НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО”
ФАКУЛЬТЕТ ІНФОРМАТИКИ ТА ОБЧИСЛЮВАЛЬНОЇ ТЕХНІКИ

Кафедра інформатики та програмної інженерії

Протокол до лабораторної роботи №2

з дисципліни

«Програмування інтелектуальних інформаційних систем»

студента 3 курсу
групи ІП-01
Пашковського Євгенія Сергійовича

Викладач:
Вавіленкова А.Д.

Київ – 2022

Завдання

1. Додати генерацію ворогів з поведінкою. При генерації ви вказуєте кількість згенерованих ворогів одного з двох типів. Перший тип шукає дорогу до гравця, другий тип рухається випадково.

2. Реалізувати алгоритми перемоги за вашою грою - minimax з та без alpha-beta pruning. Функція оцінки має оцінювати “силу” поточної позиції - чим більше число, тим краща позиція.

Загальний код

Інтерфейс, що описує позицію у будь-якому просторі:

```
export default interface Position {
  coords: {
    [key: string]: number | undefined;
  };

  getLengthTo: (position: Position) => number;
}
```

Інтерфейс, що описує внутрішні дані кожної вершини:

```
import Position from './Position';
import Vertex from '../models/Vertex';
import Enemy from '../models/Enemy';
import Player from '../models/Player';

export default interface VertexPayload<P extends Position> {
  position: P;
  previousVertex?: Vertex<P>;
  depth?: number;
  closed?: boolean;
  g?: number;
  player?: Player;
  enemy?: Enemy<P>;
}
```

Інтерфейс, що описує стратегію пошуку:

```
import Graph from '../models/Graph';
import Vertex from '../models/Vertex';
import Position from './Position';

export default interface FindPathStrategy<P extends Position> {
  findPath: (vertex1: Vertex<P>, vertex2: Vertex<P>) => Vertex<P>[];
  setGraph: (graph: Graph<P>) => void;
}
```

Інтерфейс, що описує стратегію поведінки:

```
import Position2D from '../models/Position2D';
import Vertex from '../models/Vertex';
import Position from './Position';

export default interface BehaviorStrategy<P extends Position = Position2D> {
  getNextVertex: (vertex: Vertex<P>) => Vertex<P> | undefined;
}
```

Інтерфейс об’єкту, що може рухатися:

```
export default interface Movable {
  makeMove: () => void;
}
```

Клас позиції у 2-вимірному просторі

```
import Position from '../interfaces/Position';

export default class Position2D implements Position {
  public coords: { x: number; y: number };
  constructor(x: number, y: number) {
```

```
    this.coords = { x, y };
  }
  public getLengthTo(position: Position) {
    if (
      typeof position.coords.x != 'number' ||
      typeof position.coords.y != 'number'
    )
      throw new Error('Position has inappropriate interface');
    return Math.sqrt(
      (this.coords.x - position.coords.x) ** 2 +
        (this.coords.y - position.coords.y) ** 2
    );
  }
}
```

Клас вершини графа:

```
import Position from '../interfaces/Position';
import Graph from './Graph';
import VertexPayload from '../interfaces/VertexPayload';

export default class Vertex<P extends Position> {
  protected links: Set<Vertex<P>>;
  constructor(private graph: Graph<P>, public payload: VertexPayload<P>) {
    this.links = new Set();
  }

  public setLink(vertex: Vertex<P>) {
    this.links.add(vertex);
  }

  public linkTo(vertex: Vertex<P>, value = 1): Vertex<P> {
    this.setLink(vertex);
    this.graph.linkVertices(this, vertex, value);
    return this;
  }

  public deleteLink(vertex: Vertex<P>) {
    this.links.delete(vertex);
  }

  public unlinkFrom(vertex: Vertex<P>): Vertex<P> {
    this.deleteLink(vertex);
    this.graph.unlinkVertices(this, vertex);
    return this;
  }

  public clearLinks(): Vertex<P> {
    this.links.clear();
    return this;
  }

  public isLinkedTo(vertex: Vertex<P>): boolean {
    return this.links.has(vertex);
  }

  public getLinks(): Vertex<P>[] {
    return Array.from(this.links.values());
```

```
  }

  public hasEnemy(): boolean {
    return !!this.payload.enemy;
  }

  public hasPlayer(): boolean {
    return !!this.payload.player;
  }

  public isOccupied(): boolean {
    return this.hasEnemy() || this.hasPlayer();
  }
}
```

Клас матриці:

```
export default class Matrix {
  constructor(protected arr: number[][] = []) {}

  public getElement(i: number, j: number): number {
    return this.arr[i][j];
  }
  public setElement(i: number, j: number, value: number) {
    this.arr[i][j] = value;
  }
  protected pushRow(): Matrix {
    const lastRow = this.arr[this.arr.length - 1];
    this.arr.push([]);
    if (!lastRow) return this;
    for (let i = 0; i < lastRow.length; i++) {
      this.arr[this.arr.length - 1].push(0);
    }
    return this;
  }
  protected pushCol(): Matrix {
    for (const row of this.arr) {
      row.push(0);
    }
    return this;
  }
  protected removeRow(i: number): Matrix {
    this.arr.splice(i, 1);
    return this;
  }
  protected removeCol(i: number): Matrix {
    for (const row of this.arr) {
      row.splice(i);
    }
    return this;
  }
  public getArr(): number[][] {
    const arr: number[][] = [];
    for (const row of this.arr) {
      arr.push([...row]);
    }
    return arr;
  }
```

```
}
```

