Міністерство освіти і науки України
Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського"
Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

Звіт

з лабораторної роботи No 2 з дисципліни

«Проектування алгоритмів»

«Неінформативний, інформативний та локальний пошук»

Виконав(ла) *ІП-22, Андреєва Уляна Андріївна*

(шифр, прізвище, ім'я, по батькові)

Перевірив *Ахаладзе Ілля Елдарійович*

(шифр, прізвище, ім'я, по батькові)

Київ 2023

ЗМІСТ

1 МЕТА ЛАБОРАТОРНОЇ РОБОТИ

Мета роботи – розглянути та дослідити алгоритми неінформативного, інформативного та локального пошуку. Провести порівняльний аналіз ефективності використання алгоритмів.

2 ЗАВДАННЯ

Записати алгоритм розв’язання задачі у вигляді псевдокоду, відповідно до варіанту (таблиця 2.1).

Реалізувати програму, яка розв’язує поставлену задачу згідно варіанту (таблиця 2.1) за допомогою алгоритму неінформативного пошуку **АНП**, алгоритму інформативного пошуку **АІП,** що використовує задану евристичну функцію Func, або алгоритму локального пошуку **АЛП та бектрекінгу,** що використовує задану евристичну функцію Func.

Програму реалізувати на довільній мові програмування.

Увага! Алгоритм неінформативного пошуку **АНП,** реалізовується за принципом «AS IS», тобто так, як є, без додаткових модифікацій (таких як перевірка циклів, наприклад).

Провести серію експериментів для вивчення ефективності роботи алгоритмів. Кожний експеримент повинен відрізнятись початковим станом. Серія повинна містити не менше 20 експериментів для кожного алгоритму. Початковий стан зафіксувати у таблиці експериментів. За проведеними серіями необхідно визначити:

- середню кількість етапів (кроків), які знадобилось для досягнення розв’язку (ітерації);
- середню кількість випадків, коли алгоритм потрапляв в глухий кут (не міг знайти оптимальний розв’язок) – якщо таке можливе;
- середню кількість згенерованих станів під час пошуку;
- середню кількість станів, що зберігаються в пам’яті під час роботи програми.

Передбачити можливість обмеження виконання програми за часом (30 хвилин) та використання пам’яті (1 Гб).

Використані позначення:

- **8-ферзів** – Задача про вісім ферзів полягає в такому розміщенні восьми ферзів на шахівниці, що жодна з них не ставить під удар один

одного. Тобто, вони не повинні стояти в одній вертикалі, горизонталі чи діагоналі.

- **8-puzzle** – гра, що складається з 8 однакових квадратних пластинок з нанесеними числами від 1 до 8. Пластинки поміщаються в квадратну коробку, довжина сторони якої в три рази більша довжини сторони пластинок, відповідно в коробці залишається незаповненим одне квадратне поле. Мета гри – переміщаючи пластинки по коробці досягти впорядковування їх по номерах, бажано зробивши якомога менше переміщень.
- **Лабіринт** – задача пошуку шляху у довільному лабіринті від початкової точки до кінцевої з можливими випадками відсутності шляху. Структура лабіринту зчитується з файлу, або генерується програмою.
- **LDFS** – Пошук вглиб з обмеженням глибини.
- **BFS** – Пошук вшир.
- **IDS** – Пошук вглиб з ітеративним заглибленням.
- **A*** – Пошук A*.
- **RBFS** – Рекурсивний пошук за першим найкращим співпадінням.
- **F1** – кількість пар ферзів, які б’ють один одного з урахуванням видимості (ферзь A може стояти на одній лінії з ферзем B, проте між ними стоїть ферзь C; тому A не б’є B).
- **F2** – кількість пар ферзів, які б’ють один одного без урахування видимості.
- **H1** – кількість фішок, які не стоять на своїх місцях.
- **H2** – Манхетенська відстань.
- **H3** – Евклідова відстань.
- **COLOR** – Задача розфарбування карти самостійно обраної країни, не менше 20 регіонів (областей). Необхідно розфарбувати карту не більше ніж у 4 різні кольори. Мається на увазі приписування кожному

регіону власного кольору так, щоб кольори сусідніх регіонів відрізнялись. Використовувати евристичну функцію, яка повертає кількість пар суміжних вузлів, що мають однаковий колір (тобто кількість конфліктів). Реалізувати алгоритм пошуку із поверненнями (backtracking) для розв'язання поставленої задачі. Для підвищення швидкодії роботи алгоритму використати евристичну функцію, а початковим станом вважати випадкову вершину.

