AIR 5021 Final Project Report

Path Planning for Robotic Arm Navigation in Maze

Team Number: 15 **Team Member:** Jinfan He, Ruiyu Zhang, Hao You

1 Introduction

1.1 Research Background

Maze navigation has long been studied under the assumption of a known map; robots are given a pre-built representation of the environment and must plan and follow a collision-free path. In industrial and research settings, this simplifies perception by offloading mapping to an offline process. Our work adopts this paradigm: given a pre-constructed maze map, we focus on translating the planned route into precise motion commands for a robotic arm, ensuring accurate traversal of corridors and turns without on-the-fly mapping.

1.2 Problem Statement and Objectives

This work implements autonomous maze traversal for a robotic arm on a known maze map. Our specific objectives are:

- Given a static, pre-built binarized maze map, extract the set of key turning points (corners) and the designated entry/exit.
- Construct a graph from those points, where edges represent collision-free straight-line segments through maze corridors.
- Apply the A^* algorithm with a Manhattan-distance heuristic to compute the shortest path between entry and exit.
- Convert the resulting sequence of pixel-space waypoints into real-world coordinates and generate executable motion commands for the robotic arm.

2 System Architecture

2.1 Module Overview

The system is decomposed into five primary modules, each responsible for a distinct stage of the maze-traversal pipeline:

1. **Image Processing**
   - *Inputs*: Pre-built binary maze image.
   - *Responsibilities*:
     (a) Read and validate image format (`cv2.imread`).
     (b) Convert to grayscale and threshold to enforce binary representation.
     (c) Apply morphological operations (erosion/dilation) to remove noise and smooth corridors.
   - *Outputs*: Cleaned binary image **B**.

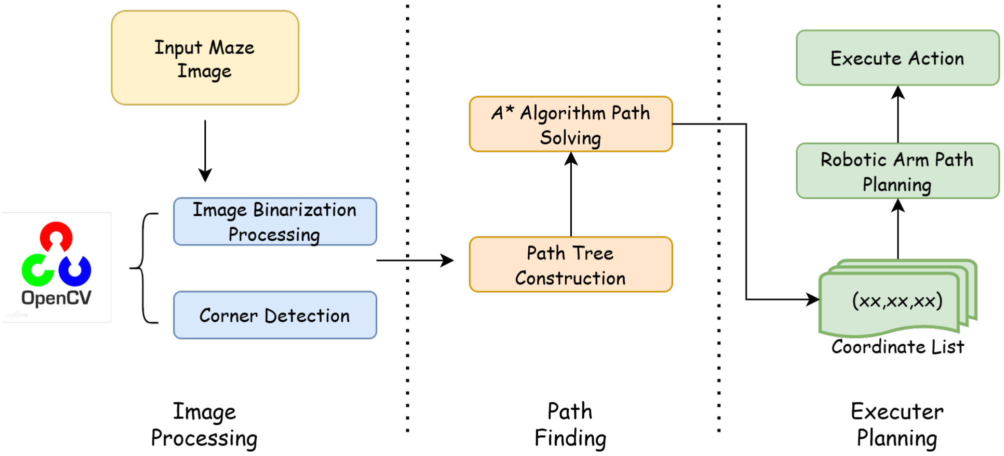

Figure 1: High-level System Workflow

2. **Feature Extraction**
   - *Inputs*: Binary image **B**.
   - *Responsibilities*:
     (a) Detect external contours via `cv2.findContours`.
     (b) Approximate contours to polylines (`cv2.approxPolyDP`) and collect corner candidates.
     (c) Cluster candidates with DBSCAN to yield a reduced corner set C.
     (d) Scan image borders for white-pixel clusters to identify `start` and `goal`.
   - *Outputs*: $C = \{c_1, \dots, c_N\}$, `start`, `goal`.

3. **Graph Construction**
   - *Inputs*: C, `start`, `goal`, **B**.
   - *Responsibilities*:
     (a) Initialize adjacency matrix $\mathbf{G} \in \{0, 1\}^{(N+2)\times(N+2)}$.
     (b) For each unordered pair (u, v), call `is_connected` to test obstacle-free direct line.
     (c) Populate $\mathbf{G}_{uv} = \mathbf{G}_{vu} = 1$ if clear, else 0.
   - *Outputs*: Undirected graph **G**.