Клас квадратної матриці:

```
import Matrix from './Matrix';
export default class SquaredMatrix extends Matrix {
  public pushSize(): Matrix {
    this.pushRow();
    this.pushCol();
    return this;
  }
  public removeSize(i = this.arr.length - 1): Matrix {
    this.removeRow(i);
    this.removeCol(i);
    return this;
  }
}
```

Клас графа:

```
import FindPathStrategy from '../interfaces/FindPathStrategy';
import Position from '../interfaces/Position';
import AStarPathFinder from './AStarPathFinder';
import Position2D from './Position2D';
import SquaredMatrix from './SquaredMatrix';
import Vertex from './Vertex';

export default class Graph<P extends Position = Position2D> {
  protected findPathStrategy: FindPathStrategy<P>;
  constructor(
    findPathStrategy?: FindPathStrategy<P>,
    protected vertices: Vertex<P>[] = [],
    protected matrix: SquaredMatrix = new SquaredMatrix()
  ) {
    this.findPathStrategy = findPathStrategy || new AStarPathFinder();
    this.findPathStrategy.setGraph(this);
  }

  public createVertex(position: P): Vertex<P> {
    const vertex = new Vertex<P>(this, {
      position,
    });
    this.vertices.push(vertex);
    this.matrix.pushSize();
    return vertex;
  }

  public addVertex(vertex: Vertex<P>): void {
    vertex.clearLinks();
    this.vertices.push(vertex);
    this.matrix.pushSize();
  }

  public removeVertexAt(i: number): void {
    this.vertices.splice(i, 1);
    this.matrix.removeSize(i);
  }

  public removeVertex(vertex: Vertex<P>): void {
    const i = this.vertices.indexOf(vertex);
```

```
  this.removeVertexAt(i);
}

public linkVerticesAt(i: number, j: number, value = 1): void {
  this.vertices[i].setLink(this.vertices[j]);
  this.matrix.setElement(i, j, value);
}

public linkVertices(vertex1: Vertex<P>, vertex2: Vertex<P>, value = 1): void {
  const i = this.vertices.indexOf(vertex1);
  const j = this.vertices.indexOf(vertex2);

  this.linkVerticesAt(i, j, value);
}

public unlinkVerticesAt(i: number, j: number): void {
  this.vertices[i].deleteLink(this.vertices[j]);
  this.matrix.setElement(i, j, 0);
}

public unlinkVertices(vertex1: Vertex<P>, vertex2: Vertex<P>): void {
  const i = this.vertices.indexOf(vertex1);
  const j = this.vertices.indexOf(vertex2);

  this.unlinkVerticesAt(i, j);
}

public isLinked(vertex1: Vertex<P>, vertex2: Vertex<P>): boolean {
  const i = this.vertices.indexOf(vertex1);
  const j = this.vertices.indexOf(vertex2);

  return vertex1.isLinkedTo(vertex2) && this.matrix.getElement(i, j) !== 0;
}

public setFindPathStrategy(findPathStrategy: FindPathStrategy<P>) {
  this.findPathStrategy = findPathStrategy;
  this.findPathStrategy.setGraph(this);
}

public findPath(
  vertex1: Vertex<P>,
  vertex2: Vertex<P>,
  avoid?: (vertex: Vertex<P>) => boolean
) {
  return this.findPathStrategy.findPath(vertex1, vertex2, avoid);
}

public getLength(
  vertex1: Vertex<P>,
  vertex2: Vertex<P>,
  avoid?: (vertex: Vertex<P>) => boolean
) {
  if (vertex1 === vertex2) return 0;
  const path = this.findPathStrategy.findPath(vertex1, vertex2, avoid);
  if (path.length === 2)
    return path[0].payload.position.getLengthTo(path[1].payload.position);
```

```
    let res = 0;

    for (let i = 0; i < path.length - 2; i++) {
      res += path[i].payload.position.getLengthTo(path[i + 1].payload.position);
    }

    return res;
  }

  public getVertices(): Vertex<P>[] {
    return [...this.vertices];
  }
}
```

Клас вузла колекції:

```
export default class Node<T> {
  constructor(
    public data: T,
    public priority: number = 0,
    public next?: Node<T>,
    public prev?: Node<T>
  ) {}
}
```

Клас черги:

```
import Node from './Node';
import QueueIterator from './QueueIterator';

export default class Queue<T> {
  protected firstNode?: Node<T>;
  protected lastNode?: Node<T>;

  public static from<T>(obj: Iterable<T>): Queue<T> {
    const queue = new Queue<T>();
    for (const data of obj) {
      queue.push(data);
    }
    return queue;
  }

  public push(data: T): void {
    const node = new Node(data);
    if (!this.firstNode || !this.lastNode) {
      this.firstNode = node;
      this.lastNode = node;
      return;
    }

    this.lastNode.next = node;
    this.lastNode = node;
  }

  public pushMany(...data: T[]): void {
    for (const item of data) {
      this.push(item);
    }
  }
```

```
  public pushFrom(obj: Iterable<T>): void {
    for (const item of obj) {
      this.push(item);
    }
  }

  public pull(): T | undefined {
    const buf = this.firstNode?.data;
    this.firstNode = this.firstNode?.next;
    return buf;
  }

  public isEmpty(): boolean {
    return !this.firstNode;
  }

  public toArray(): T[] {
    const result: T[] = [];
    let current = this.firstNode;
    while (current) {
      result.push(current.data);
      current = current.next;
    }
    return result;
  }

  public [Symbol.iterator](): Iterator<T, T> {
    return new QueueIterator(this);
  }
}
```