- **HILL –** Пошук зі сходженням на вершину з використанням із використанням руху вбік (на 100 кроків) та випадковим перезапуском (кількість необхідних разів запуску визначити самостійно).
- **ANNEAL** – Локальний пошук із симуляцією відпалу. Робоча характеристика – залежність температури T від часу роботи алгоритму t. Можна розглядати лінійну залежність: T = 1000 - k·t, де k – змінний коефіцієнт.
- **BEAM** – Локальний променевий пошук. Робоча характеристика – кількість променів k. Експерименти проводи із кількістю променів від 2 до 21.
- **MRV** – евристика мінімальної кількості значень;
- **DGR** – ступенева евристика.

Таблиця 2.1 – Варіанти алгоритмів

№	Задача	АНП	АІП	АЛП	Func
1	Лабіринт	LDFS	A*		H2

3 ВИКОНАННЯ

3.1 Псевдокод алгоритмів

```
function AStarSearch(maze, startX, startY):
    size = size of the maze
    path = 2D array of size x size filled with False
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Up, Down, Left, Right

    create an empty priority queue openSet
    add a start node (x=startX, y=startY, g=0, h=h2(startX, startY, size-2, size-2),
parent=null) to openSet
    create an empty set visitedNodes
    add the start node to visitedNodes

    while openSet is not empty:
        current = node with the lowest f value in openSet
        remove current from openSet
        remove current from visitedNodes
        x, y = current's coordinates

        if (x, y) is the goal:
            reconstructPath(path, current)
            return PathState(path, current.g)

        if path[x][y] is True:
            continue

        path[x][y] = True
```

```
        for each direction in directions:
            newX = x + direction[0]
            newY = y + direction[1]

            if (newX is within maze bounds and newY is within maze bounds and
path[newX][newY] is False and maze[newX][newY] is not a wall):
                tentativeG = current.g + 1
                h = h2(newX, newY, size-2, size-2)
                neighbor = Node(x=newX, y=newY, g=tentativeG, h=h,
parent=current)

                if (neighbor is not in visitedNodes or tentativeG <
getGValue(openSet, neighbor)):
                    add neighbor to openSet
                    add neighbor to visitedNodes

    return PathState(path, 0)

function h2(x1, y1, x2, y2):
    return abs(x1 - x2) + abs(y1 - y2)

function reconstructPath(path, goal):
    while goal is not null:
        x, y = goal's coordinates
        path[x][y] = True
        goal = goal's parent

function getGValue(set, state):
```

```
    for each node in set:
        if (node's x equals state's x and node's y equals state's y):
            return node's g
    return state's g



function LdfsSearch(maze, startX, startY):
    size = size of the maze
    path = 2D array of size x size filled with False
    steps = 0
    create an empty stack
    push the start position (x=startX, y=startY) onto the stack
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1
    while stack is not empty:
        current = pop the top position from the stack
        x, y = current's coordinates

        if (x, y) is the goal:
            steps = size * size - size of stack
            break

        if path[x][y] is True:
            continue

        path[x][y] = True

        for each direction in directions:
            newX = x + direction[0]
            newY = y + direction[1]
```

if (newX is within maze bounds and newY is within maze bounds and path[newX][newY] is False):

push the position (newX, newY) onto the stack

return PathState(path, steps)

