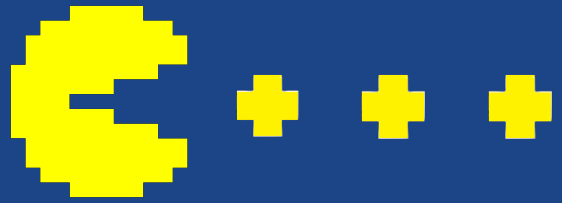


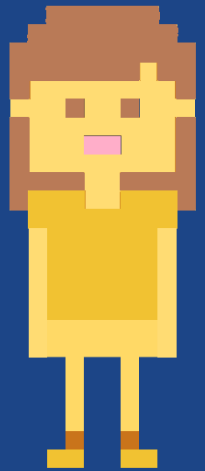
FIA-MAN

Progetto F1A 2021/2022

CHI SIAMO

FABIO SIEPE
MAT [05121 09355]

ANNAMARIA BASILE
MAT [05121 08137]



Introduzione

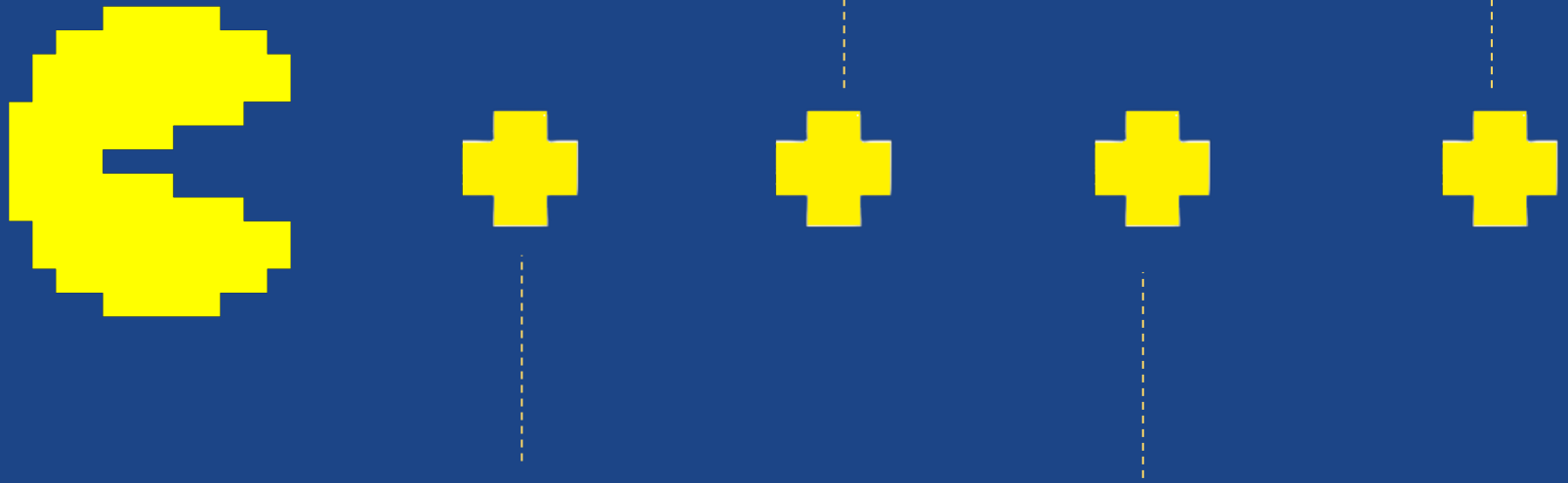
Quale gioco usare?

- Pochi input
- Conosciuto da entrambi
- Divertente

OBIETTIVI

L'obiettivo del nostro progetto è quello di creare un IA capace di giocare a pac-man.

SPECIFICA P.E.A.S



Specifiche Dell'Ambiente

Singolo Agente

Parzialmente Osservabile

? ? ?

Deterministico

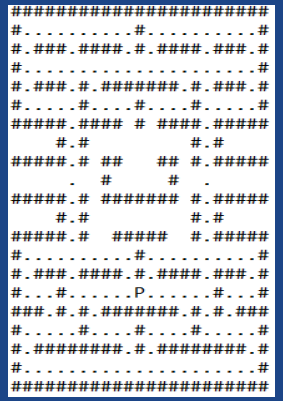
Statico

Episodico

ANALISI DEL PROBLEMA

Stato iniziale

Viene definito un file.txt chiamato board.txt



Descrizione delle azioni possibili

Pac-Man ha la possibilità di effettuare 4 azioni

Test Obiettivo

L'obiettivo di pac-man è quello di mangiare tutte le palline.

