CS 7075

ARTIFICIAL INTELLIGENCE & ROBOTICS

PROJECT REPORT

INSTRUCTOR

Dr. Waqas Majeed

Rohith Sundar Jonnalagadda

KSU ID: 001093082

Sai Veera Venkata Akshay Adabala

KSU ID: 001140226

Ramani Potlapati

KSU ID: 001125320

Comparative Analysis of Maze Path Planning Using Enhanced Sparse A* Algorithm with Obstacle Variations

SUMMARY

In our project, we aimed to enhance path planning in environments with obstacles of varying shapes and sizes by employing a combination of map segmentation and visibility graph-based approaches. Inspired by [1], who proposed decomposing map boundaries and obstacles into convex polygons for the application of a sparse A* algorithm, we adapted and expanded upon their methodology to improve efficiency and path quality. Our approach began with processing a map image to detect the map boundary and any internal obstacles. We used edge detection techniques, specifically the Canny edge detector, to identify significant edges within the image. Contour detection was then applied to extract the map boundary and obstacle contours. The largest external contour was identified as the map boundary, while internal contours were considered obstacles. To simplify the contours and reduce computational complexity, we approximated them using the Douglas-Peucker algorithm. This algorithm reduces the number of vertices in the contours while preserving their essential shape. We ensured the vertices of the polygons were sorted in a consistent clockwise order, which is crucial for accurate geometric representation and subsequent processing. We then classified each vertex of the polygons as convex or concave by analyzing the cross products of adjacent edges. This classification is important for understanding the geometric properties of the map and can influence pathfinding decisions.

To prepare the environment for efficient path planning, we decomposed the map boundary and obstacles into convex sub-polygons using constrained Delaunay triangulation. This process involved creating a triangulation that respects the original geometry of the map, including holes representing obstacles. By decomposing the environment into triangles, we created a simplified representation that is more suitable for computational analysis. With the environment represented as a set of convex sub-polygons, we constructed a visibility graph. This graph models the navigable space by connecting vertices that have a clear line of sight, avoiding intersections with obstacles. The visibility graph serves as the foundation for the A* pathfinding algorithm, allowing us to compute the shortest path between a user-defined start and goal point efficiently. We implemented the A* algorithm on the visibility graph to find the optimal path. The algorithm utilizes a heuristic based on the Euclidean distance to estimate the cost from the current node to the goal, enabling it to prioritize paths that are more likely to lead to the goal quickly. This approach significantly reduces computational complexity compared to traditional grid-based methods, as it limits the search space to relevant points in the environment. To further enhance the path, we applied Bezier curve smoothing. This process reduces sharp turns and creates a more natural and efficient route, which is particularly beneficial for applications like robotics and autonomous navigation, where smooth paths are preferable. We ensured that the smoothed path did not intersect any obstacles by performing collision detection using geometric operations with the obstacle polygons.

Our preprocessing-based method offers significant advantages, including reduced computational complexity and improved global optimality over traditional grid-based A* algorithms. By segmenting the map into convex sub-polygons and constructing a visibility graph, we limited the number of nodes and edges the algorithm needs to consider, leading to faster computations and more optimal paths. Our approach aligns with the findings of [1], which highlighted the efficiency of convex decomposition and node-based regionalization in improving the A* algorithm's performance regarding speed, path quality, and computational stability. By integrating image

processing techniques, computational geometry, graph theory, and optimization algorithms, our project contributes to the development of efficient path planning methods in complex environments.

Our Contributions:

- Map Segmentation: We utilized image processing to detect and simplify the map boundary and obstacles, leading to a more efficient representation of the environment.
- Convex Decomposition: By decomposing the environment into convex sub-polygons (traingles), we reduced computational complexity and facilitated the construction of the visibility graph.
- Visibility Graph Construction: Our method accurately models navigable spaces by accounting for line-of-sight connections between key points, ensuring that only feasible paths are considered.
- Efficient Pathfinding with A* Algorithm: Implementing the A* algorithm on the visibility graph allowed us to find optimal paths more efficiently than traditional grid-based approaches.
- Path Smoothing with Bezier Curves: We enhanced path quality by smoothing the initial path, resulting in smoother trajectories suitable for real-world navigation applications.
- User Interaction: We also provided an interactive interface for users to select start and goal points, enhancing the usability and practicality of our method.

By combining these techniques, our project demonstrates a comprehensive approach to path planning in complex environments with irregular obstacles. The integration of these methods results in a system that is both efficient and effective, capable of producing high-quality paths suitable for various applications, such as robotics, autonomous vehicles, and spatial analysis.