3.1 Програмна реалізація

3.1.1 Вихідний код

MazeSolver

```
import java.io.*;
import java.util.*;

public class MazeSolver {
   private static final Random random = new Random();

   record Maze(boolean[][] grid) {

      public boolean isWall(int row, int col) {
         return grid[row][col];
      }

      public boolean[][] getGrid() {
         return grid;
      }
   }


   static class PathState {
```

```
    protected final boolean[][] path;
    private final int steps;

    public PathState(boolean[][] path, int steps) {
        this.path = path;
        this.steps = steps;
    }

    public int getRowCount() {
        return path.length;
    }

    public int getColCount() {
        return path[0].length;
    }

    public int getSteps() {
        return steps;
    }
}

public static Maze generateMaze(int size) {
    boolean[][] grid = new boolean[size][size];
    for (boolean[] row : grid) {
        Arrays.fill(row, true);
    }

    for (int i = 0; i < size; i += 2) {
        for (int j = 0; j < size; j += 2) {
```

```
        grid[i][j] = false;
    }
}

List<int[]> frontier = new ArrayList<>();
int startX = random.nextInt(size);
int startY = random.nextInt(size);
frontier.add(new int[]{startX, startY});
grid[startX][startY] = false;

while (!frontier.isEmpty()) {
    int[] current = frontier.remove(random.nextInt(frontier.size()));
    int x = current[0];
    int y = current[1];

    List<int[]> neighbors = new ArrayList<>();

    if (x >= 2) {
        neighbors.add(new int[]{x - 2, y});
    }
    if (x < size - 2) {
        neighbors.add(new int[]{x + 2, y});
    }
    if (y >= 2) {
        neighbors.add(new int[]{x, y - 2});
    }
    if (y < size - 2) {
        neighbors.add(new int[]{x, y + 2});
    }
```

```
        for (int[] neighbor : neighbors) {
            int nx = neighbor[0];
            int ny = neighbor[1];

            if (grid[nx][ny]) {
                grid[nx][ny] = false;
                frontier.add(new int[]{nx, ny});
            }
        }
    }

    return new Maze(grid);
}

static int countDeadEnds(PathState result) {
    boolean[][] path = result.path;
    int deadEnds = 0;
    int rowCount = result.getRowCount();
    int colCount = result.getColCount();

    for (int i = 1; i < rowCount - 1; i++) {
        for (int j = 1; j < colCount - 1; j++) {
            if (!path[i][j]) {
                int neighbors = countPathNeighbors(path, i, j);
                if (neighbors < 2) {
                    deadEnds++;
                }
            }
```

```
        }
    }
    return deadEnds;
}

static int countPathNeighbors(boolean[][] path, int i, int j) {
    int neighbors = 0;
    if (!path[i - 1][j]) neighbors++;
    if (!path[i + 1][j]) neighbors++;
    if (!path[i][j - 1]) neighbors++;
    if (!path[i][j + 1]) neighbors++;
    return neighbors;
}

static int countGeneratedStates(PathState result) {
    int generatedStates = 0;
    boolean[][] path = result.path;
    int rowCount = result.getRowCount();
    int colCount = result.getColCount();

    for (int i = 1; i < rowCount - 1; i++) {
        for (int j = 1; j < colCount - 1; j++) {
            if (!path[i][j]) {
                generatedStates++;
            }
        }
    }
    return generatedStates;
}
```

```
    static List<PathState> runExperiments(int numExperiments, int size, String fileName, boolean useAStar) {
        List<PathState> results = new ArrayList<>();

        for (int i = 0; i < numExperiments; i++) {
            Maze maze = generateMaze(size);
            int startX = random.nextInt(size);
            int startY = random.nextInt(size);
            PathState result = useAStar ? AStarSearch(maze, startX, startY) : LdfsSearch(maze, startX, startY);

            results.add(result);
            saveMazeToFile(maze, fileName + "_" + (i + 1) + ".txt", startX, startY, result.path);
        }

        return results;
    }

    static void deleteOldFiles(int numExperiments, String fileName) {
        for (int i = 1; i <= numExperiments; i++) {
            String filePath = fileName + "_" + i + ".txt";
            File file = new File(filePath);
            if (file.exists() && file.isFile() && file.delete()) {
                System.out.println("File " + filePath + " was deleted.");
            }
        }
    }
```

```
    static void saveMazeToFile(Maze maze, String fileName, int startX, int
startY, boolean[][] path) {
        int rowCount = maze.grid.length;
        int colCount = maze.grid[0].length;

        try (FileWriter writer = new FileWriter(fileName)) {
            for (int row = 0; row < rowCount; row++) {
                for (int col = 0; col < colCount; col++) {
                    if (row == startX && col == startY) {
