Relazione PokéLabirinto

Aubeeluck Aditya

ITI Pino Hensemberger - a.s. 25/26

Abstract

Il progetto PokéLabirinto unisce la generazione casuale di un labirinto con le dinamiche di un gioco a tema Pokémon.
L'obiettivo è guidare un Pokémon scelto dall'utente attraverso un percorso labirintico fino a raggiungere l'uscita, affrontando nel frattempo un Pokémon avversario con cui può combattere.

Per la realizzazione sono stati applicati concetti fondamentali di informatica e programmazione.

Il risultato è un'applicazione che dimostra come concetti teorici possano essere tradotte in un contesto interattivo.

Indice

1) Introduzione

Il progetto PokéLabirinto nasce con l'obiettivo di unire la generazione casuale di un labirinto con la logica di un gioco a tema Pokémon.

Lo scopo del gioco è quello di creare un ambiente in cui un Pokémon scelto dall'utente deve muoversi all'interno di un percorso labirintico fino a raggiungere l'uscita.

Per realizzare questo obiettivo sono stati utilizzati alcuni concetti fondamentali di informatica e programmazione:

- Algoritmo ricorsivo con backtracking, che permette di generare il labirinto in modo sempre diverso, garantendo la presenza di un percorso principale e di alcune biforcazioni casuali.
- Algoritmo di mescolamento (Fisher-Yates), che consente di variare l'ordine delle direzioni durante la generazione del labirinto, evitando schemi ripetitivi.

- Distanza di Manhattan, usata per assicurare che l'uscita sia posizionata a una distanza minima dall'entrata, così da rendere il gioco più piacevole.

- Programmazione a oggetti, che ha permesso di organizzare il codice in classi distinte facilitando la gestione delle varie funzionalità andando a simulare delle entità.

Il risultato è un programma che mette in pratica tecniche di programmazione e algoritmi note, rendendo evidente come concetti teorici possano essere usati per simulazioni di giochi interattivi.

2) Basi teoriche

2.1)Backtracking

Il backtracking (BACKTRACKING, Aubeeluck Aditya, 2025) è una strategia algoritmica di esplorazione e ricerca usata per risolvere problemi che richiedono di valutare più combinazioni o sequenze di scelte. L'approccio si basa sul principio del "tentativo ed errore" dove si esplorano tutte le possibili soluzioni tornando indietro quando si scopre che una scelta conduce a un vicolo cieco.

Nella risoluzione di problemi che richiedono il superamento di vincoli si individuano 3 tipologie di algoritmi:

- Backjumping (BJ): in caso di fallimento nel trovare un valore valido per una variabile, identifica la variabile responsabile del conflitto e torna direttamente ad essa, saltando le variabili intermedie.

- Conflict-Directed Backjumping (CBJ): memorizza per ogni variabile un insieme di variabili con cui è entrata in conflitto durante la ricerca. Quando si verifica un fallimento, si utilizza questo insieme per stabilire a quale variabile saltare indietro.

- Forward Checking (FC): dopo ogni assegnazione, FC esamina le variabili non ancora assegnate e rimuove dai loro domini i valori che sono incompatibili con l'assegnazione corrente. Se il dominio di una di queste variabili si svuota, si effettua immediatamente il backtrack

2.2)Yates shuffle

Lo Yates shuffle (wikipedia, GeeksforGeeks) tratta di un algoritmo per permutare in modo casuale un array o una lista in tempo lineare O(n).

È stato descritto per la prima volta da R. A. Fisher e Frank Yates nel 1938 in un contesto statistico, successivamente reso popolare da Donald Knuth (1969, *The Art of Computer Programming*).

Funziona scegliendo progressivamente un elemento da scambiare con l'elemento corrente, garantendo che ogni permutazione sia equiprobabile. Il funzionamento:

1. Dato un vettore V di lunghezza *n*
2. Per ogni indice da *n*-1 a 1:
    a. dato un indice J gli si assegna un valore compreso fra 0 e 1
    b. si scambia V[i] con V[J]
3. Fatto ciò il vettore risulterà completamente mescolato

2.3)Distanza di Manhattan

La distanza di Manhattan (dataCamp2024, WolframMathWorld) misura le distanze tra due punti in uno spazio a griglia o tra punti in uno spazio multidimensionale sommando le differenze in valore assoluto delle coordinate dei punti.

Per P1(x1, y1) e P2(x2, y2)

$$d(P1, P2) = |x1 - x2| + |y1 - y2|$$

E' una metrica utilizzata in labirinti, IA e trova sempre applicazione quando si lavora con delle griglie.

3) Richiesta

Progettare un piccolo gioco in cui dato un Pokemon, questo viene posto in una posizione casuale di un Labirinto. Scopo del gioco è uscire dal labirinto.

Progettate un diagramma delle classi adeguato al problema. Dopodichè passate all'implementazione del gioco, se decidete di scriverlo in Java, riutilizzate pure le classi che avete sviluppato durante l'anno.

Prima variante: All'inizio del gioco l'utente sceglie un Pokemon con cui giocare. Ad ogni turno il programma dirà all'utente quali direzioni può prendere il Pokemon: avanti, indietro, sinistra e destra. Se non è possibile muoversi in una direzione (perchè bloccata da un muro), questa non verrà elencata.

Il labirinto deve essere visualizzato ad ogni mossa mentre viene percorso.

Seconda variante: Dopo aver realizzato la prima variante del gioco, aggiungere la presenza (casuale) di un altro Pokemon all'interno del Labirinto. Quando il Pokemon dell'utente raggiunge il secondo Pokemon, l'utente deciderà se farli combattere o se tornare sui propri passi.

Terza variante: Aggiungere la possibilità che il Pokemon avversario si muova (scegliendo una direzione a caso) ad ogni turno.

4) Progettazione delle classi

Analizzati i dettagli della richiesta possiamo proseguire con la progettazione delle classi necessarie, riassumibile tramite questo schema UML:

