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

Кафедра інформатики та програмної інженерії

Протокол до лабораторної роботи №1

з дисципліни

«Програмування інтелектуальних інформаційних систем»

студента 3 курсу
групи ІП-01
Пашковського Євгенія Сергійовича

Викладач:
Вавіленкова А.Д.

Київ – 2022

Завдання

Написати код (мову програмування обираєте самі), для реалізації двох алгоритмів: алгоритму Лі та алгоритму пошуку A*, розібратися в роботі алгоритмів, вміти пояснити та аргументувати написаний код, відповідати на запитання по теорії 1-2 лекцій.

Загальний код (для обох алгоритмів)

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';

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

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;
}
```

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());
  }
}
```

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>) {
    return this.findPathStrategy.findPath(vertex1, vertex2);
  }

  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);
  }
}
```

Візуалізація роботи алгоритму

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

```
import AStarPathFinder from './AStarPathFinder';
import Graph from './Graph';
import LiPathFinder from './LiPathFinder';
import Position2D from './Position2D';
import Vertex from './Vertex';

export default class Labyrinth {
  static VIEW_SYMBOLS = {
    xx: '',
    AA: 'A',
    BB: 'B',
    path: '▪',
    null: ' ',
  };

  protected arr: ('AA' | 'BB' | 'xx' | null)[][];

  constructor(arr: ('AA' | 'BB' | 'xx' | null)[][]) {
    this.arr = arr;
  }

  public getSolution(useAStar = true): 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;
    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 === 'AA') {
          a = vertex;
          continue;
        }
        if (item === 'BB') b = vertex;
      }
    }

    if (!a || !b) throw new Error('There is no A or B locations');
    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(
```

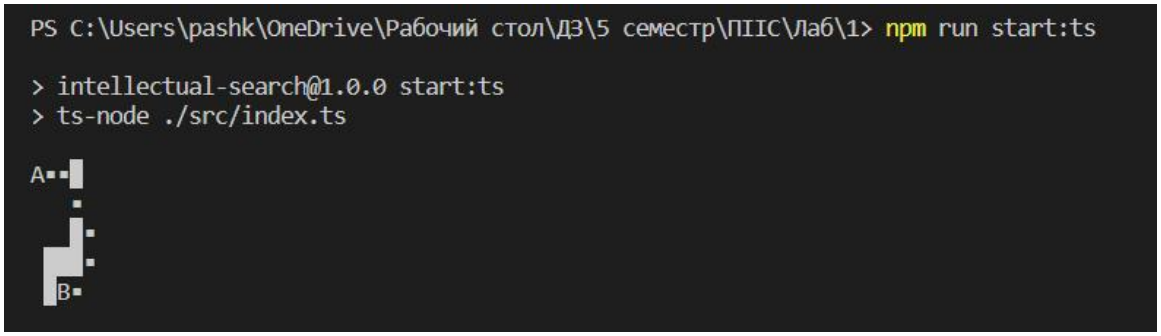
```
        (vertex) =>
          vertex.payload.position.coords.x === i &&
          vertex.payload.position.coords.y === j
      )[0];
      const possibleCoords = [
        { x: i + 1, y: j + 1 },
        { x: i + 1, y: j - 1 },
        { x: i - 1, y: j + 1 },
        { x: i - 1, y: j - 1 },
        { 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 graph.findPath(a, b).map((vertex) => vertex.payload.position);
}

public draw(): string {
  let res = '';
  for (const row of this.arr) {
    for (const item of row) {
      res += Labyrinth.VIEW_SYMBOLS[item === null ? 'null' : item];
    }
    res += '\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');
  }
}
```

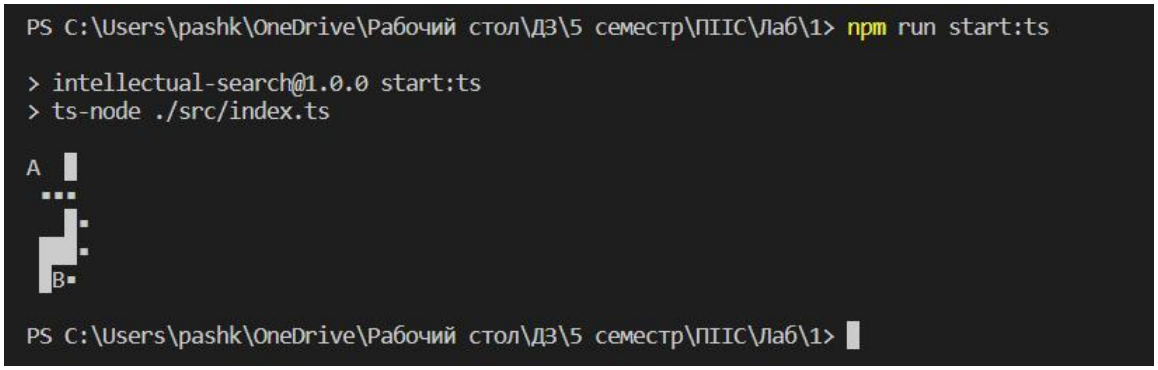
Тепер можемо запустити його і вивести результат на в консоль:

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

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

console.log(labyrinth.drawWithSolution(false));
```

Результат роботи алгоритму Лі (`labyrinth.drawWithSolution(false)`):



Результат роботи алгоритму A* (`labyrinth.drawWithSolution()` або `labyrinth.drawWithSolution(true)`) :



Як бачимо, обидва алгоритми знаходять найкоротший шлях.

Висновки

Отже, виконуючи цю работу, було реалізовано два алгоритми знаходження найкоротшого шляху, проведено їх дослідження та тестування.