University of Sarajevo
Faculty of Electrical Engineering
Department of Data Science and Artificial Intelligence

Laboratory Exercise 1

Artificial Intelligence

Authors: Benjamin Bandić, Nedim Bečić
Professor: Izudin Džafić

Contents

1 Goal of the Exercise

The goal of this laboratory exercise is to introduce students to the fundamental concepts of problem-solving in artificial intelligence. Students will learn how to formally define a problem as a state space and implement two fundamental uninformed (blind) search algorithms: **Breadth-First Search (BFS)** and **Depth-First Search (DFS)**. Through the practical example of solving a maze, students will gain experience in problem representation and the application of search algorithms. Additionally, the exercise introduces the basics of using the **natID** C++ framework to create a graphical visualization of the search process, a key skill for analyzing and understanding the behavior of AI agents.

2 Theoretical Introduction

2.1 Problem-Solving Agents

As covered in the lectures, a **problem-solving agent** is a type of goal-based agent that finds a sequence of actions leading from an initial to a goal state. This process occurs in three steps:

1. **Formulate:** The agent analyzes the problem and defines the states, actions, and goal.
2. **Search:** The agent explores sequences of actions to find a solution that leads to the goal. This is the central focus of this exercise.
3. **Execute:** The agent carries out the sequence of actions found by the search algorithm.

2.2 Formal Problem Definition

To solve a problem algorithmically, we must define it precisely. Every search problem consists of five key components:

- **Initial State:** The state from which the agent begins.
- **Actions:** The set of possible actions the agent can execute in a given state.
- **Transition Model:** Describes the result of performing an action in a state. *State' = Result(State, Action).*
- **Goal Test:** Determines whether a given state is a goal state.
- **Path Cost:** A function that assigns a numerical cost to a path from the initial state to the current state. In this exercise, each action (step) has a cost of 1.

2.3 State Representation: The Maze

In this exercise, the problem we are solving is finding a path through a maze. We can model the maze as a 2D grid.

- **State:** The agent's location, represented as a pair of coordinates (row, column).

- **Initial State:** The coordinates of the maze entrance (e.g., 'S').
- **Actions:** Moving to an adjacent cell (Up, Down, Left, Right), provided that cell is not a wall.
- **Goal Test:** The agent has reached the coordinates of the exit (e.g., 'E').

In C++, a maze can be simply represented as `std::vector<std::string>` or `std::vector<std::vector<char>>`.

2.4 Uninformed (Blind) Search

Uninformed search strategies have no additional information about the problem other than its definition. They do not know if one state is "better" or "closer" to the goal than another.

2.4.1 Breadth-First Search (BFS)

BFS systematically explores the state space level by level. It first visits all states at depth 1, then all states at depth 2, and so on. It uses a **Queue** data structure (First-In, First-Out). BFS is **complete** (it always finds a solution if one exists) and **optimal** (it finds the shortest path in terms of the number of steps).

BFS Pseudocode:

```
function BFS(start, goal):
  queue <- new Queue()
  visited <- new Set()

  queue.enqueue({path: [start], node: start})
  visited.add(start)

  while queue is not empty:
    {path, node} <- queue.dequeue()

    if node is goal:
      return path

    for each neighbor in get_neighbors(node):
      if neighbor is not in visited:
        visited.add(neighbor)
        new_path <- path + [neighbor]
        queue.enqueue({path: new_path, node: neighbor})

  return failure // No path found
```

2.4.2 Depth-First Search (DFS)

DFS explores the state space by going as deep as possible along a single branch before backtracking. It uses a **Stack** data structure (Last-In, First-Out). DFS is **complete** (in finite state spaces) but is **not optimal** (the first solution it finds is not necessarily the shortest).

DFS Pseudocode:

```
function DFS(start, goal):
  stack <- new Stack()
  visited <- new Set()

  stack.push({path: [start], node: start})

  while stack is not empty:
    {path, node} <- stack.pop()

    if node is in visited:
      continue

    visited.add(node)

    if node is goal:
      return path

    for each neighbor in get_neighbors(node):
      if neighbor is not in visited:
        new_path <- path + [neighbor]
        stack.push({path: new_path, node: neighbor})

  return failure // No path found
```

3 Visualization with the `natID` Framework

One of the key course requirements is the visualization of algorithm behavior. Using the `natID` C++ framework, which you installed and tested following the **NatID Setup Guide**, you will create a simple graphical application that displays the maze and the search process.

The goal is to draw the maze as a grid of squares, where each square is colored according to its type (wall, path, start, goal). During the algorithm's execution, you will need to dynamically change the cell colors to visualize which cells the algorithm is visiting and which form the final path.

3.1 The `natID` Drawing Paradigm

The `natID` framework employs an object-oriented approach for all custom graphics. To create an area for drawing, you must define a new class that inherits from `gui::Canvas`.

Within this custom class, you must override the virtual method `onDraw()`. This method is the heart of your visualization; the framework calls it automatically whenever the window needs to be repainted. You should never call `onDraw()` directly. To trigger a repaint after your data changes (e.g., after a search step), you must call the `reDraw()` method.

