BFS/DFS as a Simple Agent in a Maze

Zbigniew Dendzik
Institute of Physics, University of Silesia

2025

In this lab we will implement simple search algorithms (BFS and DFS) as "agents" moving in a 2D maze. This is a very good illustration of "intelligent" search in the state space: the maze defines the state space; positions in the maze are states; successor function is "possible moves"; goal test is "reached target cell".

We will use Java and simple data structures from `java.util` (lists, queues, stacks, sets, maps).

Maze Representation

We consider a rectangular maze represented as a 2D array of cells. Each cell can be:

- wall – cannot enter,
- empty field – can move through.

We will also distinguish:

- start position,
- goal (target) position.

Example Encoding

We can use a char matrix, for example:

- `''` – wall,
- `'.'` – empty field,
- `'S'` – start,
- `'G'` – goal.

```
// Example maze (5x7)
char[][] maze = {
    "#######".toCharArray(),
    "#S...G#".toCharArray(),
    "#.#.#.#".toCharArray(),
    "#.....#".toCharArray(),
    "#######".toCharArray()
};
```

Position Class

We define a simple `Pozycja` (Position) class representing a cell in the maze with integer coordinates (x, y).

We will also add some helper methods.

```java
class Pozycja {
    private int x; // column index
    private int y; // row index

    public Pozycja(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() { return x; }
    public int getY() { return y; }

    @Override
    public String toString() {
        return "(" + x + "," + y + ")";
    }

    @Override
    public boolean equals(Object o) {
        if (!(o instanceof Pozycja)) return false;
        Pozycja other = (Pozycja)o;
        return this.x == other.x && this.y == other.y;
    }

    @Override
    public int hashCode() {
        return 31 * x + y;
    }
}
```

Helper Methods

We can also add static helper methods, for example to move in four directions (up, down, left, right) or to check bounds.

However, in this lab we will usually generate neighbours directly in the maze class.

Maze Class

We define a `Labirynt` (Maze) class that stores the 2D array and provides helper methods.

```java
import java.util.*;

class Labirynt {
    private char[][] grid;
    private int width;
    private int height;

    public Labirynt(char[][] grid) {
        this.grid = grid;
        this.height = grid.length;
        this.width = grid[0].length;
    }

```

```
    public int getWidth() { return width; }
    public int getHeight() { return height; }

    public char getCell(Pozycja p) {
        return grid[p.getY()][p.getX()];
    }

    public boolean isInside(Pozycja p) {
        return p.getX() >= 0 && p.getX() < width
            && p.getY() >= 0 && p.getY() < height;
    }

    public boolean isWall(Pozycja p) {
        return getCell(p) == '#';
    }

    public boolean isFree(Pozycja p) {
        char c = getCell(p);
        return c == '.' || c == 'S' || c == 'G';
    }
}
```

Finding Start and Goal

We add methods to find the start and goal positions in the maze.

```
    public Pozycja findStart() {
        for (int y = 0; y < height; y++) {
            for (int x = 0; x < width; x++) {
                if (grid[y][x] == 'S') {
                    return new Pozycja(x, y);
                }
            }
        }
        return null;
    }

    public Pozycja findGoal() {
        for (int y = 0; y < height; y++) {
            for (int x = 0; x < width; x++) {
                if (grid[y][x] == 'G') {
                    return new Pozycja(x, y);
                }
            }
        }
        return null;
    }
```

Neighbour Generation

In BFS/DFS we need to generate possible moves from a given position: up, down, left, right (no diagonal moves), staying inside the maze and avoiding walls.

```
    public List<Pozycja> neighbours(Pozycja p) {
        List<Pozycja> result = new ArrayList<>();

        int x = p.getX();
```

```
        int y = p.getY();

        Pozycja[] candidates = {
            new Pozycja(x + 1, y),
            new Pozycja(x - 1, y),
            new Pozycja(x, y + 1),
            new Pozycja(x, y - 1)
        };

        for (Pozycja q : candidates) {
            if (isInside(q) && isFree(q)) {
                result.add(q);
            }
        }
        return result;
    }
```

Path Representation

We will represent a path as a list of positions:

- input: maze, start position, goal position,
- output: `List<Pozycja>` from start to goal (including both),
- if no path: return `null` or empty list.