Modello di transizione

Ad ogni azione si controlla se nella lista occupans del singolo Square è contenuto un pallino

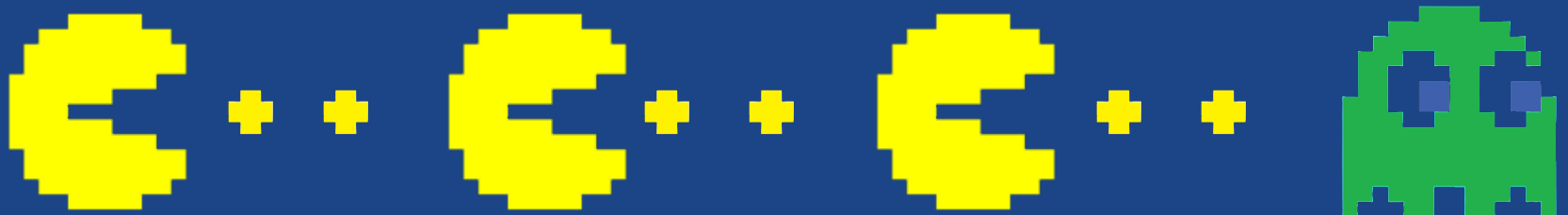
Costo del cammino

Le azioni eseguite da pac-man hanno lo stesso costo.

Come Abbiamo affrontato il problema

Ricerca in Ampiezza

Ricerca A*



Ricerca in Profondità

Min/Max

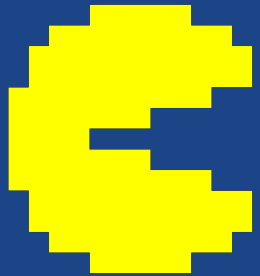
01

Ricerca In Ampiezza

Ricerca In Ampiezza

La ricerca in ampiezza è una strategia di ricerca sistematica, dove si estende prima il nodo radice e poi i suoi successori.

- Complessità temporale: $O(b^d)$
- Complessità spaziale: $O(b^d)$



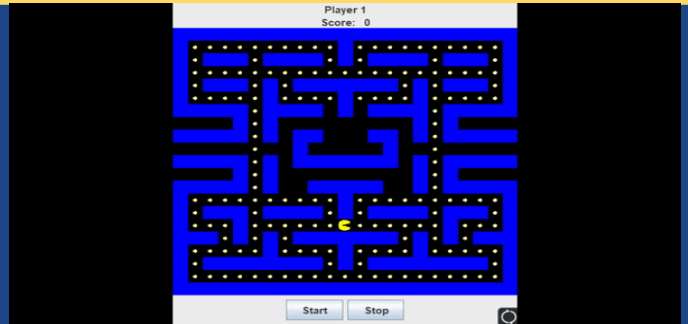
Esplorazione Albero Ampiezza

NORD

SUD

OVEST

EST



Ricerca In Ampiezza

```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;
    }
```

For per controllare se nel nodo preso in esame è contenuta una pallina

Direction.values() ritorna le direzioni possibili nell'ordine Nord, Sud, Ovest, Est

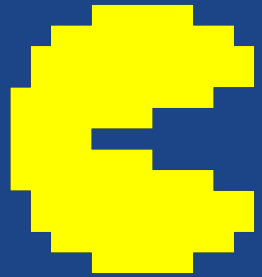
02

Ricerca In Profonfità

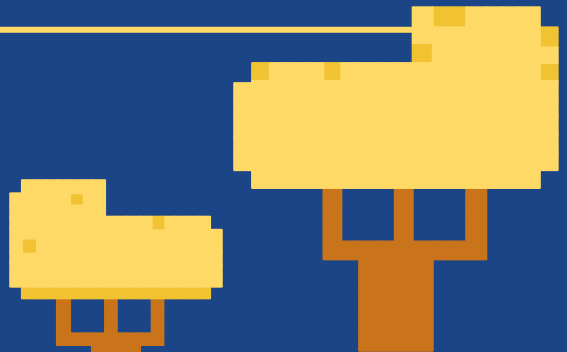
Ricerca In Profondità

Nella ricerca in profondità si espande sempre per primo il nodo più profondo nella frontiera fino a raggiungere le foglie dell'albero di ricerca

- Complessità temporale: $O(b^m)$
- Complessità spaziale: $O(b^m)$



Esplorazione Albero Profondità



Ricerca In Profondità

```java
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;
    }
```