3.2 Basic Drawing Example: A Tic-Tac-Toe Board

To help you understand the core drawing mechanics before applying them to the maze, let's review a simpler example. The code below shows a minimal 'onDraw()' implementation for a custom canvas that draws a 3x3 Tic-Tac-Toe board. This demonstrates the essential pattern you will need to adapt for your 'MazeCanvas'.

```
/*
  TicTacToeCanvas.h - A simple example of custom drawing.
  This class would inherit from gui::Canvas.
*/
#pragma once
#include <gui/Canvas.h>
#include <gui/Shape.h>

class TicTacToeCanvas : public gui::Canvas {
protected:
    // Override the onDraw method for custom rendering.
    // This method is called automatically by the framework whenever the
    // canvas needs to be repainted (e.g., on window resize or after reDraw()).
    void onDraw(const gui::Rect& rect) override {
        // Calculate the size of each cell. Using getWidth() and getHeight()
        // ensures our drawing adapts if the window is resized.
        const int cellWidth = getWidth() / 3;
        const int cellHeight = getHeight() / 3;

        // A gui::Shape object is a reusable and efficient tool for drawing.
        gui::Shape shape;

        // Loop through each cell of the 3x3 grid
        for (int r = 0; r < 3; ++r) {
            for (int c = 0; c < 3; ++c) {
                // Define the rectangle for the current cell
                gui::Rect cellRect(c * cellWidth, r * cellHeight, cellWidth, cellHeight);
                td::ColorID fillColor = td::ColorID::White; // Default cell color

                // Example: Hardcode colors for two cells to represent 'X' and 'O'
                if (r == 1 && c == 1) {
                    fillColor = td::ColorID::Green; // 'X' at (1,1)
                } else if (r == 0 && c == 2) {
                    fillColor = td::ColorID::Red;   // 'O' at (0,2)
                }

                // Use the Shape object to draw the cell.
                shape.createRect(cellRect);
                // The drawFillAndWire method is a convenient way to draw a filled
                // shape with a border in a single command.
                shape.drawFillAndWire(fillColor, td::ColorID::Black);
            }
        }
```

```
    }
};
```

This simple example provides the fundamental logic you need for Task 4. Your task is to apply this pattern to a much larger grid (the maze) and to determine the color of each cell dynamically based on the data from your ‘Maze‘ object and the current state of your search algorithm (the visited set and the solution path).

4 Tasks for the Laboratory Session

All tasks should be implemented within a single C++ project using the `natID` framework. Each task is a logical unit that builds upon the previous one.

Task 1: Implement the `Maze` Class

Create a C++ class named ‘Maze‘ that will be responsible for loading and storing the maze representation.

- The class should contain a 2D vector (‘`std::vector<std::string>`‘) to store the map.
- Implement a method ‘loadFromFile(const std::string filename)‘ that loads the maze from a text file.
- Add helper methods such as ‘getWidth()‘, ‘getHeight()‘, ‘isWall(int r, int c)‘, ‘getStart()‘, and ‘getEnd()‘.
- Example `maze.txt`:

```
###############
#S#   #       #
# # # # ##### #
#   #   #   # #
##### ### # # E
#     #   #   #
###############
```

Task 2: Implement Breadth-First Search (BFS)

Implement a function that finds the shortest path in the maze using the BFS algorithm.

- The function should have a signature similar to:
  `std::vector<Node> bfs(Maze& maze);`
- `Node` can be a simple struct or a ‘`std::pair<int, int>`‘ that stores the (row, column) coordinates.
- The function should return a vector of nodes representing the path from the start to the goal. If no path exists, return an empty vector.
- Use ‘std::queue‘ for the list of nodes to visit and ‘`std::vector<std::vector<bool>>`‘ or ‘std::set‘ to track visited nodes.

Task 3: Implement Depth-First Search (DFS)

Implement a function that finds a path in the maze using the DFS algorithm.

- The function should have a signature similar to:
  `std::vector<Node> dfs(Maze& maze);`
- Use ‘std::stack‘ for the list of nodes to visit.
- Be aware that DFS does not guarantee finding the shortest path.

Task 4: Create the `natID` Application and Visualization

Integrate the previously implemented components into a `natID` application.

- Create an application window that contains a custom ‘MazeCanvas‘ (which you must implement yourself) to draw the state of the maze.
- Add two buttons: ”Solve with BFS” and ”Solve with DFS”.
- Clicking a button should invoke the corresponding algorithm.
- Visualize the search process. It is recommended to render the search step-by-step (e.g., with a 50ms timer) to create a visible animation.
- In your ‘MazeCanvas::onDraw‘ method, use different colors for: walls, free space, visited cells, and the final path.

Bonus Task

Expand the application to display statistics for each search:

- The length of the found path (number of steps).
- The total number of visited (explored) nodes.
- Compare the results for BFS and DFS on the same maze.