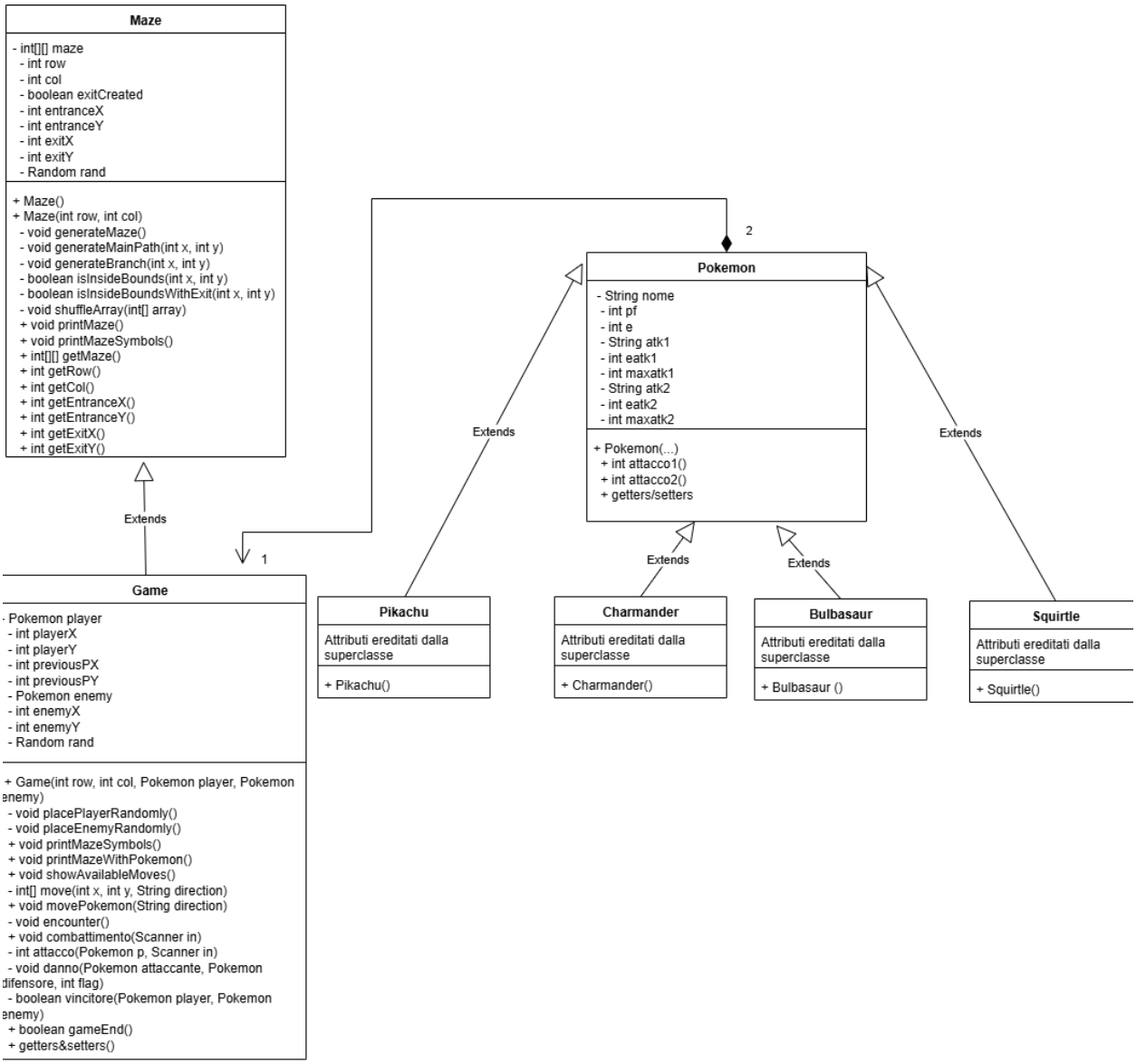


immagine 1. Grafico UML delle classi realizzate per PokéLabirinto

4.1) Relazione fra le classi

Come si può vedere dallo schema UML immagine 1 vi sono 3 classi principali, ovvero Maze, Game e Pokemon.

La classe Maze è quella che ci permette di generare un labirinto per questo motivo la classe Game eredita dalla classe Maze, poiché nella classe Game useremo il labirinto come il luogo dove avverrà il nostro gioco.

I protagonisti del nostro gioco saranno 2 Pokémon, uno del giocatore e uno avversario, il gioco prevede l'esistenza di 4 Pokemon già esistenti: Pikachu, Charmander, Bulbasaur e Squirtle che ereditano dalla classe Pokemon.

Proprio perché i Pokémon sono fondamentali per il gioco la classe Game si trova in una relazione di composizione con la classe Pokemon, nella quantità di 2 oggetti della classe Pokemon per ciascuna istanza della classe Game.

5) Pokemon

La classe Pokemon è quella che definisce le caratteristiche che ciascun pokémon del gioco dovrà avere.

Gli attributi sono:

- `private String nome, nome del pokemon.`
- `private int pf, punti ferita del pokemon.`
- `private int e, energia del pokemon. Inizializzata a zero poiché l'energia viene assegnata causalmente ad ogni turno del gioco.`
- `private String atk1, il nome del primo attacco del pokemon.`
- `private int eatk1, l'energia richiesta per eseguire il primo attacco,`
- `private int maxatk1, il quantitativo massimo di danni che il pokemon può infliggere con il suo atk1.`
- `private String atk2, il nome del secondo attacco del pokemon.`
- `private int eatk2, l'energia richiesta per eseguire il secondo attacco,`
- `private int maxatk2, il quantitativo massimo di danni che il pokemon può infliggere con il suo atk2.`

Ogni attributo ha una visibilità di tipo private.

I metodi della classe sono:

- `public Pokemon(String nome, int pf, int eatk1, int maxatk1, int eatk2, int maxatk2, String atk1, String atk2)`, metodo costruttore.
- `public int attacco1()`, metodo utilizzato per stabilire il danno inflitto da atk1 del pokémon. Danno scelto casualmente fra 0 e maxatk1.
- `public int attacco2()`, metodo utilizzato per stabilire il danno inflitto da atk1 del pokémon. Danno scelto casualmente fra 0 e maxatk2.
- `getters&setters` di ogni attributo, necessari ad accedere in maniera sicura agli attributi privati.

5.1) Codice:

```
public class Pokemon {

    private String nome;
    private int pf;
    private int e = 0;
    private String atk1;
    private int eatk1;
    private int maxatk1;
    private String atk2;
    private int eatk2;
    private int maxatk2;

    public Pokemon(String nome, int pf, int eatk1, int maxatk1, int
eatk2, int maxatk2, String atk1, String atk2) {
        this.nome = nome;
        this.pf = pf;
        this.atk1 = atk1;
```

```
        this.eatk1 = eatk1;
        this.maxatk1 = maxatk1;
        this.atk2 = atk2;
        this.eatk2 = eatk2;
        this.maxatk2 = maxatk2;
    }

    public int attacco1() {
        return (int) (Math.random() * (getMaxatk1() + 1));
    }

    public int attacco2() {
        return (int) (Math.random() * (getMaxatk1() + 1));
    }

    public String getNome() {
        return nome;
    }

    public int getPf() {
        return pf;
    }

    public void setPf(int pf) {
        this.pf = pf;
    }

    public int getE() {
        return e;
    }

    public void setE(int e) {
        this.e = e;
    }

    public int getEatk1() {
        return eatk1;
    }
```

```
    public void setEatk1(int eatk1) {
        this.eatk1 = eatk1;
    }

    public int getMaxatk1() {
        return maxatk1;
    }

    public void setMaxatk1(int maxatk1) {
        this.maxatk1 = maxatk1;
    }

    public int getEatk2() {
        return eatk2;
    }

    public void setEatk2(int eatk2) {
        this.eatk2 = eatk2;
    }

    public int getMaxatk2() {
        return maxatk2;
    }

    public void setMaxatk2(int maxatk2) {
        this.maxatk2 = maxatk2;
    }
```