LinkedList<Nodo>

Frontiera.addFirst(Figlio)

03

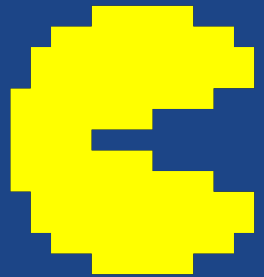
Ricerca A*

Ricerca A*

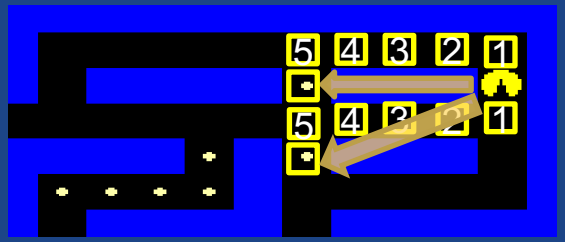
La ricerca A* si basa sull'algoritmo di ricerca a costo uniforme, a cui viene pero modificato il test di valutazione dei nodi che combina:

$$g(n) + h(n) = f(n)$$

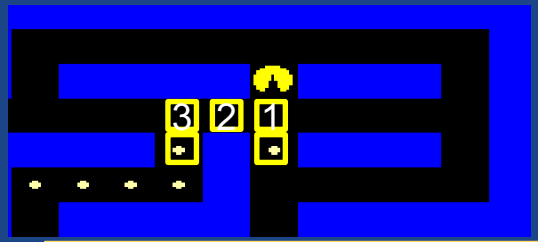
- Complessità temporale: $O(b^{\varepsilon})$
- Complessità spaziale: $O(b^{m})$



Ricerca In A*



Sceglie di andare alla pallina sopra perché
la distanza a livello di quadrati è la stessa
ma a livello di distanza tra due punti cambia



Questa scelta è più semplice ...

Ricerca A*

```java
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;

    }
```