4. **Path Planning**
   - *Inputs*: Graph **G**, node list $C \cup \{$`start`, `goal`$\}$.
   - *Responsibilities*:
     (a) Execute A* search with $h(n) = \|n - \texttt{goal}\|_1$.
     (b) Maintain open/closed sets, compute g, $f = g + h$.
     (c) Retrieve optimal sequence of node indices $[s, \dots, g]$.
   - *Outputs*: Pixel-space waypoint sequence.

5. **Motion Control**
   - *Inputs*: Waypoint list $\{w_1, \dots, w_K\}$.
   - *Responsibilities*:
     (a) Convert each w_k from pixel to Cartesian coordinates using calibration.
     (b) Plan smooth joint/Cartesian trajectories via inverse kinematics.
     (c) Publish commands to the robotic controller.
   - *Outputs*: Executable robot trajectory.

2.2 Data Flow and Functional Partition

1. **Input Acquisition** The binary maze image is loaded into memory and passed to the Image Processing module.
2. **Preprocessing → Feature Extraction**
   - The cleaned binary image **B** is forwarded to Feature Extraction.
   - Corner candidates and entry/exit points are detected and clustered, producing C, `start`, `goal`.
3. **Feature Extraction → Graph Construction**
   - C, `start`, `goal`, and **B** are inputs to Graph Construction.
   - Connectivity tests generate adjacency matrix **G**.
4. **Graph Construction → Path Planning**
   - **G** and node list feed into the Path Planning module.
   - A* returns an ordered list of pixel-space waypoints $\{w_1, \ldots, w_K\}$.
5. **Path Planning → Motion Control**
   - Waypoints are mapped to real-world coordinates.
   - Motion Control interpolates trajectories and emits robot commands.
   - The robotic arm executes the trajectory to traverse the maze.

3 Algorithm and Implementation

3.1 Image Preprocessing

Due to variations in ambient lighting (e.g., shadows, glare) and camera perspective distortions, the raw maze images may exhibit uneven contrast and skew. To obtain a standardized representation for reliable downstream processing, we perform binary thresholding on the input image to separate navigable paths (white) from walls (black), yielding a clean binary map **B**.

(a) Original Maze Image

(b) Binary Thresholded Image

Figure 2: Comparison of the raw input and its binary preprocessing result

3.2 Entry and Exit Detection

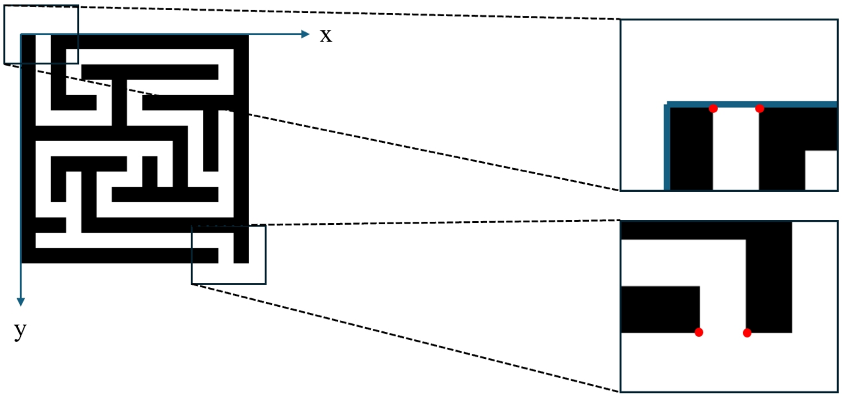

Figure 3: Example of Entrance and Exit Detection: red dots mark the boundary coordinates.

We assume that the binary maze contains exactly one entrance and one exit located on its boundary. To identify their coordinates:

1. **Border Scanning:** For each of the four image edges (top, bottom, left, right), collect all white pixels (255).
2. **Cluster Averaging:** For each edge with detected white pixels, compute the mean (x, y) coordinate of that cluster.
3. **Entrance/Exit Assignment:** The two resulting mean points are taken as the entrance and exit (order interchangeable).