5.2) Pikachu

Eredita i propri attributi dalla classe Pokemon, e presenta 1 solo metodo il costruttore.

```
public class Pikachu extends Pokemon {
```

```
    public Pikachu() {
        super("Pikachu", 35, 10, 20, 10, 20, "Thunder Shock",
"Thunderbolt");
    }
}
```

5.3) Charmander

Eredita i propri attributi dalla classe Pokemon, e presenta 1 solo metodo il costruttore.

```
public class Charmander extends Pokemon {
    public Charmander() {
        super("Charmander", 39, 12, 18, 12, 18, "Ember",
"Flamethrower");
    }
}
```

5.4) Bulbasaur

Eredita i propri attributi dalla classe Pokemon, e presenta 1 solo metodo il costruttore.

```
public class Bulbasaur extends Pokemon {
    public Bulbasaur() {
        super("Bulbasaur", 45, 10, 15, 10, 15, "Vine Whip",
"Razor Leaf");
    }
}
```

5.5) Squirtle

Eredita i propri attributi dalla classe Pokemon, e presenta 1 solo metodo il costruttore.

```
public class Squirtle extends Pokemon {
    public Squirtle() {
        super("Squirtle", 44, 8, 14, 8, 14, "Water Gun",
"Bubble");
    }
}
```

6) Maze

Maze è la classe che si occupa della generazione di un labirinto in maniera casuale.

La generazione del labirinto avviene in maniera totalmente dinamica utilizzando un approccio di backtracking utilizzando una delle sue varianti classiche, il Depth First Search, tecnica utilizzata nell'informatica per la vista di grafi e alberi che qui viene utilizzata per una matrice bidimensionale che rappresenta la griglia del labirinto.

Il labirinto è modellato con una matrice di interi i cui valori sono 0 e 1, rispettivamente 0 rappresenta una cella percorribile mentre 1 rappresenta un muro.

Il processo di costruzione del labirinto segue fasi ben definite:

1. ### Inizializzazione della griglia

All'avvio, l'intera matrice viene riempita con il valore 1, corrispondente a un labirinto completamente murato. In questo modo, l'algoritmo parte da uno spazio chiuso e successivamente "scava" i percorsi, generando progressivamente i corridoi.

2. ### Definizione dell'entrata

L'algoritmo sceglie casualmente uno dei quattro lati del labirinto (alto, basso, sinistra, destra), evitando gli angoli per evitare di posizionare l'entrata in un punto irraggiungibile. Viene aperto un varco che rappresenta l'entrata E, dal

quale prende avvio la generazione del percorso principale.

3. Generazione del percorso principale

Il cuore dell'algoritmo è il metodo `generateMainPath`, che utilizza un approccio ricorsivo.

i. A ogni chiamata, si seleziona casualmente un ordine di direzioni possibili (su, giù, destra, sinistra), ottenuto tramite la funzione di mescolamento `shuffleArray`, che si basa sullo Yales shuffle.

ii. L'algoritmo tenta quindi di spostarsi di due celle in una direzione: se la cella è ancora un muro 1 e rispetta i limiti della griglia, viene scavato un corridoio, eliminando anche il muro intermedio (per garantire connessione continua).

iii. La funzione richiama sé stessa procedendo così in profondità.

Questo meccanismo permette di generare un cammino continuo che si addentra nel labirinto. Qualora tutte le direzioni da una cella risultino invalide, perché già visitate o fuori dai limiti, la ricorsione termina in quel punto.

La funzione "torna indietro" nel call stack, riprendendo l'esplorazione da una cella precedente.

4. Definizione dell'uscita

Durante la generazione, l'algoritmo verifica se il percorso ha raggiunto una cella adiacente al bordo opposto rispetto all'entrata. Se sì, e se la distanza da quest'ultima supera una soglia minima di almeno metà della dimensione del labirinto, calcolata con la distanza di Manhattan tramite il metodo `distance`, che effettua un calcolo basato sulla distanza di Manhattan, viene creata l'uscita U. In tal modo, si evita, nella maggior parte dei casi, che l'entrata e l'uscita risultino troppo vicine, garantendo un percorso più interessante e apprezzabile ai fini del gioco.

5. Introduzione delle biforcazioni

Per evitare un labirinto composto da un unico corridoio, la classe prevede almeno tre biforcazioni aggiuntive. Tali rami vengono generati con il metodo `generateBranch`, che riprende la stessa logica ricorsiva del percorso

principale, ma viene applicata a celle facenti parte del percorso. Queste biforcazioni contribuiscono a creare vicoli ciechi e percorsi alternativi, caratteristiche che rendono il labirinto migliore dal punto di vista sia estetico che di gameplay.

6. Rappresentazione grafica

Una volta generata la struttura, il labirinto può essere stampato sia in forma numerica, 0 e 1, sia con simboli più intuitivi ovvero:

a. `E = entrata;`
b. `U = uscita;`
c. `. = cerra percorribile;`
d. `# = muro.`

6.1) Codice

Gli attributi della classe Maze sono:

- `private int[][] maze, matrice che rappresenta il labirinto.`
- `private int row, righe che costituiscono la matrice.`
- `private int col, colonne che costituiscono la matrice.`
- `private boolean exitCreated = false, variabile booleana che indica se l'uscita del labirinto è stata creata,`
- `private int entranceX, coordinate X dell'entrata.`
- `private int entranceY, coordinate Y dell'entrata.`
- `private int exitX, coordinate X dell'uscita.`
- `private int exitY, coordinate Y dell'uscita.`
- `private Random rand = new Random(), generatore di numeri casuali.`

Ogni attributo ha una visibilità di tipo private.

I metodi della classe Maze sono:

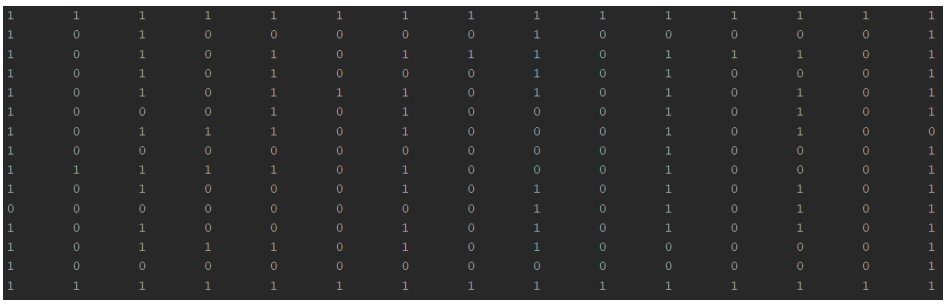
- `public Maze()`, costruttore per una matrice di dimensioni predefinite.
- `public Maze(int row, int col)`, costruttore per una matrice di dimensioni decise dall'utente.
- `private void generateMaze()`, metodo richiamato all'interno dei costruttori. Questo metodo inizializza tutta la matrice a 1, ovvero ogni cella sarà un muro e gestisce la scelta del punto d'entrata al labirinto, che corrisponde al punto da dove inizia la generazione dei corridoi del labirinto, in maniera casuale. Inoltre stabilisce il numero delle biforcazioni.
- `private void generateMainPath(int x, int y)`, metodo richiamato all'interno di `generateMaze`, che crea ricorsivamente, partendo dall'entrata del labirinto, un percorso casuale che arriva a toccare uno dei bordi, il punto raggiunto appartenente al bordo diventerà l'uscita dal labirinto.
- `private int distance(int x1, int y1, int x2, int y2)`, calcola la distanza minima tra 2 punti, ovvero 2 celle del labirinto, restituendo la distanza come un intero positivo.
- `private void generateBranch(int x, int y)`, metodo richiamato all'interno di `generateMaze`, che genera ricorsivamente una biforcazione partendo da un corridoio esistente, così creando nuovi corridoi.
- `private boolean isInsideBounds(int x, int y)`, verifica se le coordinate specificate siano all'interno dei bordi del labirinto, con i bordi inclusi, restituisce

un flag true o false a seconda se le coordinate indicate siano all'interno o meno dei limiti.

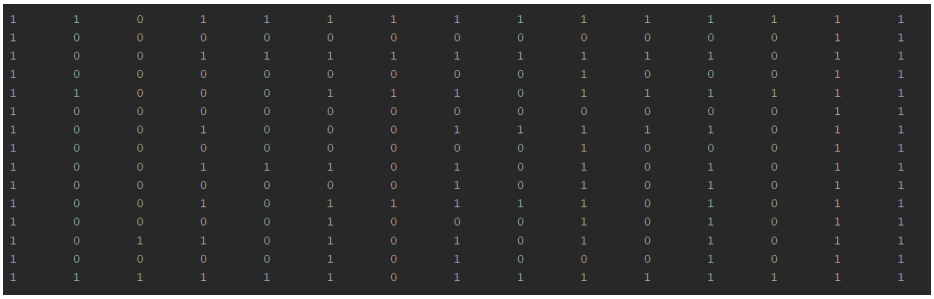
- `private boolean isInsideBoundsWithExit(int x, int y)`, verifica se le coordinate specificate siano all'interno dei bordi del labirinto, con i bordi esclusi, restituisce un flag true o false a seconda se le coordinate indicate siano all'interno o meno dei limiti.
- `private void shuffleArray(int[] array)`, metodo richiamato all'interno di `generateMainPath`, mescola casualmente gli elementi di un array.
- `public void printMaze()`, stampa a schermo la matrice rappresentando il labirinto con una sequenza numerica di 1, muri, e 0, celle percorribili.
- `public void printMazeSymbols()`, stampa a schermo la matrice seguendo una legenda di:
    - E = entrata;
    - U = uscita;
    - . = cerra percorribile;
    - # = muro.
- `getters&setters`, metodi pubblici che permettono la corretta manipolazione degli attributi privati.

Esempi di output:

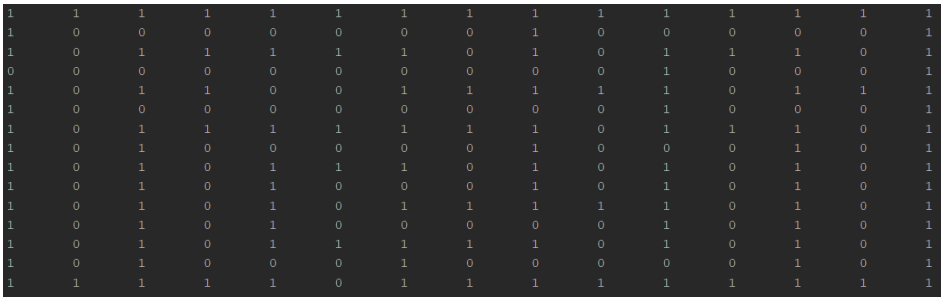
output 1



output 2



output 3



Il codice nella sua interezza:

```
public class Maze {
    private int[][] maze;
    private int row;
    private int col;
    private boolean exitCreated = false;
    private int entranceX;
    private int entranceY;
    private int exitX;
```

```
    private int exitY;
    private Random rand = new Random();
    public Maze() {
        this.row = 20;
        this.col = 20;
        maze = new int[row][col];
        generateMaze();
    }

    public Maze(int row, int col) {
        if (row > 0 && col > 0) {
            this.row = row;
            this.col = col;
            maze = new int[row][col];
            generateMaze();
        } else {
            System.out.println("ERRORE: dimensioni labirinto non
valide.");
        }
    }
    private void generateMaze() {
        for (int i = 0; i < row; i++) {
            for (int j = 0; j < col; j++) {
                maze[i][j] = 1;
            }
        }
        int side = rand.nextInt(4);

        switch (side) {
            case 0: // alto
                entranceX = 0;
                entranceY = rand.nextInt(col - 2) + 1;
                maze[entranceX][entranceY] = 0;
                maze[entranceX+1][entranceY] = 0;
                generateMainPath(entranceX+1, entranceY);
                break;
            case 1:
                entranceX = row - 1;
                entranceY = rand.nextInt(col - 2) + 1;
```

```
                maze[entranceX][entranceY] = 0;
                maze[entranceX-1][entranceY] = 0;
                generateMainPath(entranceX-1, entranceY);
                break;
            case 2:
                entranceX = rand.nextInt(row - 2) + 1;
                entranceY = 0;
                maze[entranceX][entranceY] = 0;
                maze[entranceX][entranceY+1] = 0;
                generateMainPath(entranceX, entranceY+1);
                break;
            case 3:
                entranceX = rand.nextInt(row - 2) + 1;
                entranceY = col - 1;
                maze[entranceX][entranceY] = 0;
                maze[entranceX][entranceY-1] = 0;
                generateMainPath(entranceX, entranceY-1);
                break;
        }

        int biforcazioni = 0;
        for (int i = 1; i < row-1 && biforcazioni < 3; i++) {
            for (int j = 1; j < col-1 && biforcazioni < 3; j++) {
                if (maze[i][j] == 0 && rand.nextBoolean()) {
                    generateBranch(i, j);
                    biforcazioni++;
                }
            }
        }
    }

    private void generateMainPath(int x, int y) {
        int[] dx = {-2, 2, 0, 0};
        int[] dy = {0, 0, -2, 2};
        int[] dirs = {0, 1, 2, 3};
        shuffleArray(dirs);

        for (int i = 0; i < dirs.length; i++) {
            int dir = dirs[i];
```

```
            int newX = x + dx[dir];
            int newY = y + dy[dir];

            if (!exitCreated && isInsideBoundsWithExit(newX, newY)
&& maze[newX][newY] == 1) {
                maze[(x+newX)/2][(y+newY)/2] = 0;
                maze[newX][newY] = 0;

                int minDistance = Math.min(row, col) / 2;

                if (newX == row-2 && newY > 0 && newY < col-1 &&
                    distance(newX+1, newY, entranceX, entranceY) >=
minDistance) {
                    maze[newX+1][newY] = 0;
                    exitX = newX+1;
                    exitY = newY;
                    exitCreated = true;
                } else if (newY == col-2 && newX > 0 && newX <
row-1 &&
                    distance(newX, newY+1, entranceX, entranceY) >=
minDistance) {
                    maze[newX][newY+1] = 0;
                    exitX = newX;
                    exitY = newY+1;
                    exitCreated = true;
                } else if (newX == 1 && newY > 0 && newY < col-1 &&
                    distance(newX-1, newY, entranceX, entranceY) >=
minDistance) {
                    maze[newX-1][newY] = 0;
                    exitX = newX-1;
                    exitY = newY;
                    exitCreated = true;
                } else if (newY == 1 && newX > 0 && newX < row-1 &&
                    distance(newX, newY-1, entranceX, entranceY) >=
minDistance) {
                    maze[newX][newY-1] = 0;
                    exitX = newX;
                    exitY = newY-1;
                    exitCreated = true;
```

```
                }

                generateMainPath(newX, newY);
            }
        }
    }

    private int distance(int x1, int y1, int x2, int y2) {
        return Math.abs(x1 - x2) + Math.abs(y1 - y2);
    }

    private void generateBranch(int x, int y) {
        int[] dx = {-2, 2, 0, 0};
        int[] dy = {0, 0, -2, 2};
        int[] dirs = {0, 1, 2, 3};
        shuffleArray(dirs);

        for (int i = 0; i < dirs.length; i++) {
            int dir = dirs[i];
            int newX = x + dx[dir];
            int newY = y + dy[dir];

            if (isInsideBounds(newX, newY) && maze[newX][newY] ==
1) {
                maze[(x+newX)/2][(y+newY)/2] = 0;
                maze[newX][newY] = 0;
                generateBranch(newX, newY);
            }
        }
    }

    private boolean isInsideBounds(int x, int y) {
        return x > 0 && x < row-1 && y > 0 && y < col-1;
    }

    private boolean isInsideBoundsWithExit(int x, int y) {
        return x > 0 && x <= row-2 && y > 0 && y <= col-2;
    }
```

```
    private void shuffleArray(int[] array) {
        for (int i = array.length-1; i > 0; i--) {
            int j = rand.nextInt(i+1);
            int tmp = array[i];
            array[i] = array[j];
            array[j] = tmp;
        }
    }

    public void printMaze() {
        for (int i = 0; i < row; i++) {
            for (int j = 0; j < col; j++) {
                System.out.print(this.maze[i][j] + "\t");
            }
            System.out.println();
        }
    }

    public void printMazeSymbols() {
        for (int i = 0; i < row; i++) {
            for (int j = 0; j < col; j++) {
                if (i == entranceX && j == entranceY)
System.out.print("E ");
                else if (i == exitX && j == exitY)
System.out.print("U ");
                else if (maze[i][j] == 0) System.out.print(". ");
                else System.out.print("# ");
            }
            System.out.println();
        }
    }

    public int[][] getMaze() {
        return maze;
    }

    public void setMaze(int[][] maze) {
        this.maze = maze;
    }
```

```
public int getRow() {
    return row;
}

public void setRow(int row) {
    this.row = row;
}

public int getCol() {
    return col;
}

public void setCol(int col) {
    this.col = col;
}

public boolean isExitCreated() {
    return exitCreated;
}

public void setExitCreated(boolean exitCreated) {
    this.exitCreated = exitCreated;
}

public int getEntranceX() {
    return entranceX;
}

public void setEntranceX(int entranceX) {
    this.entranceX = entranceX;
}

public int getEntranceY() {
    return entranceY;
}

public void setEntranceY(int entranceY) {
    this.entranceY = entranceY;
```

```
    }

    public int getExitX() {
        return exitX;
    }

    public void setExitX(int exitX) {
        this.exitX = exitX;
    }

    public int getExitY() {
        return exitY;
    }

    public void setExitY(int exitY) {
        this.exitY = exitY;
    }

    public Random getRand() {
        return rand;
    }

    public void setRand(Random rand) {
        this.rand = rand;
    }

}
```

7) Game

La classe Game implementa una logica di gioco al di sopra della struttura del labirinto.

7.1) Scopo

La classe Game estende la classe Maze così ereditando da essa la rappresentazione e la generazione del labirinto andando ad aggiungervi una logica di gameplay ai fini della realizzazione del PokéLabirinto, sovrapponendovi:

- Una gestione dello stato del giocatore attraverso:
  - Le caratteristiche del Pokemon;
  - coordinate attuali;
  - una memoria per la posizione precedente del giocatore.
- Gestione di un Pokemon nemico.
- Illustrazione del labirinto costantemente aggiornata.
- Meccaniche di movimento con controllo sui movimenti disponibili.
- Meccaniche di combattimento a turni.

Perché ereditarietà e non composizione?

La scelta di far ereditare a Game da Maze è dettata da una semplicità nella generazione del labirinto e nell'accesso semplificato ad attributi importanti come la posizione dell'uscita.

7.2)Codice

Gli attributi della classe Game sono:

- `private Pokemon player, ovvero il Pokémon scelto dall'utente.`
- `private int playerX, posizione X del giocatore.`
- `private int playerY, poszione Y del giocatore.`
- `private int previousPX, posizione X precedente del giocatore, al momento della generazione del giocatore in un punto casuale del labirinto previousPX corrisponde ad playerX.`
- `private int previousPY, posizione Y precedente del giocatore, al momento della generazione del giocatore in un punto casuale del labirinto previousPY corrisponde ad playerY.`
- `private Pokemon enemy, Pokémon nemico.`

- `private int enemyX, posizione X del nemico.`
- `private int enemyY, posizione Y del nemico.`
- `private Random rand = new Random(), generatore di numeri casuali.`

Ogni attributo ha una visibilità di tipo private.