ROBOTICS PROBLEM DEFINITION

The primary problem involves planning a collision-free, optimal path for a mobile robot navigating through mazes and maps filled with obstacles of various shapes and sizes. The robot must efficiently find a path from an initial position to a target position, avoiding these obstacles and passing through narrow pathways, all while adhering to constraints such as limited computational resources and real-time requirements. This problem becomes particularly challenging when the environment is complex, with densely packed or irregularly shaped obstacles, which require the robot to make precise and efficient movements. Efficient and reliable path planning in such environments is a critical issue in robotics, especially for applications like autonomous navigation, industrial automation, and unmanned aerial vehicles (UAVs). In these scenarios, the ability to navigate through dynamic or complex environments while minimizing path cost (e.g., time, distance, or energy) is crucial for both safety and operational efficiency. Furthermore, ensuring that the pathfinding algorithm is scalable to larger, more complex environments is essential for real-world deployment in sectors like logistics, surveillance, and search and rescue operations.

AI AND ROBOTICS PROBLEMS ADDRESSED

Path Planning: Our project focuses on path planning for mobile robots, particularly optimizing the process of finding obstacle-free routes in complex environments. The Sparse A* algorithm is used to improve pathfinding efficiency and path optimality.

AI METHODS

- Traditional A* Algorithm: Utilizes grid-based search with heuristics to find an optimal path but is computationally intensive in large environments.
- RRT (Rapidly-exploring Random Tree): Incrementally explores the configuration space using random sampling, suitable for high-dimensional spaces but may produce irregular paths.
- RRT* (Rapidly-exploring Random Tree*): Optimizes paths through a rewiring process, providing probabilistic completeness and asymptotic optimality.
- PRM (Probabilistic Roadmap): Builds a reusable graph of collision-free paths for efficient query-based navigation.
- Our Improved A* Algorithm: Enhances pathfinding efficiency and quality by combining map segmentation, visibility graphs, and path smoothing.

MAPS

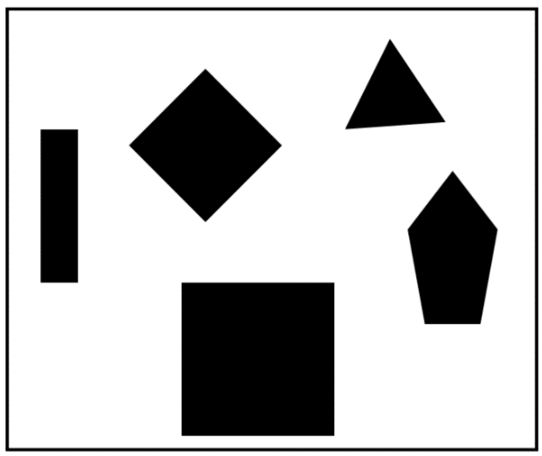


Figure 1: Map 1

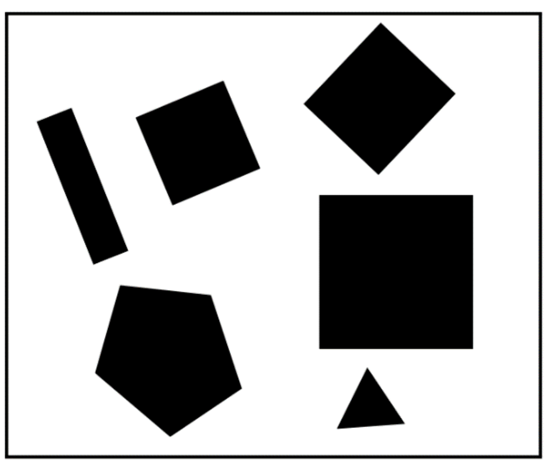


Figure 2: Map 2

METHODOLOGY

Convex Decomposition of the Map

1. Image Preprocessing

The input image is read from the specified file path where the color image is converted to grayscale, it is because grayscale images simplify the data and are sufficient for edge detection. For edge detection we used canny which detects edges by looking for areas of rapid intensity change, using gradient values. The thresholds (50 and 150) define the sensitivity. Contours are extracted from that where the full hierarchy of nested contours are calculated. Then contours are approximated where they it compresses horizontal, vertical, and diagonal segments, keeping only their end points.

2. Map Boundary Detection

The assumption is that the map boundary is the largest external contour. So, we have done hierarchy analysis where the hierarchy information is used to identify top-level contours (contours with no parent, indicated by a parent index of '-1'). For each top-level contour, the area is calculated and the contour with the maximum area among these is selected as the map boundary. Then the Douglas-Peucker algorithm is implemented to simplify the contour by reducing the number of vertices while retaining the overall shape which gives us a simplified contour with fewer points, that helps in reducing computational complexity in subsequent steps. Then we used a sorting algorithm to sort the vertices based on calculated angle to the centroid for each vertex, this ensures consistency in the representation of the polygon, which is crucial for accurate triangulation and visualization.

3. Obstacle Detection

The hierarchy data from previous process provides information about the nesting of contours that allows identification of obstacles within the map boundary by analyzing which contours are descendants of the map boundary contour. Then to identify obstacles a loop is applied through all contours where for each contour the descendant of the map boundary is checked by traversing up the hierarchy from the current contour to see if the map boundary is an ancestor. Then area filtering is applied which ignores contours with areas too close to the map boundary area (to avoid misidentifying the boundary as an obstacle) and duplicates by checking if the area is like previously identified obstacles. Then the obstacle contours are simplified using the same approximation method where the vertices of the obstacles are then sorted in clockwise order to get the correct orientation of the polygon for us to identify if the polygon is convex or concave. Then the duplicate vertices are removed by comparing distances between consecutive vertices and removing those within a certain threshold distance.