3.3 Corner Detection and Clustering

To reliably identify the maze's geometric "corners" (points where the path direction changes), we apply a three-stage pipeline on the binary map $\mathbf{B}$: morphological erosion, Douglas–Peucker contour simplification, and DBSCAN clustering. Below is a detailed description of each stage, including the mathematical operations and their objectives.

Morphological Erosion We begin by thinning all walkable corridors to a one-pixel skeleton and removing small noise. Given a binary image $\mathbf{B} \in \{0, 1\}^{H \times W}$ and a square structuring element

$$K = \mathbf{1}_{k \times k}, \quad k = \left\lceil \tfrac{2}{3} w \right\rceil,$$

where w is the average corridor width in pixels, the eroded image $\mathbf{E}$ is computed as

$$\mathbf{E}(x, y) = \big(\mathbf{B} \ominus K\big)(x, y) = \min_{(i,j) \in K} \mathbf{B}(x + i, y + j).$$

Each foreground pixel survives only if every position of K fits within the original foreground. This operation removes any white regions narrower than k and peels one pixel layer from all corridor boundaries.Figure 4 shows the result after morphological erosion.

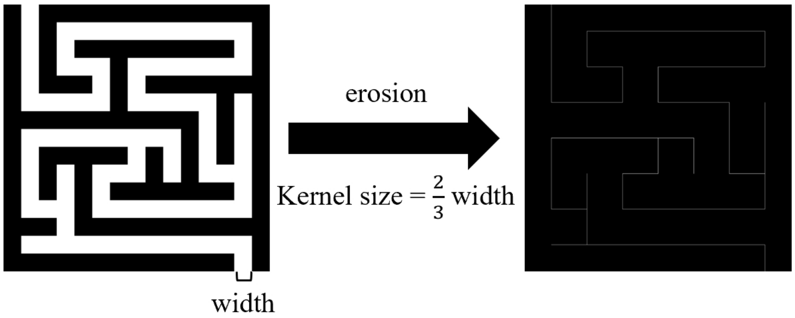

Figure 4: Illustration of morphological erosion: original binary maze map (left) vs. eroded skeleton (right), with kernel size two-thirds of the corridor width.

This operation reduces each corridor to a single-pixel skeleton and removes any white regions narrower than the structuring element, providing a clean basis for contour extraction.

Douglas–Peucker Simplification Next, we extract the external contour $C = \{\mathbf{p}_1, \ldots, \mathbf{p}_M\}$ of $\mathbf{E}$. To reduce M while preserving sharp turns, we apply the Douglas–Peucker algorithm with tolerance ϵ. For any segment between endpoints $\mathbf{p}_a$ and $\mathbf{p}_b$, we compute the perpendicular distance of each intermediate point $\mathbf{p}_i$ to the line $\overline{\mathbf{p}_a\mathbf{p}_b}$:

$$d_i = \frac{\left|(\mathbf{p}_b - \mathbf{p}_a) \times (\mathbf{p}_a - \mathbf{p}_i)\right|}{\|\mathbf{p}_b - \mathbf{p}_a\|}.$$

If $\max_i d_i > \epsilon$, the point $\mathbf{p}_k$ achieving this maximum is marked "essential," and the algorithm recurses on the sub-contours $\{\mathbf{p}_a, \ldots, \mathbf{p}_k\}$ and $\{\mathbf{p}_k, \ldots, \mathbf{p}_b\}$. Otherwise, all intermediate points are discarded. With $\epsilon \approx 10$px, this produces a reduced polyline

$$P = \{\mathbf{q}_1, \ldots, \mathbf{q}_N\}$$

comprising only those vertices where the contour bends by more than ϵ.

DBSCAN Clustering Finally, to merge nearly coincident corner candidates and remove outliers, we cluster P using DBSCAN with parameters ε and *MinPts*. Two points $\mathbf{q}_i, \mathbf{q}_j$ are direct-neighbors if

$$\|\mathbf{q}_i - \mathbf{q}_j\| \le \varepsilon, \quad \varepsilon \approx 20\,\text{px}.$$

Any core point has at least *MinPts* neighbors (we set $MinPts = 1$ to include singletons). Clusters C_l are formed by density-reachability, and each cluster is then represented by its centroid

$$c_l = \frac{1}{|C_l|} \sum_{\mathbf{q}_i \in C_l} \mathbf{q}_i.$$

Noise points not belonging to any cluster are discarded. The final corner set

$$C = \{c_1, c_2, \ldots, c_L\}$$

is compact, free of redundancies, and accurately located at the maze's path-direction changes.
The set C then serves as the node list for adjacency-matrix construction and subsequent A* path planning.