I metodi della classe Game sono:

- `public Game(int row, int col, Pokemon player, Pokemon enemy), metodo costruttore che al suo interno richiama il costruttore della classe Maze, andando a creare un oggetto di tipo Game.`
- `private void placePlayerRandomly(), metodo privato richiamato all'interno del costruttore per posizionare in maniera casuale il giocatore in un corridoio del labirinto.`
- `private void placeEnemyRandomly(), metodo privato richiamato all'interno del costruttore per posizionare in maniera casuale il nemico in un corridoio del labirinto.`
- `public void printMazeSymbols(), metodo che richiama il metodo printMazeSymbols() della superclasse.`
- `public void printMazeWithPokemon(), stampa a schermo il labirinto seguendo una legenda di:`
    - `P = Pokémon del giocatore;`
    - `X = Pokémon nemico;`
    - `U = uscita;`
    - `. = cerra percorribile;`
    - `# = muero.`

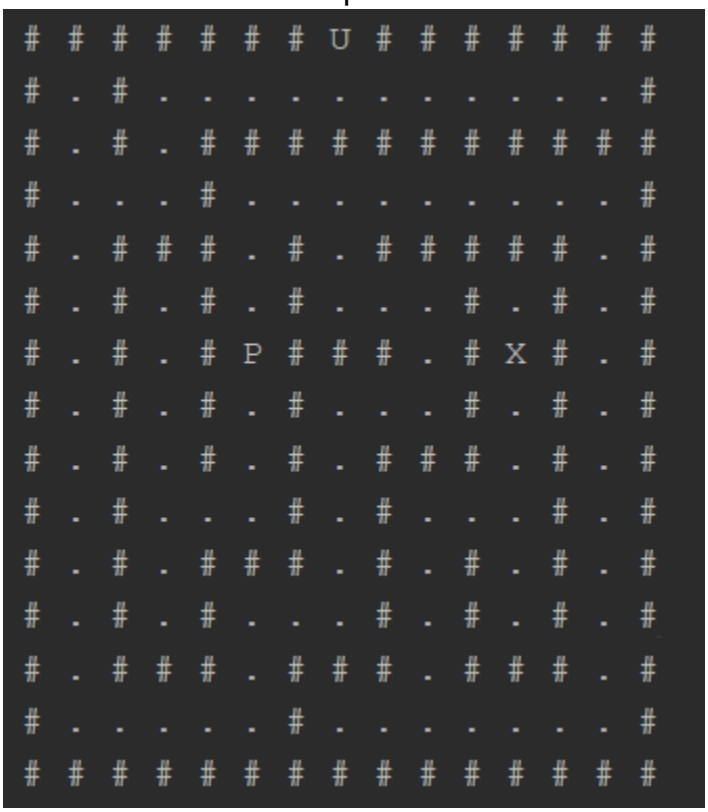
- `public void showAvailableMoves()`, metodo che mostra a schermo le mosse possibili, ovvero le direzioni in cui il giocatore può spostarsi, senza mostrare quelle bloccate da un muro, seguendo una legenda:
    - W = su;
    - A = sinistra;
    - S = giù;
    - D = destra.
- `public void movePokemon(String direction)`, metodo che in base alla direzione decisa dall'utente va ad aggiornare la posizione del giocatore salvando anche la vecchia posizione, inoltre controlla se le nuove coordinate corrispondono a quelle del Pokémon nemico.
- `public void moveEnemy()`, metodo che muove in maniera casuale il Pokémon nemico in una direzione valida, inoltre, controlla se la nuova posizione del Pokémon nemico corrisponde a quella del giocatore.
- `private int[] move(int x, int y, String direction)`, metodo privato richiamato all'interno di `movePokemon` e di `moveEnemy`, restituisce alle funzioni chiamanti un vettore di 2 elementi che corrispondono alle nuove coordinate X e Y.
- `private void encounter()`, metodo privato richiamato in `movePokemon()` e in `moveEnemy()` nel caso in cui le coordinate del giocatore corrispondano a quelle del pokémon nemico, permettendo al giocatore di decidere se dar partire un combattimento oppure se tornare indietro alla posizione precedente.
- `private void combattimento(Scanner in)`, metodo privato richiamtao in `encounter()` per la gestione del combattimento in fasi a turni, fornendo nuova energia a

ciascun Pokémon e stampando a schermo i punti ferita e l'energia rimasta a ciascun Pokémon.

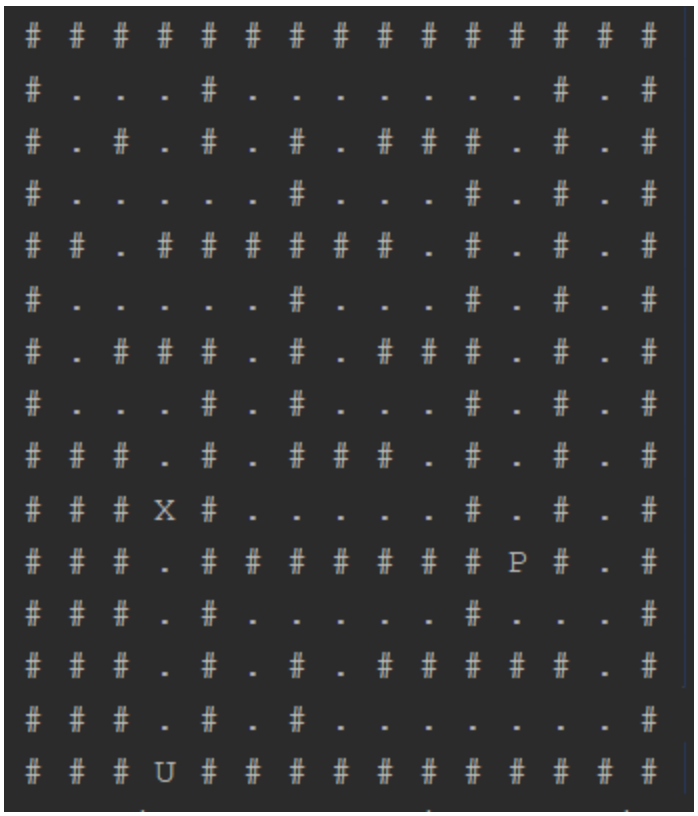
- `private int attacco(Pokemon p, Scanner in)`, metodo privato richiamato in `combattimento` che stampa a schermo gli attacchi che un Pokémon può eseguire fornendo le relative informazioni e permette all'utente di scegliere l'attacco che preferisce controllando che il Pokémon disponga dell'energia necessaria.
- `private void danno(Pokemon attaccante, Pokemon difensore, int flag)`, metodo privato richiamato all'interno di `combattimento` che applica il danno inflitto dall'attaccante al difensore, tramite una chiamata a funzioni proprie della classe Pokémon ovvero `attacco1` e `attacco2`, per poi sottrarre l'energia necessaria al Pokémon attaccante.
- `private boolean vincitore(Pokemon player, Pokemon enemy)`, metodo privato richiamato in `gameEnd`, restituisce un flag booleano che comunica alla funzione chiamante e stampa a schermo il vincitore di un combattimento, è a true se vince il giocatore a false se vince l'enemy,
- `public boolean gameEnd()`, metodo che controlla se il gioco è terminato o meno, controllando se il giocatore ha trovato l'uscita al labirinto oppure se ha perso un combattimento. Restituisce al chiamante un flag booleano, è a true se il gioco è finito mentre è a flase se il gioco non è terminato
- `getters&setters`, metodi pubblici che permettono la corretta manipolazione degli attributi privati.