Costo() calcola il costo per arrivare a quel determinato quadrato

Controllo della frontiera nel caso in cui incontriamo un nodo già esplorato

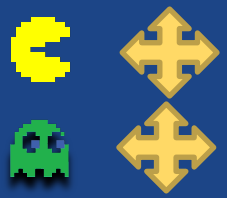
04

Ricerca con avversari

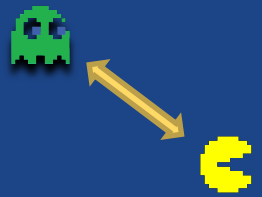
Formulazione

Multi - Agente

Azioni Possibili



Risultato



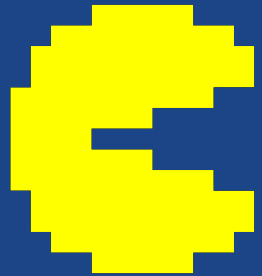
Test Terminazione

Funzione utilità

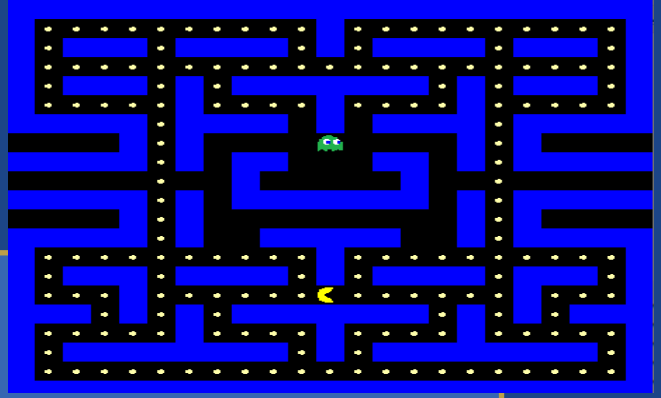
Ricerca MINMAX

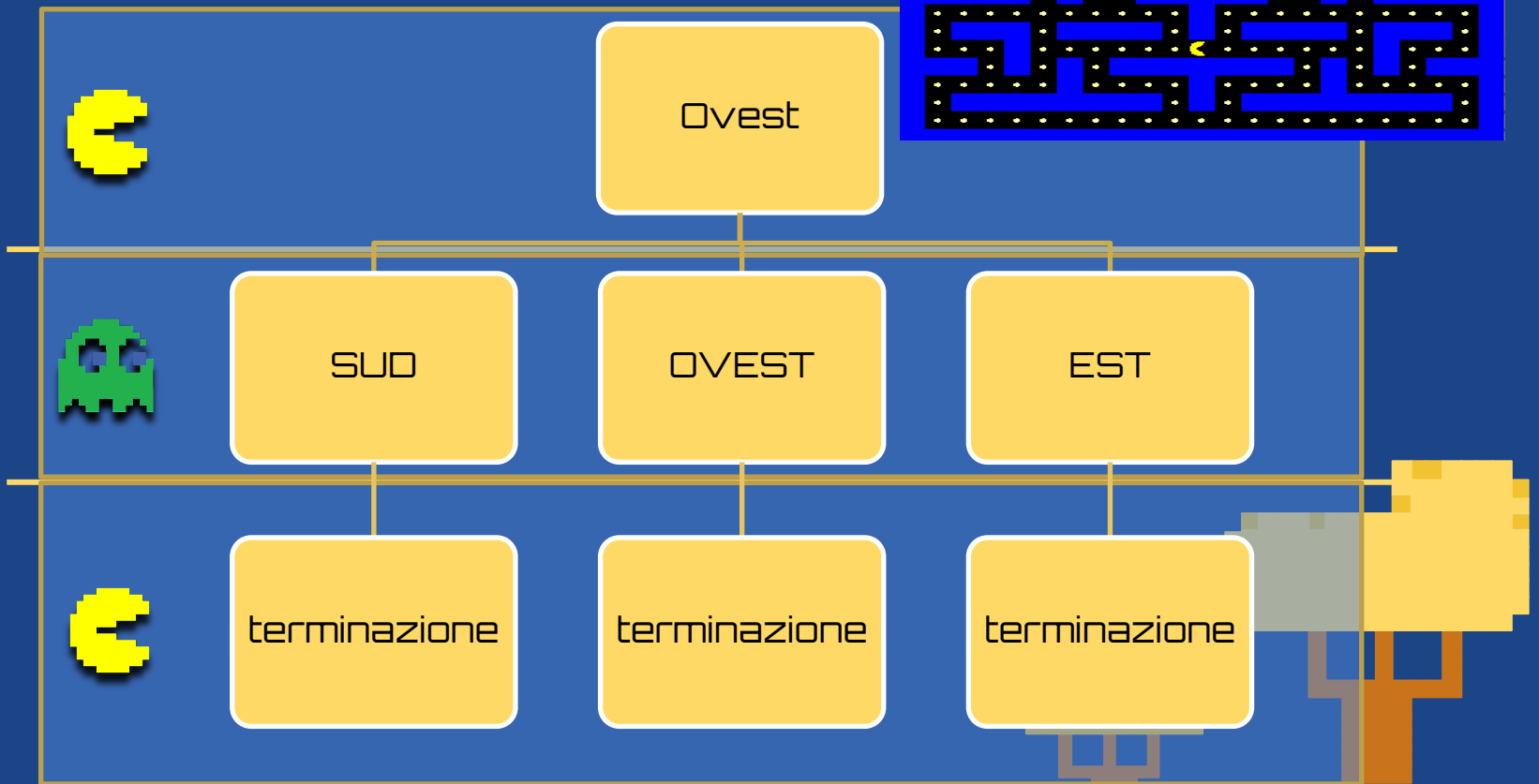
L'**algoritmo minimax** è un **algoritmo** ricorsivo per la ricerca della mossa migliore in un gioco a somma zero che si svolge tra due giocatori:

- Completezza: è completa se l'albero è finito
- Ottimalità: è ottimo nel caso in cui l'avversario è ottimo
- Complessità temporale: $O(b^m)$
- Complessità spaziale: O(bm)