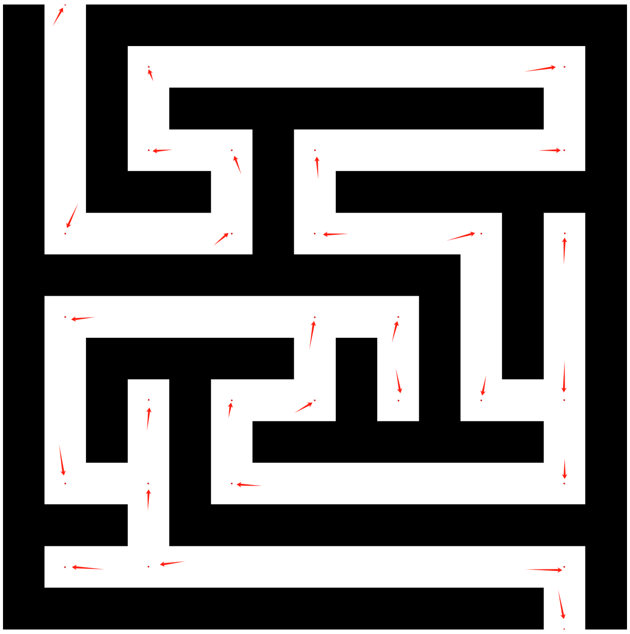

Figure 5: Example of DBSCAN Clustering: Red dots mark corner points(indicated by arrows).

3.4 Path Graph Construction

Let $C = \{c_0, c_1, \ldots, c_N, c_{N+1}\}$ be the corner set including the entrance c_0 and exit c_{N+1}, and let $\mathbf{B} \in \{0,1\}^{H\times W}$ be the binary maze map. We construct an undirected adjacency matrix $\mathbf{G} \in \{0,1\}^{(N+2)\times(N+2)}$ as follows:
For each unordered pair (c_i, c_j), we sample all integer-pixel points along the straight line segment between them. If every sampled point lies in free space ($\mathbf{B} = 1$), we set

$$\mathbf{G}_{ij} = \mathbf{G}_{ji} = 1;$$

if any sampled point is a wall (black pixel, $\mathbf{B} = 0$), we set

$$\mathbf{G}_{ij} = \mathbf{G}_{ji} = 0.$$

Diagonal entries $\mathbf{G}_{ii}$ remain zero. The resulting symmetric matrix $\mathbf{G}$ encodes all straight-line, collision-free connections between maze corners and serves as the graph input for A* path planning.

3.5 A* Path Planning

Algorithm Introduction The A* (A-Star) algorithm is a best-first search method that finds the shortest path in a graph by combining the actual cost from the start node with a heuristic estimate to the goal. At each step, it selects the node n with minimal

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

where

- $g(n)$ is the cost from the start node to n,
- $h(n)$ is the heuristic estimate of the cost from n to the goal.

In our maze graph $\mathbf{G}$, each edge has unit cost, so

$$g(n) = \text{number of edges from start to } n.$$

We use the Manhattan distance between corner coordinates as the heuristic:

$$h(n) = \|c_n - c_g\|_1.$$

Notation

- s, g: start and goal node indices.
- $g(i)$: cost from s to node i.
- $h(i)$: heuristic cost from i to g, here $h(i) = \|c_i - c_g\|_1$.
- O: open set of (f, g, n, π) tuples, where $f = g + h$, n is node index, and π is path.
- $\mathbf{G}_{nm}$: 1 if nodes n and m are connected, else 0.

Algorithm Steps

1. Initialization: $g(s) = 0$, $O = \{(0, 0, s, [s])\}$.
2. While O not empty:
   (a) Pop $(f, g_{\text{score}}, n, \pi)$ with smallest f.
   (b) If $n = g$, return $\{c_i \mid i \in \pi\}$.
   (c) For each m with $\mathbf{G}_{n,m} = 1$: let $\text{tg} = g_{\text{score}} + 1$. If $\text{tg} < g(m)$ or m unseen, set $g(m) = \text{tg}$, $f = \text{tg} + h(m)$, and push $(f, \text{tg}, m, \pi + [m])$.
3. If exhausted, return None.