Клас ітератора черги:

```
import Queue from './Queue';

export default class QueueIterator<T> implements Iterator<T, T> {
  private array: T[];
  private i = 0;
  constructor(queue: Queue<T>) {
    this.array = queue.toArray();
  }

  public next() {
    const result = {
      value: this.array[this.i],
      done: this.i === this.array.length,
    };
    this.i++;
    return result;
  }
}
```

Клас черги з пріоритетом:

```
import Queue from './Queue';
import Node from './Node';

export default class PriorityQueue<T> extends Queue<T> {
  constructor(private asc = false) {
    super();
```

```
  }

  public push(data: T, priority = 0): void {
    const node = new Node(data, priority);
    if (!this.firstNode || !this.lastNode) {
      this.firstNode = node;
      this.lastNode = node;
      return;
    }

    if (
      this.asc
        ? this.firstNode.priority > node.priority
        : this.firstNode.priority < node.priority
    ) {
      node.next = this.firstNode;
      this.firstNode = node;
      return;
    }

    let current = this.firstNode;
    while (
      current.next &&
      (this.asc
        ? priority > current.next.priority
        : priority < current.next.priority)
    ) {
      current = current.next;
    }

    if (!current.next) {
      this.lastNode.next = node;
      this.lastNode = node;
      return;
    }

    node.next = current.next;
    current.next = node;
  }
}
```

Реалізація стратегії пошуку шляху за допомогою алгоритму Лі

```
import FindPathStrategy from '../interfaces/FindPathStrategy';
import Position from '../interfaces/Position';
import Graph from './Graph';
import Queue from './Queue';
import Vertex from './Vertex';

export default class LiPathFinder<P extends Position>
  implements FindPathStrategy<P>
{
  constructor(protected graph?: Graph<P>) {}

  public findPath(vertex1: Vertex<P>, vertex2: Vertex<P>): Vertex<P>[] {
    if (!this.graph) throw new Error('Graph is not specified');
    const queue = new Queue<Vertex<P>>();
    vertex1.payload.depth = 0;
    queue.push(vertex1);
    let found = false;
    while (!queue.isEmpty()) {
      const vertex = queue.pull();
      if (!vertex)
        throw new Error(
          'queue.isEmpty() returned false, but queue.pull() - undefined'
        );
      if (vertex === vertex2) {
        found = true;
        break;
      }
      vertex.payload.closed = true;
      const children = vertex.getLinks();
      for (const child of children) {
        const childDepth =
          (vertex.payload.depth || 0) + this.getC(vertex, child);
        if (
          !child.payload.closed &&
          childDepth <= (child.payload.depth || Infinity)
        ) {
          child.payload.depth = childDepth;
          child.payload.previousVertex = vertex;
          queue.push(child);
        }
      }
    }
    if (!found) throw new Error('Path not found');

    const path: Vertex<P>[] = [];
    let previousVertex: Vertex<P> | undefined = vertex2;
    while (previousVertex) {
      path.unshift(previousVertex);
      previousVertex = previousVertex.payload.previousVertex;
    }
    return path;
  }

  public setGraph(graph: Graph<P>): void {
    this.graph = graph;
```

```
  }

  protected getC(vertex: Vertex<P>, vertexEnd: Vertex<P>): number {
    const vertexPosition = vertex.payload.position;
    const vertexEndPosition = vertexEnd.payload.position;
    return vertexPosition.getLengthTo(vertexEndPosition);
  }
}
```

Реалізація стратегії пошуку шляху за допомогою алгоритму A*

```
import FindPathStrategy from '../interfaces/FindPathStrategy';
import Position from '../interfaces/Position';
import Graph from './Graph';
import PriorityQueue from './PriorityQueue';
import Vertex from './Vertex';

export default class AStarPathFinder<P extends Position>
  implements FindPathStrategy<P>
{
  constructor(protected graph?: Graph<P>) {}

  public findPath(vertex1: Vertex<P>, vertex2: Vertex<P>): Vertex<P>[] {
    if (!this.graph) throw new Error('Graph is not specified');
    if (!vertex1.payload.position)
      throw new Error('Vertex1 has invalid position');
    if (!vertex2.payload.position)
      throw new Error('Vertex2 has invalid position');
    const queue = new PriorityQueue<Vertex<P>>(true);
    vertex1.payload.g = 0;
    queue.push(vertex1, this.getH(vertex1, vertex2));
    let found = false;
    while (!queue.isEmpty()) {
      const vertex = queue.pull();
      if (!vertex)
        throw new Error(
          'queue.isEmpty() returned false, but queue.pull() - undefined'
        );
      if (vertex === vertex2) {
        found = true;
        break;
      }
      if (!vertex.payload.g) vertex.payload.g = 0;
      vertex.payload.closed = true;
      const children = vertex.getLinks();
      for (const child of children) {
        const g =
          vertex.payload.g +
          vertex.payload.position.getLengthTo(child.payload.position);
        if (!child.payload.closed && g < (child.payload.g || Infinity)) {
          child.payload.previousVertex = vertex;
          const h = this.getH(child, vertex2);
          child.payload.g = g;
          queue.push(child, g + h);
        }
      }
    }

    if (!found) throw new Error('Path not found');

    const path: Vertex<P>[] = [];
    let previousVertex: Vertex<P> | undefined = vertex2;
    while (previousVertex) {
      path.unshift(previousVertex);
      previousVertex = previousVertex.payload.previousVertex;
    }
```

```
    for (const vertex of this.graph.getVertices()) {
      vertex.payload.closed = undefined;
      vertex.payload.g = undefined;
      vertex.payload.previousVertex = undefined;
    }
    return path;
  }

  public setGraph(graph: Graph<P>): void {
    this.graph = graph;
  }

  protected getH(vertex: Vertex<P>, vertexEnd: Vertex<P>): number {
    const position1 = vertex.payload.position;
    const position2 = vertexEnd.payload.position;
    return position1.getLengthTo(position2);
  }
}
```

