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

Laboratory Exercise 4

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 **informed (heuristic) search** strategies. Building on the foundation of uninformed search from Lab 1, students will now learn how a heuristic function can intelligently guide a search algorithm to find solutions more efficiently. The main objective is to implement the powerful **A* Search** algorithm to solve the classic **8-Puzzle** problem. The primary tasks focus on the problem's underlying logic and the search algorithm itself, with console-based output. The more complex task of creating a graphical visualization with the **natID** framework is included as a bonus, allowing students to first master the core AI concepts.

2 Theoretical Introduction

2.1 The Limits of Uninformed Search

In the first exercise, we saw that BFS and DFS explore the state space systematically without any knowledge of which state might be "better" or closer to the goal. While BFS guarantees the shortest path, it can be very inefficient in large state spaces as it explores all nodes at a given depth. This "blind" exploration is computationally expensive.

2.2 Informed (Heuristic) Search

Informed search algorithms use problem-specific knowledge to find solutions more efficiently. This knowledge is encapsulated in a **heuristic function**, denoted as `h(n)`. A heuristic function estimates the cost of the cheapest path from the current state `n` to a goal state. It's an "educated guess", it doesn't have to be perfect, but a good heuristic leads to a more efficient search.

2.3 The 8-Puzzle Problem

The 8-Puzzle is a classic problem in AI. It consists of a 3x3 grid with 8 numbered tiles and one blank space. The goal is to rearrange the tiles from a given initial configuration into a target configuration by sliding tiles into the blank space.

- **States:** A specific 3x3 configuration of the tiles.
- **Actions:** Moving the blank space Up, Down, Left, or Right.
- **Goal Test:** The current state matches the goal configuration.

2.4 Designing Heuristics for the 8-Puzzle

A good heuristic for the 8-Puzzle should estimate how many moves are needed to reach the goal.

- **Number of Misplaced Tiles:** A simple heuristic. Count every tile that is not in its correct goal position.
- **Manhattan Distance:** A more powerful heuristic. For each tile, sum the horizontal and vertical distance from its current position to its goal position. The total is the sum of these distances for all tiles.

2.5 The Priority Queue Data Structure

Unlike the simple Queue (FIFO) and Stack (LIFO), a **Priority Queue** is a data structure where each element has an associated "priority". The key operations are:

- **push(element):** Adds an element to the queue.
- **pop():** Removes the element with the highest priority.
- **top():** Returns a reference to the element with the highest priority.

In C++, `std::priority_queue` is a **max-priority queue** by default, meaning `top()` returns the largest element. For informed search algorithms, we need a **min-priority queue** that returns the smallest element (i.e., the node with the lowest cost). This requires providing a custom comparator, as shown in the C++ example section.

2.6 Greedy Best-First Search

Greedy Best-First Search is a simple informed algorithm that always expands the node that appears to be closest to the goal. It uses a priority queue ordered by the heuristic value `h(n)`.

- **Flaw:** It is "greedy" because it only considers `h(n)`, ignoring the cost of the path taken so far (`g(n)`). This makes it fast but also **not optimal**; it may find a solution, but not necessarily the shortest one.

Greedy Best-First Search Pseudocode:

```
function GreedyBestFirstSearch(start, goal, heuristic):
  priority_queue <- new MinPriorityQueue() // ordered by h(n)
  visited <- new Set()

  priority_queue.push(start, with_priority=heuristic(start))

  while priority_queue is not empty:
    node <- priority_queue.pop()

    if node is goal: return reconstruct_path(node)

    if node is in visited: continue
    visited.add(node)

    for each neighbor in get_neighbors(node):
      if neighbor is not in visited:
        priority_queue.push(neighbor, with_priority=heuristic(neighbor))

  return failure
```

2.7 A* Search: The Optimal Approach

A* (A-star) is the most widely used informed search algorithm. It improves upon Greedy search by combining the path cost so far with the heuristic estimate. It evaluates nodes using the formula: `f(n) = g(n) + h(n)`

- `g(n)`: The actual cost of the path from the start state to the current state `n`.
- `h(n)`: The estimated cost from state `n` to the goal (the heuristic).
- `f(n)`: The total estimated cost of the best solution that passes through node `n`.

A* uses a priority queue ordered by the `f(n)` value. By balancing the cost already paid (`g(n)`) with the estimated future cost (`h(n)`), A* is both efficient and **optimal**, guaranteeing it will find the shortest path if its heuristic is "admissible" (i.e., it never overestimates the true cost). Both Misplaced Tiles and Manhattan Distance are admissible heuristics.