                        writer.write("S ");
                    } else if (row == rowCount - 2 && col == colCount - 2) {
                        writer.write("E ");
                    } else if (path[row][col]) {
                        writer.write(". ");
                    } else if (maze.isWall(row, col)) {
                        writer.write("██");
                    } else {
                        writer.write("  ");
                    }
                }
                writer.write("\n");
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    static PathState LdfsSearch(Maze maze, int startX, int startY) {
```

```
int size = maze.grid.length;
boolean[][] path = new boolean[size][size];
int steps = 0;

Stack<Position> stack = new Stack<>();
stack.push(new Position(startX, startY));

int[][] directions = {{1, 0}, {-1, 0}, {0, 1}, {0, -1}};

while (!stack.isEmpty()) {
    Position current = stack.pop();
    int x = current.x;
    int y = current.y;

    if (x == size - 2 && y == size - 2) {
        steps = path.length * path[0].length - stack.size();
        break;
    }

    if (path[x][y]) {
        continue;
    }

    path[x][y] = true;

    for (int[] dir : directions) {
        int newX = x + dir[0];
        int newY = y + dir[1];
```

```
            if (newX >= 0 && newX < size && newY >= 0 && newY < size &&
!path[newX][newY]) {
                stack.push(new Position(newX, newY));
            }
        }
    }

    return new PathState(path, steps);
  }

  static PathState AStarSearch(Maze maze, int startX, int startY) {
    int size = maze.grid.length;
    boolean[][] path = new boolean[size][size];
    int[][] directions = {{1, 0}, {-1, 0}, {0, 1}, {0, -1}};

    PriorityQueue<AStarState> openSet = new
PriorityQueue<>(Comparator.comparingInt(AStarState::f));
    openSet.add(new AStarState(startX, startY, 0, h2(startX, startY, size - 2,
size - 2), null));

    Set<AStarState> visitedNodes = new HashSet<>();
    visitedNodes.add(openSet.peek());

    while (!openSet.isEmpty()) {
      AStarState current = openSet.poll();
      visitedNodes.remove(current);
      int x = current.x;
      int y = current.y;
```

```
        if (x == size - 2 && y == size - 2) {
            reconstructPath(path, current);
            return new PathState(path, current.g);
        }

        if (path[x][y]) {
            continue;
        }

        path[x][y] = true;

        for (int[] dir : directions) {
            int newX = x + dir[0];
            int newY = y + dir[1];

            if (newX >= 0 && newX < size && newY >= 0 && newY < size &&
!path[newX][newY] && !maze.isWall(newX, newY)) {
                int tentativeG = current.g + 1;
                int h = h2(newX, newY, size - 2, size - 2);
                AStarState neighbor = new AStarState(newX, newY, tentativeG, h,
current);

                if (!visitedNodes.contains(neighbor) || tentativeG <
getGValue(openSet, neighbor)) {
                    openSet.add(neighbor);
                    visitedNodes.add(neighbor);
                }
            }
        }
```

```
    }

    return new PathState(path, 0);
}

static int getGValue(PriorityQueue<AStarState> set, AStarState state) {
    return set.stream()
            .filter(s -> s.x == state.x && s.y == state.y)
            .findFirst()
            .map(s -> s.g)
            .orElse(state.g);
}
static int h2(int x1, int y1, int x2, int y2) {
    return Math.abs(x1 - x2) + Math.abs(y1 - y2);
}
static void reconstructPath(boolean[][] path, AStarState goal) {
    while (goal != null) {
        int x = goal.x;
        int y = goal.y;
        path[x][y] = true;
        goal = goal.parent;
    }
}

public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);

    int sizeMB = getInputSize(scanner);
    int numExperiments = getInputNumExperiments(scanner);
```

```
        String fileName = getFileName(scanner);

        List<PathState> experimentResults = new ArrayList<>();

        int algorithmChoice;

        while (true) {
            System.out.println("Choose an algorithm to use:");
            System.out.println("1. A* with Manhattan Distance (Enter '1')");
            System.out.println("2. LDFS (Enter '2')");

            if (scanner.hasNextInt()) {
                algorithmChoice = scanner.nextInt();
                if (algorithmChoice == 1 || algorithmChoice == 2) {
                    break;
                } else {
                    System.out.println("Invalid choice. Please choose 1 or 2.");
                }
            } else {
                System.out.println("Invalid input. Please enter a valid number (1 or
2).");
                scanner.next();
            }
        }

        long startTime = System.currentTimeMillis();