The algorithm returns the optimal sequence of corner coordinates $\{c_s, c_{i_1}, \ldots, c_{\text{goal}}\}$, which is then overlaid on the maze map as the planned trajectory.

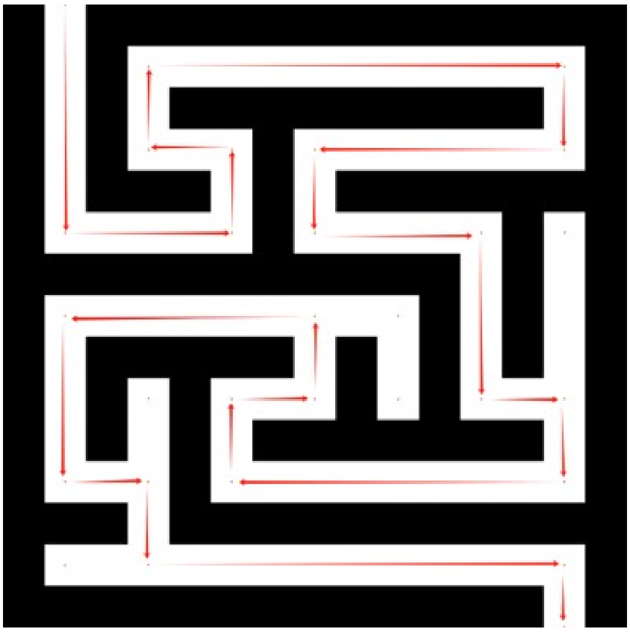

Figure 6: Planned path (red) following corners $\{c_s, c_{i_1}, \ldots, c_{\text{goal}}\}$ over the binary maze.

4 Robotic Arm Control

4.1 Coordinate Transformation

The mapping from pixel coordinates (u, v) to arm-frame coordinates (X, Y) is modeled as a 2D affine transform. During calibration, we move the end effector to the four corners of the maze image—recording pixel targets $\{(u_i, v_i)\}_{i=1}^4$ and corresponding arm positions $\{(X_i, Y_i)\}_{i=1}^4$. We then solve the linear system

$$\begin{pmatrix} X_i \\ Y_i \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} u_i \\ v_i \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \end{pmatrix}, \quad i = 1, \ldots, 4,$$

to determine the six affine parameters a, b, c, d, t_x, t_y. With the transform matrix $\mathbf{A} = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ and translation $\mathbf{t} = (t_x, t_y)^\mathsf{T}$ established, any planner waypoint (u, v) is mapped by

$$\begin{pmatrix} X \\ Y \end{pmatrix} = \mathbf{A} \begin{pmatrix} u \\ v \end{pmatrix} + \mathbf{t}.$$

4.2 Motion Execution

Once all (X, Y) waypoints are stored in `points.txt`, the runtime handoff is managed by two ROS nodes:

1. **High-Level Task Node (`txt_waypoint_executor`)**

- On launch, reads the ROS parameters (`txt_path`, `pick_pose`, `arm_ns`) and opens `points.txt`.
- Sends a single "pick" goal to the `/sgr_ctrl` Action server to close the gripper at the configured pick pose.
- Iterates through each waypoint, wrapping (X, Y) and a fixed orientation into an `SGRCtrlGoal` and calling `send_goal()`.
- Waits for each goal's result and monitors feedback to detect any anomalies (e.g. planning failure).
- After the final waypoint, issues a "release" goal, then commands a return to the original pick pose before shutting down.

2. **Low-Level Execution Node (`sgr_ctrl`)**

- Implements the `SGRCtrlAction` server under the `/sgr_ctrl` namespace.
- Upon receiving a goal, determines whether it is a Pick, Put, or Move action based on the `is_grasp` flag.
- Uses MoveIt! to plan a collision-free Cartesian trajectory to an approach pose, applies a fixed offset (e.g. 7 cm forward) for engagement, then performs gripper I/O and retreat.
- Streams real-time feedback on trajectory progress and gripper status back to the client.
- Handles error conditions (e.g. unreachable targets or grasp failures) with preconfigured retry or abort logic.

