodi di costo minore o uguale a C* (il costo di cammino della soluzione ottima). Ottimo perché h(n) è sia ammissibile che consistente, l'ammissibilità deriva dal fatto che l'euristica non sbaglia mai per eccesso la stima del costo per arrivare all'obbiettivo, mentre la consistenza esiste perché per ogni nodo n e ogni successore n', il costo stimato per raggiungere l'obbiettivo partendo da n non è superiore al costo di passo per arrivare a n' sommato al costo stimato per andare da lì all'obbiettivo.

2.3 Ricerca con avversari

Per le soluzioni affrontate fino ad adesso, gli algoritmi visti sopra sono stati implementati senza avversari(fantasmi), abbiamo cercato di includerli un algoritmo di ricerca con avversari in particolare il min/max. Per affrontare questo problema abbiamo dovuto rivedere la formulazione del problema e abbiamo tenuto conto di un ambiente multi-agente. La formulazione in questo caso è:

- **Stato iniziale :** lo stato iniziale è definito tramite un file.txt chiamato board.txt. Ad ogni singolo valore contenuto in questo file corrisponde un valore della board:
  - **# :** corrisponde un muro
  - **. :** corrisponde il cibo
  - **G :** corrisponde un fantasma (ghost)
  - **P :** corrisponde un player
- **Giocatore:** abbiamo due tipi di giocatori:
  - Player(pac-man)
  - Fantasma(Ghost)
- **Descrizioni delle azioni possibili :**
  - Pac-man ha la possibilità di effettuare 4 possibili azioni: muoversi verso nord, sud, ovest ed est: move (player, Direction)
  - Il fantasma ha la possibilità di effettuare 4 possibili azioni: muoversi verso nord, sud, ovest ed est: move (ghosta, Direction)
- **Risultato :** Ad ogni mossa pac-man cerca di mangiare le palline e scappare dal fantasma. Il fanstasma inseguirà pac-man per mangiarlo.
- **Test di terminazione :** ritorna vero nel momento in cui siamo su un quadrato obiettivo.
  - **Player o pac-man :** il quadrato contiene una pallina
  - **Fantasma :** il quadrato contiene pac-man.

  In tutti gli altri casi ritorna false. Per ovviare al problema temporale della soluzione min/max, la terminazione è dettata anche da un certo valore chiamato profondità che terminerà la ricerca nel caso in cui è zero.

- **Funzione utilità :** restituisce la distanza (intesa come distanza fra due punti + numero di passi) tra i due giocatori.

2.3.1 Ricerca MIN/MAX

Per lo sviluppo di questo algoritmo abbiamo implementato diverse strategie. In particolare, abbiamo iniziato implementando una strategia di min max per poi spostarci ad una implementazione con potatura. Per motivi di tempo l'implementazione con potatura non è terminata.

```
public class MinMax {

    MinMax() {

    }

    public static Direction mossaMigliore(Boolean isPlayer, Square attuale, Board board, Unit giocatore,
            Unit nemico) {
        return minMax(isPlayer, attuale,nemico.getSquare(), board, giocatore, nemico, 8).getDirections();
    }

    public static ScoredMove minMax(Boolean isPlayer, Square attuale, Square attualeN, Board board, Unit giocatore,
            Unit nemico, int profondità) {

        float bestScore = isPlayer ? Integer.MIN_VALUE : Integer.MAX_VALUE;
        Direction bestMove = null;

        if (terminazione(isPlayer, attuale) || (profondità == 0)) {
                bestScore = utility(attuale, attualeN, board, giocatore, profondità);
        } else {
            for (Direction dir : Direction.values()) {
                if (isPlayer) {
                    if (attuale.getSquareAt(dir).isAccessibleTo(giocatore)) {
                        ScoredMove scoredMove = minMax(false, attualeN, attuale.getSquareAt(dir), board, nemico,
                                giocatore, profondità - 1);
                        if (scoredMove.getDistanza() > bestScore) {
                            bestScore = scoredMove.getDistanza();
                            bestMove = dir;
                        }
                    }
                } else {
                    if (attuale.getSquareAt(dir).isAccessibleTo(giocatore)
                            && (!giocatore.getSquare().equals(attuale.getSquareAt(dir)))) {
                        ScoredMove scoredMove = minMax(true, attualeN, attuale.getSquareAt(dir), board, nemico,
                                giocatore, profondità - 1);
                        if (scoredMove.getDistanza() < bestScore) {
                            bestScore = scoredMove.getDistanza();
                            bestMove = dir;
                        }
                    }
                }
            }
        }
        return new ScoredMove(bestMove, bestScore);
    }
```

In questa implementazione funziona in maniera ricorsiva. Vediamo all'inizio del primo if che esegue una funzione di terminazione che come scritto sopra ritorna true nel momento in cui ci si trova in uno stato obiettivo, altrimenti false. Un altro modo per terminare è che la profondità sia zero(valore che viene decrementato ad ogni chiamata ricordiva), questo tipo di soluzione è stata adottata per ovviare ai problemi di complessità temporale della min/max e per gestire alberi di profondità troppo grandi. Quindi in alcuni casi ci accontentiamo anche

di soluzioni sub-ottimali. Nel momento in cui ci troviamo nello stato obiettivo andremo a calcolare l'utilità e la ricorsione si blocca. quando non ci troviamo in uno stato obiettivo, invece, andiamo ad eseguire la ricorsione per tutte e e 4 le possibili direzioni. Una volta controllate le direzioni andremo a ritornare la miglior mossa da poter fare in questo stato.

2.4 Conclusioni

In conclusione possiamo dire che quetsa esperienza di progetto è stata molto divertente perchè ci ha permesso di lavorare su uno dei videogiochi più iconici mai creati e in più oltre al divertimento ci ha permesso di capire meglio il funzionamento degli algoritmi visti a lezione e implementati nel progetto.

CAPITOLO 3

Glossario

Termini, abbreviazioni e sigle

- **Specifica P.E.A.S.** : Performance Environment Actuators Sensors, sintesi in una parola degli aspetti da prendere in considerazione nello studio dell'intelligenza Artificiale.

Bibliografia

[1] Wikipedia;. Available from: https://it.wikipedia.org/wiki/Pagina_principale.

[2] FIA-PAC-MAN GitHub;. Available from: https://github.com/FabioSiepe/FIA-PAC-MAN.

[3] PAC-MAN GitHub;. Available from: https://github.com/SERG-Delft/jpacman-framework.