        if (algorithmChoice == 1) {
            experimentResults = runExperiments(numExperiments, sizeMB,
```

```
fileName, true);
        } else if (algorithmChoice == 2) {
            experimentResults = runExperiments(numExperiments, sizeMB,
fileName, false);
        }
        long endTime = System.currentTimeMillis();
        long executionTimeMillis = endTime - startTime;
        double executionTimeSeconds = (double) executionTimeMillis / 1000.0;
        System.out.println("Total execution time: " + executionTimeSeconds + "
seconds");



        if (!experimentResults.isEmpty()) {
            displayExperimentSummary(experimentResults, numExperiments);
            deleteExperimentFiles(scanner, numExperiments, fileName);
            writeMetricsToFile(experimentResults);
        }
    }

    private static int getInputSize(Scanner scanner) {
        int size = 0;

        while (size <= 0 || size > 1000) {
            try {
                System.out.print("Enter the size of the labyrinth (in cells, up to 1000):
");
                size = scanner.nextInt();
                if (size <= 0 || size > 1000) {
                    System.out.println("Labyrinth size must be between 1 and 1000
```

```
cells.");
                }
            } catch (Exception e) {
                System.out.println("Invalid input. Please enter a positive integer
between 1 and 1000.");
                scanner.next();
            }
        }

        return size;
    }

    private static int getInputNumExperiments(Scanner scanner) {
        int numExperiments = 0;

        while (numExperiments <= 0 || numExperiments > 1000) {
            try {
                System.out.print("Enter the number of experiments (up to 1000): ");
                numExperiments = scanner.nextInt();
                if (numExperiments <= 0 || numExperiments > 1000) {
                    System.out.println("The number of experiments must be between 1
and 1000.");
                }
            } catch (Exception e) {
                System.out.println("Invalid input. Please enter a positive integer
between 1 and 1000.");
                scanner.next();
            }
        }
```

```
        return numExperiments;
    }

    private static String getFileName(Scanner scanner) {
        System.out.print("Enter the filename for saving the labyrinth: ");
        return scanner.next();
    }

    private static void displayExperimentSummary(List<PathState>
experimentResults, int numExperiments) {
        int totalDeadEnds = 0;
        int totalGeneratedStates = 0;
        int totalStoredStates = 0;

        for (PathState result : experimentResults) {
            int steps = result.getSteps();
            int rowCount = result.getRowCount();
            int colCount = result.getColCount();


            int deadEnds = countDeadEnds(result);
            totalDeadEnds += deadEnds;

            int generatedStates = countGeneratedStates(result);
            totalGeneratedStates += generatedStates;

            int storedStates = rowCount * colCount;
            totalStoredStates += storedStates;
```

```
        }

        double averageDeadEnds = (double) totalDeadEnds / numExperiments;
        double averageGeneratedStates = (double) totalGeneratedStates /
numExperiments;
        double averageStoredStates = (double) totalStoredStates /
numExperiments;

        System.out.println("[SUMMARY]The average amount of dead ends: " +
averageDeadEnds);
        System.out.println("[SUMMARY]The average amount of generated states:
" + averageGeneratedStates);
        System.out.println("[SUMMARY]The average amount of the stored states:
" + averageStoredStates);
        double averageStoredStatesInMB = averageStoredStates / 1024;
        System.out.println("[SUMMARY]The average amount of the stored states
(MB): " + averageStoredStatesInMB);
    }

    private static void deleteExperimentFiles(Scanner scanner, int
numExperiments, String fileName) {
        System.out.print("Delete files (Y/N)? ");
        String confirm = scanner.next();
        if (confirm.equalsIgnoreCase("Y")) {
            deleteOldFiles(numExperiments, fileName);
            System.out.println("Files were deleted.");
        } else {
            System.out.println("Files weren't deleted.");
        }
```

```
    }

    private static void writeMetricsToFile(List<PathState> experimentResults) {
        String metricsFileName = "experiment_metrics.txt";
        try (FileWriter writer = new FileWriter(metricsFileName)) {
            for (PathState result : experimentResults) {
                writer.write("[EXPERIMENT] Steps: " + result.getSteps() + "\n");
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

AStarState