A* Search Pseudocode:

```
function AStarSearch(start, goal, heuristic):
  priority_queue <- new MinPriorityQueue() // ordered by f(n)
  visited <- new Set()

  // Store g(n) cost for each node, default is infinity
  g_costs[start] = 0

  priority_queue.push(start, with_priority=heuristic(start)) // f(n) =
   0 + h(n)

  while priority_queue is not empty:
    node <- priority_queue.pop()

    if node is goal: return reconstruct_path(node)

    if node is in visited: continue
    visited.add(node)

    for each neighbor in get_neighbors(node):
      tentative_g_cost = g_costs[node] + cost(node, neighbor)
      if tentative_g_cost < g_costs[neighbor]:
        g_costs[neighbor] = tentative_g_cost
        f_cost = tentative_g_cost + heuristic(neighbor)
        priority_queue.push(neighbor, with_priority=f_cost)

  return failure
```

3 Guiding C++ Example: Using a Priority Queue

The key to implementing both Greedy and A* search is a min-priority queue. The C++ `std::priority_queue` requires a custom comparator to achieve this. The following example shows how to do this for a simple 'Task' struct. This is the main C++ pattern you will need to adapt for this lab.

```cpp
#include <iostream>
#include <queue>
#include <vector>
#include <string>

// A simple struct to represent a task with a priority
struct Task {
    int priority; // Lower values mean higher priority
    std::string name;
};

// A custom comparator struct to order tasks.
// It overloads operator() to define the comparison logic.
// This creates a MIN-priority queue by returning true if 'a' should
// come AFTER 'b' (i.e., 'a' has a higher priority value).
struct CompareTask {
    bool operator()(const Task& a, const Task& b) {
        return a.priority > b.priority;
    }
};

int main() {
    // Declare a priority queue that uses our custom Task and
   CompareTask
    std::priority_queue<Task, std::vector<Task>, CompareTask>
   task_queue;

    // Add tasks with different priorities
    task_queue.push({3, "Do laundry"});
    task_queue.push({1, "Implement A* algorithm"});
    task_queue.push({2, "Study theory"});

    std::cout << "Processing tasks based on priority:" << std::endl;
    while (!task_queue.empty()) {
        // .top() gets the element with the highest priority (lowest
   value)
        Task current_task = task_queue.top();
        task_queue.pop();
        std::cout << "Priority: " << current_task.priority
                  << ", Name: " << current_task.name << std::endl;
    }
    return 0;
}
/*
Expected Output:
Processing tasks based on priority:
Priority: 1, Name: Implement A* algorithm
Priority: 2, Name: Study theory
Priority: 3, Name: Do laundry
*/
```

4 Main Tasks for the Laboratory Session

The main goal is to implement the logic of the 8-Puzzle and the A* Search algorithm. All output for these tasks can be directed to the standard console.

Task 1: Create the 8-Puzzle State Representation

Create a C++ class or struct to represent the state of the 8-Puzzle.

- The class should contain a 3x3 array or `std::vector<std::vector<int>>` to store the tile configuration.
- Implement methods to find the position of the blank tile and to generate valid successor states (neighbors).
- To be used in a ‘std::set‘ or as a key in ‘std::map‘, your state representation must be comparable. Overload the `operator<` for this purpose. For hash-based containers, you would need to provide a custom hash function.

Task 2: Implement Heuristic Functions

Create two free functions that take a puzzle state and the goal state, and return the heuristic value.

- int misplacedTiles(const PuzzleState current, const PuzzleState goal);
- int manhattanDistance(const PuzzleState current, const PuzzleState goal);

Task 3: Implement A* Search Algorithm

Implement the A* search function.

- The function should have a signature like:
  `std::vector<PuzzleState> aStarSearch(const PuzzleState start,`
  `const PuzzleState goal, HeuristicFunction h);`
  (where ‘HeuristicFunction‘ could be a function pointer or ‘std::function‘).
- You will need a struct to store nodes in the priority queue, which should contain the state, its `g(n)` cost, and its `f(n)` cost.
- Use a `std::priority_queue` with a custom comparator that orders nodes based on their `f(n)` value.
- Create a ‘main‘ function to define a start and goal state, run the A* search, and print the sequence of board states in the solution path to the console.

5 Bonus Tasks (Visualization)

For the bonus part, integrate your solution from the main tasks into a `natID` application to visualize the solving process.

Bonus Task 1: Implement `PuzzleCanvas`

Create a custom canvas class inheriting from ‘gui::Canvas‘ to draw the 8-Puzzle board.

- The ‘onDraw‘ method should iterate through the 3x3 grid of a ‘PuzzleState‘ object.
- Draw each tile as a colored rectangle with its number centered inside. The blank tile can be a different color (e.g., black or gray).

Bonus Task 2: Animate the Solution

In your main application window, after the search algorithm finds a solution path, use a `gui::Timer` to animate the solution.

- Set the timer to trigger, for example, every 500ms.
- In the timer’s event handler, update your ‘PuzzleCanvas‘ with the next state from the solution path and call ‘reDraw()‘.

Bonus Task 3: Add UI and Statistics

Add controls to your application to enhance usability.

- Add buttons to ”Shuffle” the puzzle and ”Solve”.
- Add a radio button or dropdown to let the user select which heuristic (Misplaced Tiles or Manhattan Distance) to use for the search.
- Display statistics after a solution is found: path length, number of explored nodes, and time taken.