Реалізація поведінки ворогів

Клас ворога

```
import BehaviorStrategy from '../interfaces/BehaviorStrategy';
import Movable from '../interfaces/Movable';
import Position from '../interfaces/Position';
import Position2D from './Position2D';
import RandomEnemyBehavior from './RandomEnemyBehavior';
import Vertex from './Vertex';

export default class Enemy<P extends Position = Position2D> implements Movable {
  constructor(
    private behaviorStrategy: BehaviorStrategy<P>,
    private vertex: Vertex<P>
  ) {
    this.vertex.payload.enemy = this;
  }

  public makeMove() {
    const nextVertex = this.behaviorStrategy.getNextVertex(this.vertex);
    if (!nextVertex) return;
    if (nextVertex.payload.player) nextVertex.payload.player.isDead = true;
    nextVertex.payload.enemy = this;
    this.vertex.payload.enemy = undefined;
    this.vertex = nextVertex;
  }

  public makeUnsafeMove(vertex: Vertex<P>) {
    vertex.payload.enemy = this;
    this.vertex.payload.enemy = undefined;
    this.vertex = vertex;
  }

  public getPossibleMoves() {
    return this.vertex.getLinks();
  }

  public getVertex() {
    return this.vertex;
  }

  public hasRandomBehavior() {
    return this.behaviorStrategy instanceof RandomEnemyBehavior;
  }
}
```

Клас випадкової поведінки ворога

```
import BehaviorStrategy from '../interfaces/BehaviorStrategy';
import Position from '../interfaces/Position';
import Position2D from './Position2D';
import Vertex from './Vertex';

export default class RandomEnemyBehavior<P extends Position = Position2D>
  implements BehaviorStrategy<P>
{
  public getNextVertex(vertex: Vertex<P>): Vertex<P> {
    const links = vertex.getLinks();
    const index = Math.round(Math.random() * (links.length - 1));
    return links[index];
  }
}
```

Клас поведінки ворога з пошуком гравця

```
import BehaviorStrategy from '../interfaces/BehaviorStrategy';
import Graph from './Graph';
import Player from './Player';
import Position2D from './Position2D';
import Vertex from './Vertex';

export default class FindPathEnemyBehavior
  implements BehaviorStrategy<Position2D>
{
  constructor(private graph: Graph<Position2D>, private player: Player) {}

  public getNextVertex(vertex: Vertex<Position2D>) {
    try {
      const path = this.graph.findPath(
        vertex,
        this.player.getVertex(),
        (vertex) => vertex.hasEnemy()
      );
      if (path.length < 2) return;
      return path[1];
    } catch (e) {
      return;
    }
  }
}
```

Реалізація гравця та алгоритму мінімакс з та без альфа-бета відсікання

Клас гравця

```
import MinimaxProvider from '../interfaces/MinimaxProvider';
import Movable from '../interfaces/Movable';
import Labyrinth from './Labyrinth';
import PlayerMinimaxAlphaBetaBehavior from './PlayerMinimaxAlphaBetaBehavior';
import PlayerMinimaxBehavior from './PlayerMinimaxBehavior';
import Position2D from './Position2D';
import Vertex from './Vertex';

export default class Player implements Movable {
  public isDead = false;

  private playerMinimaxBehavior: MinimaxProvider;

  constructor(
    private vertex: Vertex<Position2D>,
    private labyrinth: Labyrinth,
    useOptimizedMinimax = true
  ) {
    vertex.payload.player = this;
    this.playerMinimaxBehavior = useOptimizedMinimax
      ? new PlayerMinimaxAlphaBetaBehavior()
      : new PlayerMinimaxBehavior();
  }

  public makeMove() {
    const { move } = this.playerMinimaxBehavior.minimax({
      move: undefined,
      state: this.labyrinth,
    });
    if (!move) return;

    move.payload.player = this;
    this.vertex.payload.player = undefined;
    this.vertex = move;
  }

  public makeUnsafeMove(vertex: Vertex<Position2D>) {
    this.vertex.payload.player = undefined;
    vertex.payload.player = this;
    this.vertex = vertex;
  }

  public getVertex() {
    return this.vertex;
  }
}
```