```
class AStarState {
    int x, y, g, h;
    AStarState parent;

    public AStarState(int x, int y, int g, int h, AStarState parent) {
        this.x = x;
        this.y = y;
        this.g = g;
        this.h = h;
        this.parent = parent;
    }
    public int f() {
        return g + h;
    }
}
```

Position

```
public class Position {
        int x;
        int y;

        Position(int x, int y) {
            this.x = x;
            this.y = y;
        }
    }
```

```
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import java.lang.reflect.Field;
import static org.junit.jupiter.api.Assertions.*;
```

MazeSolverTest

```
public class MazeSolverTest {
    private MazeSolver.Maze maze;
    private int size = 5;

    @BeforeEach
    public void setUp() {
        maze = MazeSolver.generateMaze(size);
    }

    @Test
    public void testGenerateMaze() {
```

```
        assertNotNull(maze);
        assertTrue(getMazeGrid(maze).length == size);
        assertTrue(getMazeGrid(maze)[0].length == size);
    }

    @Test
    public void testCountPathNeighbors() {
        boolean[][] path = new boolean[size][size];
        int neighbors = MazeSolver.countPathNeighbors(path, 2, 2);
        assertEquals(4, neighbors);
    }

    @Test
    public void testCountDeadEnds() {
        boolean[][] path = new boolean[size][size];
        int deadEnds = MazeSolver.countDeadEnds(new
MazeSolver.PathState(path, 0));
        assertEquals(0, deadEnds);
    }

    private boolean[][] getMazeGrid(MazeSolver.Maze maze) {
        try {
            Field field = MazeSolver.Maze.class.getDeclaredField("grid");
            field.setAccessible(true);
            return (boolean[][]) field.get(maze);
        } catch (NoSuchFieldException | IllegalAccessException e) {
            throw new RuntimeException(e);
        }
    }
```

```
    @Test
    public void testLdfsSearch() {
        int startX = 0;
        int startY = 0;
        MazeSolver.PathState result = MazeSolver.LdfsSearch(maze, startX,
startY);
        assertNotNull(result);
        assertTrue(result.getSteps() > 0);
    }

    @Test
    public void testAStarSearchInvalidStart() {
        int startX = 0;
        int startY = 0;
        MazeSolver.PathState pathState = MazeSolver.AStarSearch(maze, startX,
startY);

        assertNotNull(pathState);
    }

    @Test
    public void testAStarSearchNoPath() {
        int startX = 0;
        int startY = 0;
        MazeSolver.PathState pathState = MazeSolver.AStarSearch(maze, startX,
startY);
        assertNotNull(pathState);
        assertFalse(isValidPath(pathState, startX, startY));
```

