BFS/DFS as Simple Agent in a Maze (State Space Search)

Zbigniew Dendzik

March 2025

Lecture Plan

- Introduction to state space search
- Maze as a state space problem
- Graph search algorithms: BFS and DFS
- Maze representation in 2D array
- Pozycja class – position in maze
- Labirynt class – maze structure
- BFS implementation – breadth-first search
- DFS implementation – depth-first search
- Path reconstruction
- Heuristic search and A* introduction
- Comparison: BFS vs DFS vs A*

Introduction to State Space Search

Goal: Find a path from initial state to goal state.

State space:

- Set of all possible states
- Transitions between states (actions)
- Initial state
- Goal state(s)

Application examples:

- Maze solving – position in maze
- Puzzle solving (8-puzzle, Rubik's cube)
- Route planning – cities on map
- Game playing – board configurations

Intelligent Agent Concept

Agent: An entity that perceives environment and acts.

Rational agent:

- Perceives current state
- Has a goal
- Chooses actions to achieve goal
- Uses search algorithm to plan

Maze agent:

- Perceives: current position, walls, goal
- Goal: reach target position
- Actions: move up, down, left, right
- Search: BFS/DFS to find path

Why State Space Search?

Fundamental AI technique:

- One of the oldest AI methods (1950s)
- Foundation for planning and problem-solving
- Used in robotics, games, logistics

Modern applications:

- GPS navigation – shortest path
- Robot motion planning – obstacle avoidance
- Game AI – move selection
- Network routing – packet delivery

Educational value:

- Illustrates graph algorithms
- Shows search strategies
- Introduces heuristics and optimization

Maze as State Space Problem

State: Position (x, y) in maze

Initial state: Start position S

Goal state: Target position G

Actions: Move to adjacent cells

- Up: $(x, y) \rightarrow (x, y - 1)$
- Down: $(x, y) \rightarrow (x, y + 1)$
- Left: $(x, y) \rightarrow (x - 1, y)$
- Right: $(x, y) \rightarrow (x + 1, y)$

Constraints:

- Cannot move through walls
- Must stay inside maze bounds

Maze Representation

2D character array:

- `'#'` – wall (impassable)
- `'.'` – empty field (passable)
- `'S'` – start position
- `'G'` – goal position

Example maze (5x7):

```
#######
#S...G#
#.#.#.#
#.....#
#######
```

Coordinates:

- x – column index (horizontal)
- y – row index (vertical)
- Origin $(0, 0)$ at top-left corner

Graph View of Maze

Maze = Graph:

- Vertices = passable cells
- Edges = possible moves between adjacent cells
- Start vertex = S
- Goal vertex = G

Graph properties:

- Undirected (can move both ways)
- Unweighted (all moves have cost 1)
- Sparse (each vertex has max 4 neighbors)

Path finding:

- Find sequence of vertices from S to G
- Graph search algorithms: BFS, DFS

Search Algorithms Overview

Uninformed search (blind search):

- No knowledge about goal location
- Systematic exploration
- Examples: BFS, DFS

Informed search (heuristic search):

- Uses heuristic to guide search
- Estimates distance to goal
- Examples: Greedy search, A*

Key differences:

- BFS – explores layer by layer (queue)
- DFS – explores deep paths first (stack)
- A* – explores promising paths first (priority queue)

BFS: Breadth-First Search

Strategy: Explore all neighbors before going deeper.

Data structure: Queue (FIFO)

Algorithm:

1. Start with initial position in queue
2. While queue not empty:
   - Dequeue position
   - If goal found, return path
   - Enqueue all unvisited neighbors
3. If queue empty, no path exists

Properties:

- Complete – finds solution if exists
- Optimal – finds shortest path (unweighted graph)
- Time: $O(V + E)$ where V = vertices, E = edges
- Space: $O(V)$ – stores all visited nodes

BFS Visual Example

Exploration order (distance from start):

```
#######
#S123G#
#2#4#5#
#34567#
#######
```

Layer-by-layer expansion:

- Layer 0: S (start)
- Layer 1: neighbors of S
- Layer 2: neighbors of layer 1
- Layer 3: neighbors of layer 2
- Found: G at layer 4

DFS: Depth-First Search

Strategy: Explore as deep as possible, then backtrack.