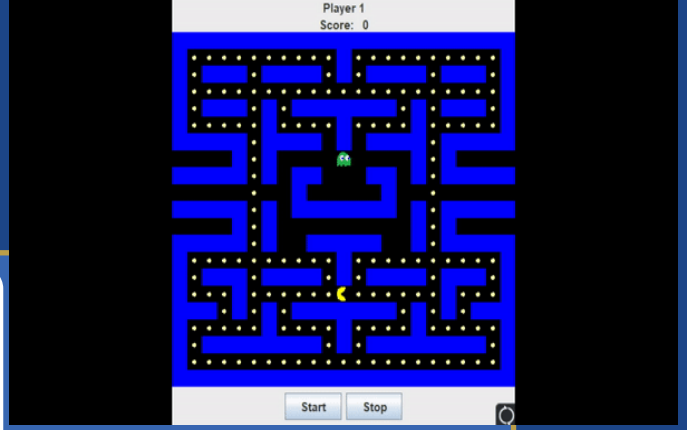


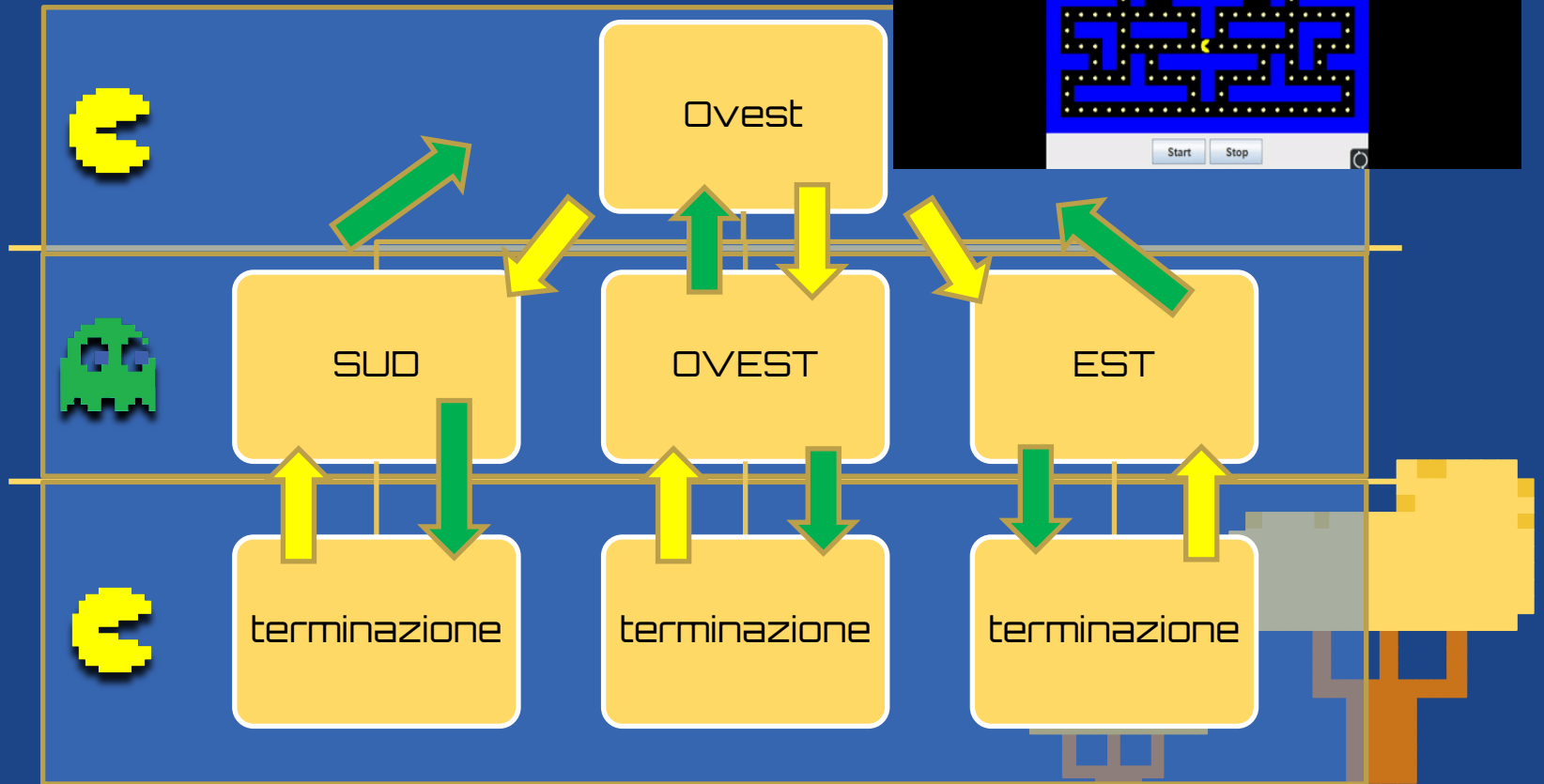
Esplorazione Albero Min Max





Esplorazione Albero Min Max





MinMax

```java
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);
    }
```

BestScore e bestMove

Test Di terminazione e funzione utilità

Chiamata Ricorsiva

Valore di ritorno

Conclusioni

In conclusione, possiamo dire che questa esperienza di progetto è stata molto divertente e costruttiva poiché ci ha permesso di capire meglio il funzionamento degli algoritmi studiati.

Grazie per l'attenzione!