Esempi di output:

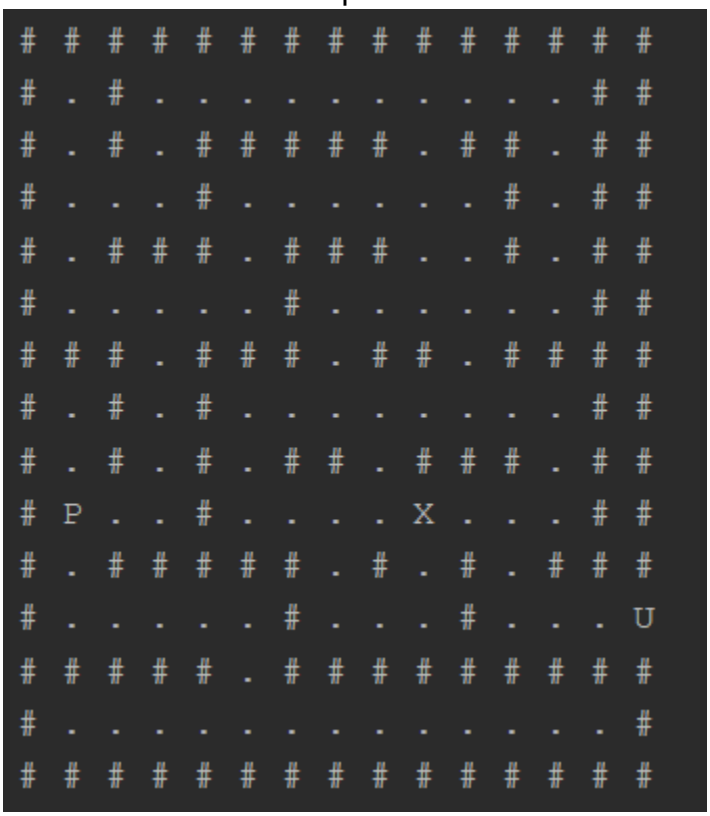
Output 1



Output 2



Output 3



Il codice nella sua interezza:

```
public class Game extends Maze {

    private Pokemon player;
    private int playerX;
    private int playerY;
    private int previousPX;
    private int previousPY;
    private Pokemon enemy;
    private int enemyX;
    private int enemyY;
    private Random rand = new Random();

    public Game(int row, int col, Pokemon player, Pokemon enemy) {
        super(row, col);
        this.player = player;
        this.enemy = enemy;
        placePlayerRandomly();
        placeEnemyRandomly();
    }

    private void placePlayerRandomly() {
        int[][] maze = getMaze();
        do {
            playerX = (int) (Math.random() * super.getRow());
            playerY = (int) (Math.random() * super.getCol());
            previousPX = playerX;
            previousPY = playerY;
        } while (maze[playerX][playerY] != 0 || (playerX ==
getExitX() && playerY == getExitY()));
    }

    private void placeEnemyRandomly() {
        int[][] maze = getMaze();
        do {
            enemyX = (int) (Math.random() * super.getRow());
```

```
            enemyY = (int) (Math.random() * super.getCol());
        } while (maze[enemyX][enemyY] != 0 || (playerX ==
getExitX() && playerY == getExitY())
                || (enemyX == playerX && enemyY == playerY));
    }

    public void printMazeSymbols(){
        super.printMazeSymbols();
        System.out.println(" 'E' corrisponde all'entrata che
corrisponde al punto di generazione del labirinto");
    }

    public void printMazeWithPokemon() {
        int[][] maze = getMaze();
        for (int i = 0; i < maze.length; i++) {
            for (int j = 0; j < maze[0].length; j++) {
                if (i == playerX && j == playerY) {
                    System.out.print("P ");
                } else if (i == enemyX && j == enemyY) {
                    System.out.print("X ");
                } else if (i == super.getExitX() && j ==
super.getExitY()) {
                    System.out.print("U ");
                } else if (maze[i][j] == 0 && !(i ==
super.getEntranceX() && j == super.getEntranceY())) {
                    System.out.print(". ");
                } else {
                    System.out.print("# ");
                }
            }
            System.out.println();
        }
    }

    public void showAvailableMoves() {
        int[][] maze = getMaze();
        System.out.println("Puoi muoverti in:");
        if (playerX > 0 && maze[playerX-1][playerY] == 0)
            System.out.println(" - su (W)");
```

```
        if (playerY > 0 && maze[playerX][playerY-1] == 0)
            System.out.println(" - sinistra (A)");

        if (playerX < maze.length-1 && maze[playerX+1][playerY] ==
0)
            System.out.println(" - giù (S)");

        if (playerY < maze[0].length-1 && maze[playerX][playerY+1]
== 0)
            System.out.println(" - destra (D)");
    }

    private int[] move(int x, int y, String direction) {
        int[][] maze = getMaze();
        int newX = x;
        int newY = y;

        switch (direction.toUpperCase()) {
            case "W":
                if (x > 0 && maze[x-1][y] == 0
                        && !((x-1) == super.getEntranceX() && y ==
super.getEntranceY())) {
                    newX = x - 1;
                }
                break;
            case "S":
                if (x < maze.length-1 && maze[x+1][y] == 0
                        && !((x+1) == super.getEntranceX() && y ==
super.getEntranceY())) {
                    newX = x + 1;
                }
                break;
            case "A":
                if (y > 0 && maze[x][y-1] == 0
                        && !(x == super.getEntranceX() && (y-1) ==
super.getEntranceY())) {
                    newY = y - 1;
                }
```

```
                break;
            case "D":
                if (y < maze[0].length-1 && maze[x][y+1] == 0
                        && !(x == super.getEntranceX() && (y+1) ==
super.getEntranceY())) {
                    newY = y + 1;
                }
                break;
            default:
                System.out.println("Errore. Mossa non valida");
        }

        return new int[]{newX, newY};
    }

    public void movePokemon(String direction) {
        int[] newPos = move(playerX, playerY,
direction.toUpperCase());
        if (newPos[0] != playerX || newPos[1] != playerY) {
            previousPX = playerX;
            previousPY = playerY;
            playerX = newPos[0];
            playerY = newPos[1];
        }

        if (playerX == enemyX && playerY == enemyY) {
            encounter();
        }
    }

    private void encounter() {
        Scanner in = new Scanner(System.in);
        String choice;
        do{
            System.out.println("Hai incontrato un Pokémon
selvatico: " + enemy.getNome() + "!");
            System.out.print("Vuoi combattere (C) o tornare
indietro (T)? ");
            choice = in.nextLine();
```

```
            if(!choice.equalsIgnoreCase("C") &&
!choice.equalsIgnoreCase("T")){
                System.out.println("Errore. Scekta non valida");
            }else{
                if (choice.equalsIgnoreCase("C")) {
                    combattimento(in);
                } else if (choice.equalsIgnoreCase("T")) {
                    System.out.println("Sei tornato indietro!");
                    playerX = previousPX;
                    playerY = previousPY;
                }
            }

        }while(!choice.equalsIgnoreCase("C") &&
!choice.equalsIgnoreCase("T"));

    }

    private void combattimento(Scanner in) {
        int flag;
        do {
            player.setE(player.getE() + (int) (Math.random() * 100
+ 1));
            enemy.setE(enemy.getE() + (int) (Math.random() * 100 +
1));

            if (Math.random() < 0.5) {
                // turno player
                flag = attacco(player, in);
                if (flag != 0)
                    danno(player, enemy, flag);
                else
                    System.out.println(player.getNome() + " non ha
energia sufficiente!");

                // turno enemy
                if (enemy.getPf() > 0) {
                    flag = attacco(enemy, in);