Data structure: Stack (LIFO) or recursion

Algorithm:

1. Start with initial position on stack
2. While stack not empty:
   - Pop position
   - If goal found, return path
   - Push all unvisited neighbors
3. If stack empty, no path exists

Properties:

- Complete – finds solution (with cycle detection)
- Not optimal – path may be longer than BFS
- Time: $O(V + E)$
- Space: $O(V)$ in worst case, often less

DFS Visual Example

Exploration order (depth-first):

```
#######
#S123G#
#4#56#7#
#89ABC#
#######
```

Deep path exploration:

- Goes deep along one path
- Backtracks when hitting dead end
- May find longer path than BFS
- Order depends on neighbor ordering

BFS vs DFS Comparison

Aspect	**BFS**	**DFS**
Data structure	Queue	Stack
Exploration	Layer by layer	Deep first
Path optimality	Shortest	May be long
Memory	More	Less
Implementation	Iterative	Iterative/Recursive
Best for	Shortest path	Path existence

When to use:

- BFS – need shortest path
- DFS – only need any path, memory limited

System Architecture

System components:

1. **Data representation**
   - `Pozycja` – position (x, y) in maze
   - `Labirynt` – maze structure and operations

2. **Search algorithms**
   - `znajdzSciezkeBFS()` – BFS path finding
   - `znajdzSciezkeDFS()` – DFS path finding
   - `znajdzSciezkeHeurystyczna()` – heuristic search

3. **Helper structures**
   - `Queue` / `Stack` – frontier management
   - `Set` – visited positions
   - `Map` – parent tracking for path reconstruction

Pozycja Class

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

    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 + ")";
    }

    // equals() and hashCode() required for Set/Map
}
```

Pozycja Class – equals and hashCode

Why needed:

- Used in `Set<Pozycja>` for visited tracking
- Used in `Map<Pozycja, Pozycja>` for parent tracking
- Java requires proper equality comparison

Implementation:

- `equals()` – compare x and y coordinates
- `hashCode()` – combine x and y into hash
- Common pattern: `31 * x + y`

Contract:

- If `a.equals(b)` then `a.hashCode() == b.hashCode()`
- Critical for correct `HashSet` and `HashMap` behavior

Pozycja – equals and hashCode Implementation

```java
@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;
}
```

Usage:

```java
Set<Pozycja> visited = new HashSet<>();
visited.add(new Pozycja(1, 2));
visited.contains(new Pozycja(1, 2)); // true
```

Labirynt Class – Structure

```
1  import java.util.*;
2
3  class Labirynt {
4      private char[][] grid;
5      private int width;
6      private int height;
7
8      public Labirynt(char[][] grid) {
9          this.grid = grid;
10         this.height = grid.length;
11         this.width = grid[0].length;
12     }
13
14     public int getWidth() { return width; }
15     public int getHeight() { return height; }
16
17     public char getCell(Pozycja p) {
18         return grid[p.getY()][p.getX()];
19     }
20 }
```

Labirynt – Validation Methods

```
1  public boolean isInside(Pozycja p) {
2      return p.getX() >= 0 && p.getX() < width
3          && p.getY() >= 0 && p.getY() < height;
4  }
5
6  public boolean isWall(Pozycja p) {
7      return getCell(p) == '#';
8  }
9
10 public boolean isFree(Pozycja p) {
11     char c = getCell(p);
12     return c == '.' || c == 'S' || c == 'G';
13 }
```

Purpose:

- Check position validity before moving
- Avoid array index out of bounds
- Filter valid neighbors

Labirynt – Finding Start and Goal

```
1  public Pozycja findStart() {
2      for (int y = 0; y < height; y++) {
3          for (int x = 0; x < width; x++) {
4              if (grid[y][x] == 'S') {
5                  return new Pozycja(x, y);
6              }
7          }
8      }
9      return null;
10 }
11
12 public Pozycja findGoal() {
13     for (int y = 0; y < height; y++) {
14         for (int x = 0; x < width; x++) {
15             if (grid[y][x] == 'G') {
16                 return new Pozycja(x, y);
17             }
18         }
19     }
20     return null;
21 }
```

Labirynt – Neighbour Generation

```java
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),   // right
        new Pozycja(x - 1, y),   // left
        new Pozycja(x, y + 1),   // down
        new Pozycja(x, y - 1)    // up
    };

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

Neighbour Generation – Explanation

Four-directional movement:

- Right: $(x + 1, y)$
- Left: $(x - 1, y)$
- Down: $(x, y + 1)$
- Up: $(x, y - 1)$

Filtering:

- Check if inside maze bounds
- Check if not a wall
- Return only valid neighbors

Order matters:

- DFS path depends on neighbor order
- BFS always finds shortest (order irrelevant)

Path Representation

Path = sequence of positions from start to goal

Data structure: `List<Pozycja>`

Example path:

- Start: $(1, 1)$
- Step 1: $(2, 1)$
- Step 2: $(3, 1)$
- Step 3: $(4, 1)$
- Goal: $(5, 1)$

Path length: Number of steps = list size - 1

Empty path: `null` or empty list when no solution exists

Parent Tracking for Path Reconstruction

Problem: How to remember the path while searching?