Клас поведінки гравця за допомогою мінімаксу без альфа-бета відсікання

```
import MinimaxProvider from '../interfaces/MinimaxProvider';
import Labyrinth from './Labyrinth';
import Position2D from './Position2D';
import Vertex from './Vertex';

export default class PlayerMinimaxBehavior implements MinimaxProvider {
  constructor(private maxDepth: number = 3) {}

  public minimax(
    data: {
      move: Vertex<Position2D> | undefined;
      state: Labyrinth;
    },
    depth = 0,
    isMax = true
  ): { move: Vertex<Position2D> | undefined; score: number } {
    const { move, state } = data;
    if (depth === this.maxDepth) return { move, score: this.getScore(state) };
    const nextStates = state.getPossibleStates(isMax);
    if (isMax) {
      let maxResult = { move, score: -Infinity };
      for (const state of nextStates) {
        const result = this.minimax(state, depth + 1, false);
        if (result.score > maxResult.score) maxResult = result;
      }
      return { move: move || maxResult.move, score: maxResult.score };
    }

    let minResult = { move, score: Infinity };
    for (const state of nextStates) {
      const result = this.minimax(state, depth + 1, true);
      if (result.score < minResult.score) minResult = result;
    }

    return { move, score: minResult.score };
  }

  private getScore(labyrinth: Labyrinth) {
    const playerVertex = labyrinth.getPlayer().getVertex();
    const lengthToGoal = labyrinth.getLength(playerVertex, labyrinth.goal);
    const enemies = labyrinth.getEnemies();
    const lengthToNearestEnemy = Math.min(
      ...enemies.map((enemy) =>
        labyrinth.getLength(enemy.getVertex(), playerVertex)
      )
    );

    return (
      (lengthToNearestEnemy === Infinity ? 0 : lengthToNearestEnemy) -
      (lengthToGoal === Infinity ? 0 : lengthToGoal)
    );
  }
}
```

Клас поведінки гравця за допомогою мінімаксу з альфа-бета відсіканням

```
import MinimaxProvider from '../interfaces/MinimaxProvider';
import Labyrinth from './Labyrinth';
import Position2D from './Position2D';
import Vertex from './Vertex';

export default class PlayerMinimaxAlphaBetaBehavior implements MinimaxProvider {
  constructor(private maxDepth: number = 5) {}

  public minimax(
    data: {
      move: Vertex<Position2D> | undefined;
      state: Labyrinth;
    },
    depth = 0,
    isMax = true,
    alpha = -Infinity,
    beta = Infinity
  ): { move: Vertex<Position2D> | undefined; score: number } {
    const { move, state } = data;
    if (depth === this.maxDepth) return { move, score: this.getScore(state) };
    const nextStates = state.getPossibleStates(isMax);
    if (isMax) {
      let maxResult = { move, score: -Infinity };
      for (const state of nextStates) {
        const result = this.minimax(state, depth + 1, false, alpha, beta);
        if (result.score > maxResult.score) maxResult = result;
        if (result.score > alpha) alpha = result.score;
        if (beta <= alpha) break;
      }
      return { move: move || maxResult.move, score: maxResult.score };
    }

    let minResult = { move, score: Infinity };
    for (const state of nextStates) {
      const result = this.minimax(state, depth + 1, true);
      if (result.score < minResult.score) minResult = result;
      if (result.score < beta) beta = result.score;
      if (beta <= alpha) break;
    }

    return { move, score: minResult.score };
  }

  private getScore(labyrinth: Labyrinth) {
    const playerVertex = labyrinth.getPlayer().getVertex();
    const lengthToGoal = labyrinth.getLength(playerVertex, labyrinth.goal);
    const enemies = labyrinth.getEnemies();
    const lengthToNearestEnemy = Math.min(
      ...enemies.map((enemy) =>
        labyrinth.getLength(enemy.getVertex(), playerVertex)
      )
    );

    return (
```

```
      (lengthToNearestEnemy === Infinity ? 0 : lengthToNearestEnemy) -
      (lengthToGoal === Infinity ? 0 : lengthToGoal)
    );
  }
}
```