```
    }
    private boolean isValidPath(MazeSolver.PathState pathState, int startX, int
startY) {
        boolean[][] path = pathState.path;
        int size = path.length;

        if (startX != 0 || startY != 0) {
            return false;
        }

        if (!path[size - 2][size - 2]) {
            return false;
        }
        return true;
    }
}
```

Таблиця 3.1 – Характеристики оцінювання алгоритму LDFS

Початкові стани	Ітерації	К-сть гл. кутів	Всього станів/сер	Всього станів у пом’яті
Стан (10;20)	50.45	0.15	14.23	100.0
Стан (20;40)	166.825	0.1	9.95	400.0
Стан(15;20)	103.55	0.01	21.45	225.0
Стан(30;50)	428.16	0.08	32.52	900.0
Стан(100;140)	4510.49	0.029	189.76	10000.0
Стан(300;400)	43902.14	0.01	155.7975	90000.0
Стан(1000;1000)	494982.585	0.004	1465.601	1000000.0

Таблиця 3.1 – Характеристики оцінювання алгоритму A* with H2

Початкові стани	Ітерації/сер	К-сть гл. кутів	Всього станів/сер	Всього станів у пом’яті
Стан (10;20)	0.95	0.2	58.75	100.0
Стан (20;40)	0.625	0.225	314.85	400.0
Стан(15;20)	0.340	0.4	157.9	225.0
Стан(30;50)	0.320	0.62	762.76	900.0
Стан(100;140)	1.05714	0.664	9512.74	10000.0
Стан(300;400)	4.08714	0.6125	88551.545	90000.0
Стан(1000;1000)	6.08714	0.507	995242.479	1000000.0

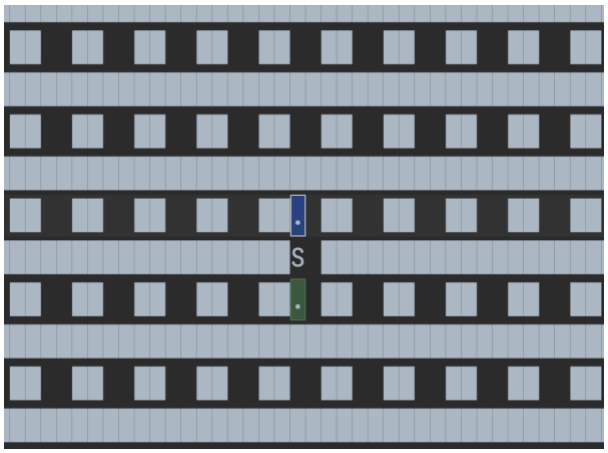
Таблиця 3.1 надає характеристики оцінювання двох алгоритмів пошуку шляху: LDFS та A* with H2, для різних розмірів лабіринтів

- випадку LDFS, ітерації та кількість глибоких кутів зазвичай збільшуються разом із збільшенням розміру лабіринту, що свідчить про збільшену складність виконання при великих лабіринтах.
- У випадку A* with H2, ітерації/сер та кількість глибоких кутів, навпаки, зазвичай залишаються стабільними навіть при зростанні розміру лабіринту.
- Всього станів та станів у пам'яті для LDFS зростають значно швидше зі збільшенням розміру лабіринту, в той час як в A* with H2 ці значення збільшуються набагато повільніше.

Загалом, A* with H2 виявляється більш ефективним та менш вимогливим щодо пам'яті алгоритмом порівняно з LDFS при пошуку шляху в великих

лабіринтах.

3.2.2 Приклади роботи



ВИСНОВОК

У ході виконання завдання я розробила програму для розв'язання різноманітних задач, використовуючи різні алгоритми пошуку, такі як АНП, АІП, АЛП та бектрекінг. Це було зацікавлюючим завданням, яке вимагало від мене розробки алгоритмів і проведення експериментів для оцінки їх ефективності.

Перш за все, я написала псевдокод для кожного із заданих алгоритмів та реалізувала ці алгоритми на вибраній мові програмування. Відмінності між цими алгоритмами дозволили мені краще зрозуміти, як вони працюють, і вибрати найбільш підходящий для кожного випадку.

Після реалізації алгоритмів я створила серію експериментів з різними початковими умовами. Кожен експеримент відрізнявся своїм початковим станом, і я записувала дані про середню кількість ітерацій, кількість потраплянь в глухі кути, кількість згенерованих станів та кількість станів у пам'яті під час роботи програми. Ці дані допомогли мені оцінити ефективність різних алгоритмів у різних умовах.