                    if (flag != 0) danno(enemy, player, flag);
```

```
                    else System.out.println(enemy.getNome() + "
non ha energia sufficiente!");
                }
            } else {
                // turno enemy
                flag = attacco(enemy, in);
                if (flag != 0)
                    danno(enemy, player, flag);
                else
                    System.out.println(enemy.getNome() + " non ha
energia sufficiente!");

                // turno player
                if (player.getPf() > 0) {
                    flag = attacco(player, in);
                    if (flag != 0)
                        danno(player, enemy, flag);
                    else
                        System.out.println(player.getNome() + "
non ha energia sufficiente!");
                }
            }

            System.out.println(player.getNome() + "  PF: " +
player.getPf() + ", Energia: " + player.getE());
            System.out.println(enemy.getNome() + "  PF: " +
enemy.getPf() + ", Energia: " + enemy.getE());

System.out.println("---------------------------------");

        } while (player.getPf() > 0 && enemy.getPf() > 0);

    }

    private int attacco(Pokemon p, Scanner in) {
        int choice = 0;
        System.out.println("Turno di " + p.getNome());
        System.out.println("1) " + p.getAtk1() + " (costo " +
p.getEatk1() + " energia, massimo " + p.getMaxatk1() + " danni)");
```

```
        System.out.println("2) " + p.getAtk2() + " (costo " +
p.getEatk2() + " energia, massimo " + p.getMaxatk2() + " danni)");
        int scelta = in.nextInt();

        if (scelta == 1 && p.getE() >= p.getEatk1())
            choice = 1;   // attacco 1
        if (scelta == 2 && p.getE() >= p.getEatk2())
            choice = 2;   // attacco 2

        return choice;
    }


 */
    private void danno(Pokemon attaccante, Pokemon difensore, int
flag) {
        int danni;
        if (flag == 1) {
            danni = attaccante.attacco1();
            difensore.setPf(difensore.getPf() - danni);
            attaccante.setE(attaccante.getE() -
attaccante.getEatk1());
            System.out.println(attaccante.getNome() + " usa " +
attaccante.getAtk1() + " e infligge " + danni + " danni!");
        } else {
            danni = attaccante.attacco2();
            difensore.setPf(difensore.getPf() - danni);
            attaccante.setE(attaccante.getE() -
attaccante.getEatk2());
            System.out.println(attaccante.getNome() + " usa " +
attaccante.getAtk2() + " e infligge " + danni + " danni!");
        }
    }

    private boolean vincitore(Pokemon player, Pokemon enemy) {
        boolean flag = true;
        if (player.getPf() > 0 && enemy.getPf() < 0) {
            System.out.println(player.getNome() + " vince la
battaglia!");
```

```
        } else if (player.getPf() < 0 && enemy.getPf() > 0){
            System.out.println(enemy.getNome() + " vince la
battaglia!");
            flag = false;
        }

        return flag;
    }

    public boolean gameEnd(){
        boolean flag = false;
        if(playerX == super.getExitX() && playerY ==
super.getExitY()){
            System.out.println("Uscita dal labirinto trovata !\n
Hai vinto il gioco");
            flag = true;
        }else{
            boolean isAlive;
            isAlive = vincitore(player, enemy);
            if(!isAlive){
                flag = true;
            }
        }
        return flag;
    }

    public void moveEnemy() {
        String[] directions = new String[]{"W", "S", "A", "D"};
        int[][] maze = getMaze();

        boolean moved = false;
        while (!moved) {
            String dir = directions[rand.nextInt(4)];
            int[] newPos = move(enemyX, enemyY, dir);
            if (newPos[0] != enemyX || newPos[1] != enemyY) {
                enemyX = newPos[0];
                enemyY = newPos[1];
                moved = true;
            }
```

```
        }

        if (enemyX == playerX && enemyY == playerY) {
            encounter();
        }
    }



    public Pokemon getPlayer() {
        return player;
    }

    public void setPlayer(Pokemon player) {
        this.player = player;
    }

    public int getPlayerX() {
        return playerX;
    }

    public void setPlayerX(int playerX) {
        this.playerX = playerX;
    }

    public int getPlayerY() {
        return playerY;
    }

    public void setPlayerY(int playerY) {
        this.playerY = playerY;
    }

    public int getPreviousPX() {
        return previousPX;
    }

    public void setPreviousPX(int previousPX) {
        this.previousPX = previousPX;
```

```
    }

    public int getPreviousPY() {
        return previousPY;
    }

    public void setPreviousPY(int previousPY) {
        this.previousPY = previousPY;
    }

    public int getEnemyX() {
        return enemyX;
    }

    public void setEnemyX(int enemyX) {
        this.enemyX = enemyX;
    }

    public int getEnemyY() {
        return enemyY;
    }

    public void setEnemyY(int enemyY) {
        this.enemyY = enemyY;
    }

}
```

8) Conclusione

Il progetto PokéLabirinto ha rappresentato un punto d'incontro tra teoria e pratica, permettendo di applicare concetti algoritmici e di programmazione orientata agli oggetti in un contesto ludico e interattivo. La generazione del labirinto tramite backtracking ricorsivo, l'uso dello shuffle di Fisher-Yates e dalla distanza di Manhattan, concetti usati dalla classe Maze, ha reso possibile la creazione di un ambiente dinamico.

La successiva integrazione della logica di gioco nella classe *Game* ha consentito di trasformare il labirinto in un sistema interattivo dove il giocatore deve muoversi fino all'uscita così da vincere il gioco.

PokéLabirinto dimostra come la programmazione possa essere usata per problemi astratti e per progettare esperienze coinvolgenti. Questo progetto è un esempio di come concetti matematici e informatici possano essere tradotti in un prodotto funzionante e divertente.

Invito a provare il gioco al link:

- GitHub:
  https://github.com/Adyxarex/Poke-LabirintoITA/tree/be8e9b354762f3824b7a4c5725687303cd0e65b0/Classi
- GitTea (richiede l'accesso):
  https://git.hensemberger.it/Aubeeluck/Compiti_Estivi/src/branch/main/Pokemon/PokemonEs3/Classi