Візуалізація роботи алгоритмів

Для візуалізації реалізуємо клас для створення лабіринту, у якому протестуємо наші алгоритми:

```
import getCombinations from '../utils/getCombinations';
import AStarPathFinder from './AStarPathFinder';
import Enemy from './Enemy';
import FindPathEnemyBehavior from './FindPathEnemyBehavior';
import Graph from './Graph';
import LiPathFinder from './LiPathFinder';
import Player from './Player';
import Position2D from './Position2D';
import RandomEnemyBehavior from './RandomEnemyBehavior';
import Vertex from './Vertex';

export default class Labyrinth {
  static VIEW_SYMBOLS = {
    XX: '▉',
    PP: 'P',
    AA: 'A',
    BB: 'B',
    ER: 'e',
    EF: 'E',
    BP: 'P',
    path: '▪',
    null: ' ',
  };

  protected graph: Graph;
  protected player: Player;
  protected enemies: Enemy[] = [];
  public goal: Vertex<Position2D>;

  constructor(
    protected arr: ('AA' | 'BB' | 'XX' | 'PP' | 'ER' | 'EF' | 'BP' | null)[][]
  ) {
    const { graph, player, enemies, b } = this.toGraph(true);
    this.graph = graph;
    this.player = player;
    this.enemies = enemies;
    this.goal = b;
  }

  public getSolution(useAStar = true): Position2D[] {
    const { a, b, graph } = this.toGraph(useAStar);
    return graph.findPath(a, b).map((vertex) => vertex.payload.position);
  }

  public draw(): string {
    let res = '';
    res += Labyrinth.VIEW_SYMBOLS.XX.repeat(this.arr[0].length + 2) + '\n';
    for (let i = 0; i < this.arr.length; i++) {
      res += Labyrinth.VIEW_SYMBOLS.XX;
      for (let j = 0; j < this.arr[i].length; j++) {
        const playerCords = this.player.getVertex().payload.position.coords;
```

```
      if (
        !this.player.isDead &&
        playerCords?.x === i &&
        playerCords?.y === j
      ) {
        res += Labyrinth.VIEW_SYMBOLS.PP;
        continue;
      }
      const foundEnemyHere = this.enemies.find((enemy) => {
        const coord = enemy.getVertex().payload.position.coords;
        return coord.x === i && coord.y === j;
      });
      if (foundEnemyHere) {
        res +=
          Labyrinth.VIEW_SYMBOLS[
            foundEnemyHere.hasRandomBehavior() ? 'ER' : 'EF'
          ];
        continue;
      }
      res += Labyrinth.VIEW_SYMBOLS[this.arr[i][j] ?? 'null'];
    }
    res += Labyrinth.VIEW_SYMBOLS.XX + '\n';
  }
  res += Labyrinth.VIEW_SYMBOLS.XX.repeat(this.arr[0].length + 2) + '\n';
  return res;
}

public drawWithSolution(useAStar = true): string {
  const solution = this.getSolution(useAStar);
  const drawnSplitted = this.draw()
    .split('\n')
    .map((row) => row.split(''));

  for (const position of solution) {
    if (
      drawnSplitted[position.coords.x][position.coords.y] ===
      Labyrinth.VIEW_SYMBOLS.null
    )
      drawnSplitted[position.coords.x][position.coords.y] =
        Labyrinth.VIEW_SYMBOLS.path;
  }

  return drawnSplitted.map((row) => row.join('')).join('\n');
}

public generateEnemies(
  count: number,
  useRandomBehavior = false
): Enemy<Position2D>[] {
  const enemies: Enemy[] = [];
  const behaviorStrategy = useRandomBehavior
    ? new RandomEnemyBehavior()
    : new FindPathEnemyBehavior(this.graph, this.player);
  const possibleVertices = this.graph
    .getVertices()
    .filter((vertex) => !vertex.isOccupied());
```

```
  if (possibleVertices.length < count)
    throw new Error('There is no enough place for so many enemies');
  for (let i = 0; i < count; i++) {
    const vertexIndex = Math.round(
      Math.random() * (possibleVertices.length - 1)
    );
    const vertex = possibleVertices[vertexIndex];
    const enemy = new Enemy(behaviorStrategy, vertex);
    enemies.push(enemy);
    possibleVertices.splice(vertexIndex, 1);
  }
  this.enemies.push(...enemies);
  return enemies;
}

public nextStep(): boolean {
  this.player.makeMove();
  this.enemies.forEach((enemy) => enemy.makeMove());
  return this.player.isDead || this.isWin;
}

public getPossibleStates(playerMove: boolean): {
  move: Vertex<Position2D> | undefined;
  state: Labyrinth;
}[] {
  if (this.isWin) return [{ move: undefined, state: this }];
  const res: { move: Vertex<Position2D> | undefined; state: Labyrinth }[] =
    [];
  if (playerMove) {
    const possibleMoves = this.player
      .getVertex()
      .getLinks()
      .filter((vertex) => !vertex.isOccupied());
    for (const possibleMove of possibleMoves) {
      const copyArr = this.arr
        .slice()
        .map((row) => row.slice())
        .map((row) =>
          row.map((item) =>
            ['EF', 'ER', 'PP', 'BP'].includes(item || '') ? null : item
          )
        );
      const { x: playerX, y: playerY } = possibleMove.payload.position.coords;
      copyArr[playerX][playerY] = 'PP';
      for (const enemy of this.enemies) {
        const { x, y } = enemy.getVertex().payload.position.coords;
        copyArr[x][y] = 'EF';
      }
      const { x: goalX, y: goalY } = this.goal.payload.position.coords;
      copyArr[goalX][goalY] =
        goalX === playerX && goalY === playerY ? 'BP' : 'BB';
      const labyrinth = new Labyrinth(copyArr);
      res.push({ move: possibleMove, state: labyrinth });
    }
  } else {
    const enemiesPossibleMoves = this.enemies.map((enemy) =>
```

```
      enemy
        .getPossibleMoves()
        .filter((vertex) => !vertex.isOccupied() && vertex !== this.goal)
    );
    const possibleCombinations = getCombinations(enemiesPossibleMoves);
    for (const possibleCombination of possibleCombinations) {
      const copyArr = this.arr
        .slice()
        .map((row) => row.slice())
        .map((row) =>
          row.map((item) =>
            ['EF', 'ER', 'PP', 'BP'].includes(item || '') ? null : item
          )
        );
      const { x: playerX, y: playerY } =
        this.player.getVertex().payload.position.coords;
      copyArr[playerX][playerY] = 'PP';
      const { x: goalX, y: goalY } = this.goal.payload.position.coords;
      copyArr[goalX][goalY] =
        goalX === playerX && goalY === playerY ? 'BP' : 'BB';
      for (let i = 0; i < this.enemies.length; i++) {
        const { x, y } = possibleCombination[i].payload.position.coords;
        copyArr[x][y] = 'EF';
      }
      const labyrinth = new Labyrinth(copyArr);
      res.push({ move: undefined, state: labyrinth });
    }
  }

  return res;
}

public getLength(vertex1: Vertex<Position2D>, vertex2: Vertex<Position2D>) {
  try {
    return this.graph.getLength(vertex1, vertex2);
  } catch (e) {
    return Infinity;
  }
}

public toGraph(useAStar = true): {
  a: Vertex<Position2D>;
  b: Vertex<Position2D>;
  graph: Graph<Position2D>;
  player: Player;
  enemies: Enemy<Position2D>[];
} {
  const findPathStrategy = useAStar
    ? new AStarPathFinder<Position2D>()
    : new LiPathFinder<Position2D>();
  const graph = new Graph(findPathStrategy);
  let a: Vertex<Position2D> | undefined;
  let b: Vertex<Position2D> | undefined;
  let p: Vertex<Position2D> | undefined;
  const enemies: Enemy[] = [];
  for (let i = 0; i < this.arr.length; i++) {
```