By issuing the given (X, Y) coordinates, the arm traverses the maze while maintaining a stable end-effector orientation.

Note: The `sgr_ctrl` server was adapted from the Lab 2 codebase, reusing its Action server structure and MoveIt! wrappers for consistency across assignments.

5 Result and Conclusion

To illustrate the arm's traversal through the maze, we capture four key moments in a 2×2 grid of static snapshots. Figure 7 highlights the sequence from the process of the robotic arm moving in the maze.

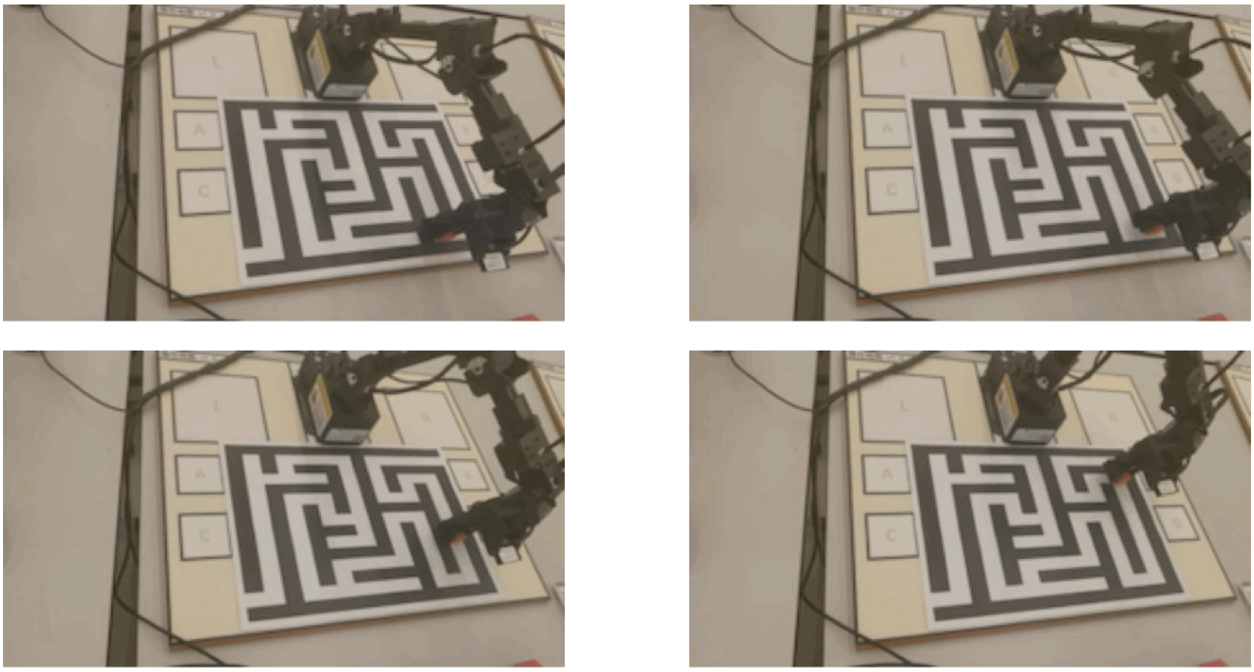

Figure 7: Static snapshots showing moving stages of the robotic arm navigating the maze.

These snapshots confirm that the planned (X, Y) waypoints are executed, the end-effector maintains a stable orientation, and the arm successfully follows the computed path through the maze.
In this project, we demonstrated a complete pipeline for robotic arm maze navigation, from image preprocessing and corner extraction to A* path planning and real-world motion execution. By leveraging a pre-built binary map, we reduced perception complexity and focused on robust graph-based planning. Coordinate transformation via a calibrated 2D affine model enabled accurate mapping from pixel to arm frame, and our ROS-based architecture cleanly separated high-level task sequencing from low-level motion control. Experimental results show reliable execution of complex trajectories within the maze, with consistent gripper performance and collision-free motion. Future work may explore dynamic obstacle handling, real-time map updates, and extension to more degrees of freedom or different end-effectors to broaden applicability.

Appendix

The source code for this project is available at the following GitHub repository:
https://github.com/RM15-15/AIR-5021-Team15-FinalProject