Solution: Parent map

Data structure: `Map<Pozycja, Pozycja>`

- Key: current position
- Value: parent position (where we came from)

Example:

- `parent.put((2,1), (1,1))` – came to (2,1) from (1,1)
- `parent.put((3,1), (2,1))` – came to (3,1) from (2,1)
- `parent.put((1,1), null)` – start has no parent

Reconstruction: Follow parent links backward from goal to start

BFS Implementation – Setup

```
1  public static List<Pozycja> znajdzSciezkeBFS(
2          Labirynt lab, Pozycja start, Pozycja goal) {
3
4      Queue<Pozycja> queue = new ArrayDeque<>();
5      Map<Pozycja, Pozycja> parent = new HashMap<>();
6      Set<Pozycja> visited = new HashSet<>();
7
8      queue.add(start);
9      visited.add(start);
10     parent.put(start, null);
11
12     // main loop...
13 }
```

Initialization:

- Queue with start position
- Mark start as visited
- Start has no parent

BFS Implementation – Main Loop

```
while (!queue.isEmpty()) {
    Pozycja current = queue.remove();

    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);
            queue.add(next);
        }
    }
}

return null; // no path found
```

BFS Algorithm – Step by Step

Iteration:

1. Dequeue current position
2. Check if goal reached
3. If yes, reconstruct and return path
4. If no, explore neighbors:
   - Skip already visited
   - Mark as visited
   - Record parent
   - Enqueue for later exploration
5. Repeat until queue empty or goal found

Termination:

- Success: goal found
- Failure: queue empty (no path exists)

Path Reconstruction

```
1  private static List<Pozycja> reconstructPath(
2          Map<Pozycja, Pozycja> parent,
3          Pozycja goal) {
4
5      List<Pozycja> path = new ArrayList<>();
6      Pozycja current = goal;
7
8      while (current != null) {
9          path.add(current);
10         current = parent.get(current);
11     }
12
13     Collections.reverse(path);
14     return path;
15 }
```

Process: Follow parent links from goal to start, then reverse

DFS Implementation

```
1  public static List<Pozycja> znajdzSciezkeDFS(
2          Labirynt lab, Pozycja start, Pozycja goal) {
3
4      Deque<Pozycja> stack = new ArrayDeque<>();
5      Map<Pozycja, Pozycja> parent = new HashMap<>();
6      Set<Pozycja> visited = new HashSet<>();
7
8      stack.push(start);
9      visited.add(start);
10     parent.put(start, null);
11
12     // main loop...
13 }
```

Key difference: Stack instead of queue

DFS Implementation – Main Loop

```
1  while (!stack.isEmpty()) {
2      Pozycja current = stack.pop();
3
4      if (current.equals(goal)) {
5          return reconstructPath(parent, goal);
6      }
7
8      for (Pozycja next : lab.neighbours(current)) {
9          if (!visited.contains(next)) {
10             visited.add(next);
11             parent.put(next, current);
12             stack.push(next);
13         }
14     }
15 }
16
17 return null;
```

Almost identical to BFS, just stack operations

Test Program – MazeSearch

```
1  public class MazeSearch {
2      public static void main(String[] args) {
3          char[][] grid = {
4              "#######".toCharArray(),
5              "#S...G#".toCharArray(),
6              "#.#.#.#".toCharArray(),
7              "#.....#".toCharArray(),
8              "#######".toCharArray()
9          };
10
11         Labirynt lab = new Labirynt(grid);
12         Pozycja start = lab.findStart();
13         Pozycja goal = lab.findGoal();
14
15         // search...
16     }
17 }
```

Test Program – Running Searches

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

List<Pozycja> pathBFS = znajdzSciezkeBFS(
    lab, start, goal);
System.out.println("BFS path: " + pathBFS);
System.out.println("BFS length: "
    + (pathBFS.size() - 1));

List<Pozycja> pathDFS = znajdzSciezkeDFS(
    lab, start, goal);
System.out.println("DFS path: " + pathDFS);
System.out.println("DFS length: "
    + (pathDFS.size() - 1));
```

Compare: Path lengths and exploration patterns

Example Output

Input maze:

```
#######
#S...G#
#.#.#.#
#.....#
#######
```

Output:

- Start: (1,1)
- Goal: (5,1)
- BFS path: [(1,1), (2,1), (3,1), (4,1), (5,1)]
- BFS length: 4
- DFS path: [(1,1), (1,2), (1,3), (2,3), ..., (5,1)]
- DFS length: 8 (example, depends on order)

Heuristic Search Introduction

Problem with BFS/DFS:

- No information about goal location
- Explore uniformly in all directions
- May waste time exploring wrong direction

Solution: Use heuristic function

Heuristic $h(p)$: Estimated distance from position p to goal

- Guide search toward goal
- Explore promising positions first
- Example: Manhattan distance

Manhattan distance:

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

Manhattan Distance Example

Current position: $(2, 3)$
Goal position: $(5, 1)$

Manhattan distance:

$$h = |2 - 5| + |3 - 1| = 3 + 2 = 5$$

Interpretation:

- Minimum number of moves (if no walls)
- Admissible heuristic – never overestimates
- Consistent – satisfies triangle inequality

Why "Manhattan":

- Like walking in Manhattan street grid
- Can only move horizontally or vertically
- Cannot cut diagonally

A* Algorithm Introduction

A* (A-star): Best-first search with cost and heuristic

Evaluation function:

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

where:

- $g(p)$ – actual cost from start to p (distance traveled)
- $h(p)$ – estimated cost from p to goal (heuristic)
- $f(p)$ – estimated total cost through p

Strategy:

- Use priority queue ordered by $f(p)$
- Always expand position with lowest $f(p)$
- Balances exploration and exploitation

A* Properties

If heuristic is admissible and consistent:

- Complete – finds solution if exists
- Optimal – finds shortest path
- Efficient – explores fewer nodes than BFS

Comparison:

- BFS: $f(p) = g(p)$ (ignore heuristic)
- Greedy: $f(p) = h(p)$ (ignore cost)
- A*: $f(p) = g(p) + h(p)$ (best balance)

Trade-off:

- More complex implementation
- Requires good heuristic
- Priority queue overhead
- But: significantly faster for large mazes

Heuristic Search – Manhattan Distance

```
1 private static int manhattan(Pozycja p, Pozycja goal)
      {
2      return Math.abs(p.getX() - goal.getX())
3           + Math.abs(p.getY() - goal.getY());
4 }
```

Usage in A*:

```
1 int g = distance.get(current);
2 int h = manhattan(next, goal);
3 int f = g + h;
```

Priority queue ordering:

- Positions with lower f value expanded first
- Guides search toward goal

A* Implementation Sketch

```java
public static List<Pozycja> znajdzSciezkeHeurystyczna(
        Labirynt lab, Pozycja start, Pozycja goal) {

    PriorityQueue<Pozycja> pq = new PriorityQueue<>(
        (a, b) -> Integer.compare(f.get(a), f.get(b))
    );

    Map<Pozycja, Integer> g = new HashMap<>();
    Map<Pozycja, Integer> f = new HashMap<>();
    Map<Pozycja, Pozycja> parent = new HashMap<>();
    Set<Pozycja> visited = new HashSet<>();

    g.put(start, 0);
    f.put(start, manhattan(start, goal));
    pq.add(start);

    // main loop...
}
```

BFS vs DFS vs A* Summary

Aspect	**BFS**	**DFS**	**A***
Structure	Queue	Stack	PriorityQueue
Optimality	Yes	No	Yes
Efficiency	Medium	Low	High
Memory	High	Medium	High
Heuristic	No	No	Yes

Recommendation:

- Small maze – BFS (simple, optimal)
- Large maze – A* (fast, optimal)
- Memory constrained – DFS
- Learning purposes – implement all three

Applications Beyond Mazes

Robotics:

- Motion planning in 2D/3D space
- Obstacle avoidance
- Path optimization

Games:

- NPC pathfinding
- Strategy game unit movement
- Puzzle solving (Sokoban, sliding puzzles)

Logistics:

- Warehouse robot navigation
- Delivery route planning
- Network packet routing

Extensions and Variants

Bidirectional search:

- Search from both start and goal
- Meet in the middle
- Faster for large spaces

Iterative deepening:

- DFS with increasing depth limit
- Combines DFS memory efficiency with BFS optimality

Jump point search:

- Optimization for grid-based pathfinding
- Skips symmetric paths
- Much faster than A* on grids

Dijkstra's algorithm:

- A* with $h(p) = 0$
- Optimal for weighted graphs

Summary

- **State space search** – fundamental AI technique
- **Maze** – simple but illustrative problem domain
- **BFS** – layer-by-layer, optimal, queue-based
- **DFS** – depth-first, non-optimal, stack-based
- **A*** – heuristic-guided, optimal, priority queue
- **Manhattan distance** – admissible heuristic for grids
- **Parent tracking** – enables path reconstruction
- **Graph view** – maze as vertices and edges
- **Applications** – robotics, games, logistics