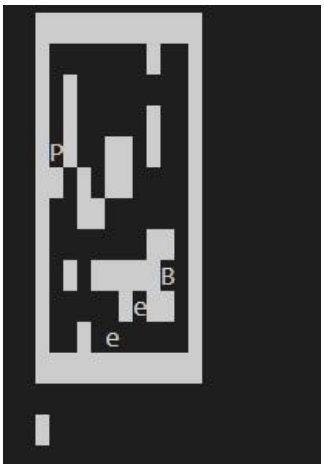
```
    for (let j = 0; j < this.arr[i].length; j++) {
      const item = this.arr[i][j];

      if (!a && item === 'AA') {
        const vertex = graph.createVertex(new Position2D(i, j));
        a = vertex;
      }
      if (!b && item === 'BB') {
        const vertex = graph.createVertex(new Position2D(i, j));
        b = vertex;
      }
      if (!p && item === 'PP') {
        const vertex = graph.createVertex(new Position2D(i, j));
        p = vertex;
      }

      if (!b && !p && item === 'BP') {
        const vertex = graph.createVertex(new Position2D(i, j));
        b = vertex;
        p = vertex;
      }
    }
  }
  const playerVertex = p || a;
  if (!playerVertex || !b) throw new Error('There is no A or B locations');

  const player = new Player(playerVertex, this);

  for (let i = 0; i < this.arr.length; i++) {
    for (let j = 0; j < this.arr[i].length; j++) {
      const item = this.arr[i][j];
      if (item === 'XX') continue;

      const vertex = graph.createVertex(new Position2D(i, j));
      if (item === 'EF') {
        enemies.push(
          new Enemy(new FindPathEnemyBehavior(graph, <Player>player), vertex)
        );
        this.arr[i][j] = null;
      }
      if (item === 'ER') {
        enemies.push(new Enemy(new RandomEnemyBehavior(), vertex));
        this.arr[i][j] = null;
      }
    }
  }

  const vertices = graph.getVertices();
  for (let i = 0; i < this.arr.length; i++) {
    for (let j = 0; j < this.arr[i].length; j++) {
      const item = this.arr[i][j];
      if (item === 'XX') continue;
      const vertex1 = vertices.filter(
        (vertex) =>
          vertex.payload.position.coords.x === i &&
          vertex.payload.position.coords.y === j
```

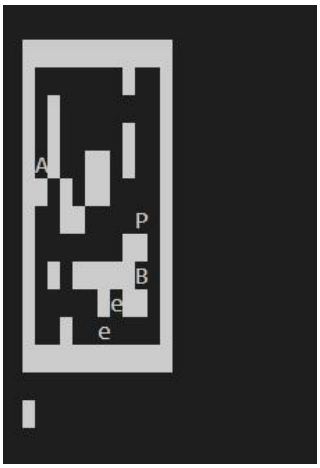
```
      )[0];
      const possibleCoords = [
        { x: i + 1, y: j + 1 },
        { x: i + 1, y: j - 1 },
        { x: i - 1, y: j + 1 },
        { x: i - 1, y: j - 1 },
        { x: i + 1, y: j },
        { x: i - 1, y: j },
        { x: i, y: j + 1 },
        { x: i, y: j - 1 },
      ];

      for (const possibleCoord of possibleCoords) {
        if (
          possibleCoord.x < 0 ||
          possibleCoord.y < 0 ||
          possibleCoord.x > this.arr.length - 1
        )
          continue;
        const possibleItem = this.arr[possibleCoord.x][possibleCoord.y];
        if (possibleItem === 'XX' || possibleItem === undefined) continue;
        const vertex2 = vertices.filter(
          (vertex) =>
            vertex.payload.position.coords.x === possibleCoord.x &&
            vertex.payload.position.coords.y === possibleCoord.y
        )[0];
        vertex1.linkTo(vertex2);
      }
    }
  }
  return { a: playerVertex, b, graph, player, enemies };
}

public getPlayer(): Player {
  return this.player;
}

public getEnemies(): Enemy<Position2D>[] {
  return this.enemies;
}

get isWin() {
  return (
    this.player
      .getVertex()
      .payload.position.getLengthTo(this.goal.payload.position) === 0
  );
}
}
```