BFS: Breadth-First Search

BFS explores the maze level by level: first all positions at distance 1, then distance 2, etc.

BFS Properties

- Uses a queue (FIFO).
- On an unweighted grid, finds a shortest path (fewest steps).
- "Intelligent" in the sense of systematic exploration of layers.

BFS Path Finding Skeleton

We define a method:

- `List<Pozycja> znajdzSciezkeBFS(Labirynt lab, Pozycja start, Pozycja goal)`

```
public static List<Pozycja> znajdzSciezkeBFS(
        Labirynt lab, Pozycja start, Pozycja goal) {
    Queue<Pozycja> queue = new ArrayDeque<>();
    Map<Pozycja, Pozycja> parent = new HashMap<>();
    Set<Pozycja> visited = new HashSet<>();

    queue.add(start);
    visited.add(start);
    parent.put(start, null);

    while (!queue.isEmpty()) {
```

```
        Pozycja current = queue.remove();

        if (current.equals(goal)) {
            // reconstruct path
            return reconstructPath(parent, goal);
        }

        for (Pozycja next : lab.neighbours(current)) {
            if (!visited.contains(next)) {
                visited.add(next);
                parent.put(next, current);
                queue.add(next);
            }
        }
    }
    // no path
    return null;
}
```

Path Reconstruction

We reconstruct the path by following the `parent` map from goal to start.

```
private static List<Pozycja> reconstructPath(
        Map<Pozycja, Pozycja> parent, Pozycja goal) {
    List<Pozycja> path = new ArrayList<>();
    Pozycja current = goal;
    while (current != null) {
        path.add(current);
        current = parent.get(current);
    }
    Collections.reverse(path);
    return path;
}
```

DFS: Depth-First Search

DFS explores the maze by going as deep as possible along one path, then backtracking when reaching dead ends.

DFS Properties

- Uses a stack (LIFO) or recursion.
- Does not guarantee shortest path.
- Simple to implement.
- Can get "lost" in deep branches.

DFS Path Finding Skeleton

We define a method:

- `List<Pozycja> znajdzSciezkeDFS(Labirynt lab, Pozycja start, Pozycja goal)`

Iterative version with explicit stack:

```
public static List<Pozycja> znajdzSciezkeDFS(
        Labirynt lab, Pozycja start, Pozycja goal) {
    Deque<Pozycja> stack = new ArrayDeque<>();
    Map<Pozycja, Pozycja> parent = new HashMap<>();
    Set<Pozycja> visited = new HashSet<>();

    stack.push(start);
    visited.add(start);
    parent.put(start, null);

    while (!stack.isEmpty()) {
        Pozycja current = stack.pop();

        if (current.equals(goal)) {
            return reconstructPath(parent, goal);
        }

        for (Pozycja next : lab.neighbours(current)) {
            if (!visited.contains(next)) {
                visited.add(next);
                parent.put(next, current);
                stack.push(next);
            }
        }
    }
    return null;
}
```

Test Program: MazeSearch

We write a simple test program that:

- creates a maze,
- finds start and goal,
- calls BFS and DFS,
- prints the found paths.

```
public class MazeSearch {
    public static void main(String[] args) {
        char[][] grid = {
            "#######".toCharArray(),
            "#S...G#".toCharArray(),
            "#.#.#.#".toCharArray(),
            "#.....#".toCharArray(),
            "#######".toCharArray()
        };

        Labirynt lab = new Labirynt(grid);
        Pozycja start = lab.findStart();
        Pozycja goal  = lab.findGoal();

        System.out.println("Start: " + start);
        System.out.println("Goal:  " + goal);

        List<Pozycja> pathBFS = znajdzSciezkeBFS(lab, start, goal);
```

```
        System.out.println("BFS path: " + pathBFS);

        List<Pozycja> pathDFS = znajdzSciezkeDFS(lab, start, goal);
        System.out.println("DFS path: " + pathDFS);
    }
}
```