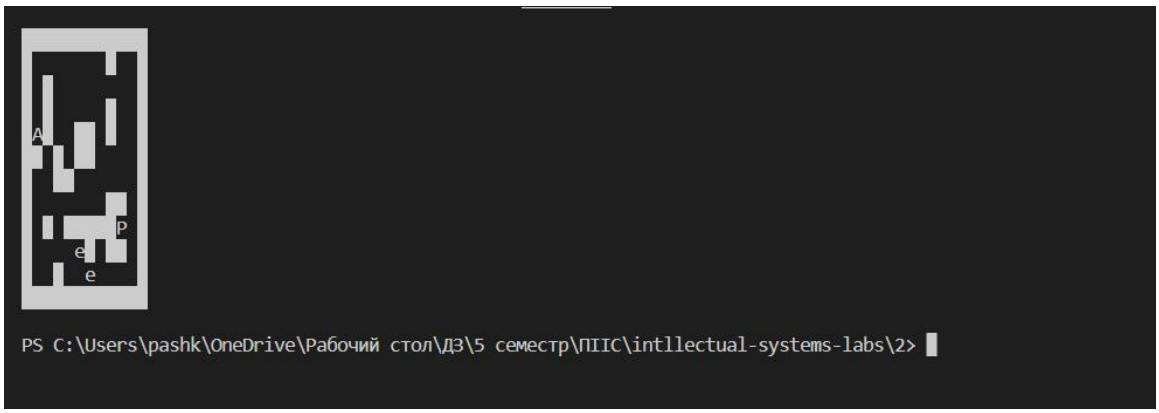
Тепер можемо запустити гру і виводити результат в консоль:

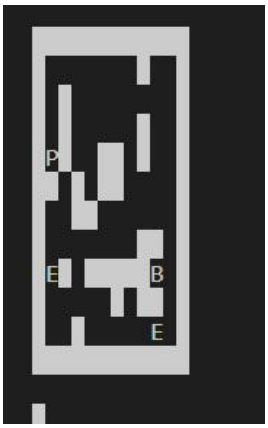
```
import Labyrinth from './models/Labyrinth';

import Labyrinth from './models/Labyrinth';
import readline from 'readline';

const labyrinth = new Labyrinth([
  //     0     1     2     3     4     5     6     7     8     9    - y
  /*0*/ [null, null, null, null, null, null, null, 'XX', null, null],
  /*1*/ [null, 'XX', null, null, null, null, null, null, null, null],
  /*2*/ [null, 'XX', null, null, null, null, null, 'XX', null, null],
  /*3*/ ['AA', 'XX', null, null, 'XX', 'XX', null, 'XX', null, null],
  /*4*/ ['XX', null, 'XX', null, 'XX', 'XX', null, null, null, null],
  /*5*/ [null, null, 'XX', 'XX', null, null, null, null, null, null],
  /*6*/ [null, null, null, null, null, null, null, 'XX', 'XX', null],
  /*7*/ [null, 'XX', null, 'XX', 'XX', 'XX', 'XX', 'XX', 'BB', null],
  /*8*/ [null, null, null, null, null, 'XX', null, 'XX', 'XX', null],
  /*9*/ [null, null, 'XX', null, null, null, null, null, null, null],
  /*x*/
]);

labyrinth.generateEnemies(2);

const readInterface = readline.createInterface({
  input: process.stdin,
  output: process.stdout,
  terminal: true,
});

const start = () => {
  readline.cursorTo(process.stdout, 0, 0);
  readline.clearScreenDown(process.stdout);
  console.log(labyrinth.draw());
};

readInterface.on('line', () => {
  const finished = labyrinth.nextStep();

  if (finished) {
    readInterface.close();
  }

  start();
});

start();
```

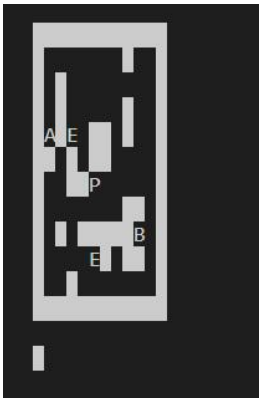
Результати роботи програми:

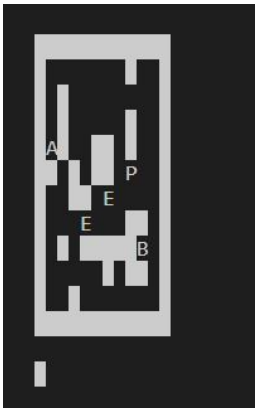







P
E
B
E


A
E
P
B
E


A
P
E
E
B


A
E
E
P
PS C:\Users\pashk\OneDrive\Рабочий стол\ДЗ\5 семестр\ПIIС\intllectual-systems-labs\2>

Висновки

Отже, виконуючи цю работу, було реалізовано алгоритм мінімакс у двох версіях (без та з альфа-бета відсіканням), було створено гру, схожу на “Pacman”, проведено дослідження та тестування алгоритмів за певних умов та станів гри.