Exercise 5.1: Maze Representation

- Create a `Labirynt` class as shown above.
- Implement methods:
  - `getWidth()`, `getHeight()`,
  - `isInside(Pozycja p)`,
  - `isWall(Pozycja p)`,
  - `isFree(Pozycja p)`,
  - `findStart()`, `findGoal()`.
- Test: create a small maze and print the coordinates of start and goal.

Exercise 5.2: Neighbours and Position Class

- Implement the `Pozycja` class with:
  - private fields `x`, `y`,
  - constructor `Pozycja(int x, int y)`,
  - getters `getX()`, `getY()`,
  - `toString()`,
  - `equals()` and `hashCode()`.
- Implement method `List<Pozycja> neighbours(Pozycja p)` in `Labirynt`, returning all free neighbour cells (up, down, left, right).
- Test: for a chosen position in the maze, print the list of neighbours.

Exercise 5.3: BFS Path Search

- Implement method:
  - `List<Pozycja> znajdzSciezkeBFS(Labirynt lab, Pozycja start, Pozycja goal)`.
- Use:
  - `Queue<Pozycja>` (e.g. `ArrayDeque`),
  - `Set<Pozycja>` for visited positions,
  - `Map<Pozycja,Pozycja>` to store parents.
- When goal is found, reconstruct the path using the `parent` map.
- Test: print the BFS path and its length.
- Think: why is the BFS path shortest (fewest steps)?

Exercise 5.4: DFS Path Search

- Implement method:
  - `List<Pozycja> znajdzSciezkeDFS(Labirynt lab, Pozycja start, Pozycja goal)`.
- Use:
  - `Deque<Pozycja>` as a stack (methods `push()`, `pop()`),
  - same `visited` set and `parent` map as in BFS.
- Test: compare DFS path with BFS path:
  - Are they the same?
  - Which one is shorter?
  - How does order of neighbours influence DFS path?

Exercise 5.5: Simple Heuristic and A* Idea

BFS and DFS do not use any information about where the goal is located. They are "blind" search strategies.

We can introduce a simple heuristic:

- heuristic value $h(p)$ = Manhattan distance from position p to goal:

$$h(p) = |x_p - x_{\text{goal}}| + |y_p - y_{\text{goal}}|.$$

Heuristic-Guided Search (Informal)

- Instead of a simple queue (BFS), we can use a priority queue ordered by:

$$f(p) = g(p) + h(p),$$

  where:
  - $g(p)$ = distance (number of steps) from start to p,
  - $h(p)$ = heuristic estimate to goal.
- Always expand the position with smallest $f(p)$.
- This is the basic idea of the A* algorithm (without full formal proof and conditions).

Task

- Extend your code with a method:
  - `List<Pozycja> znajdzSciezkeHeurystyczna(...)`

  which uses a `PriorityQueue` ordered by $f(p) = g(p) + h(p)$.
- Compute $g(p)$ as number of steps from start (you can store it in a `Map<Pozycja,Integer>`).
- Use Manhattan distance as heuristic $h(p)$.
- Compare:
  - number of visited nodes for BFS and heuristic search,

  - length of found path.

- Think: in which situations heuristic search expands fewer states than BFS?

Discussion:

- How do BFS and DFS behave in large mazes?
- How does the choice of data structure (queue, stack, priority queue) change the "personality" of the search agent?
- How can we visualize the path found by the agent in the maze (e.g. by marking the path with `'*'` characters)?