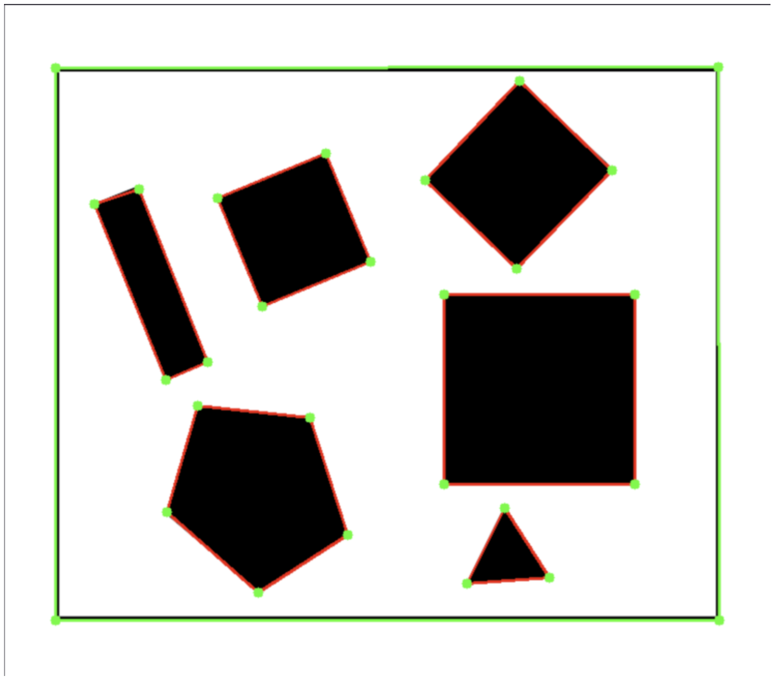


Figure 3: Vertices of map boundary and obstacles detected in Map 1

4. Vertex Convexity Classification

Then the determination of each vertex in the polygon if it is convex or concave is done by calculating the cross product of adjacent vertices.

5. Orientation Correction

Here we ensure the polygon's vertices are ordered correctly (clockwise or counterclockwise) for the triangulation algorithm using Shoelace Formula for calculating the signed area and determining polygon orientation. If the orientation does not meet the requirement (e.g., if they are convex), the order of vertices are reversed to make them concave.

6. Constrained Triangulation

To preparing data for triangulation we firstly assign a unique index to each vertex and store them in a list, then created segments (edges) by pairing indices of consecutive vertices, then organized the vertices and segments into a dictionary 'A' suitable for the 'triangle' library. Obstacles are treated as holes in the triangulation where for each hole, compute a point inside the obstacle and add these points to the 'holes' key in the dictionary 'A'. Then we used the 'triangle' library (a Python wrapper for the Triangle software) to perform constrained Delaunay triangulation where a dictionary 'B' containing the triangulated vertices and the indices forming each triangle are obtained. To extract sub-polygons, we retrieved the corresponding vertices for each set of triangle indices in 'B['triangles']' and stored these vertices as individual sub-polygons (triangles) in result we got a list of convex sub-polygons representing the decomposed map. This is done for decomposing the polygon into convex sub-polygons while respecting the original geometry.

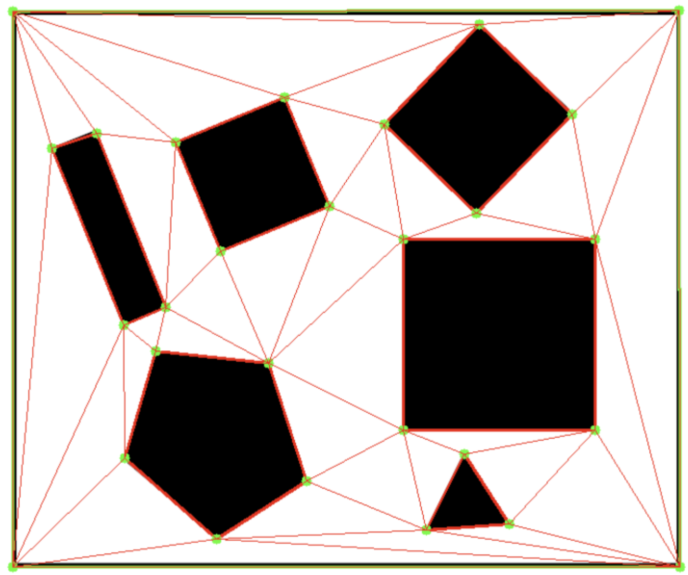


Figure 4: Convex Decomposition of Map 1

7. Calculating and Visualizing Centroids and Edge Midpoints

For each sub-polygon(triangle), we computed the centroid using the moments obtained to draw the centroids as blue circles and the edge midpoints as green circles on the image.

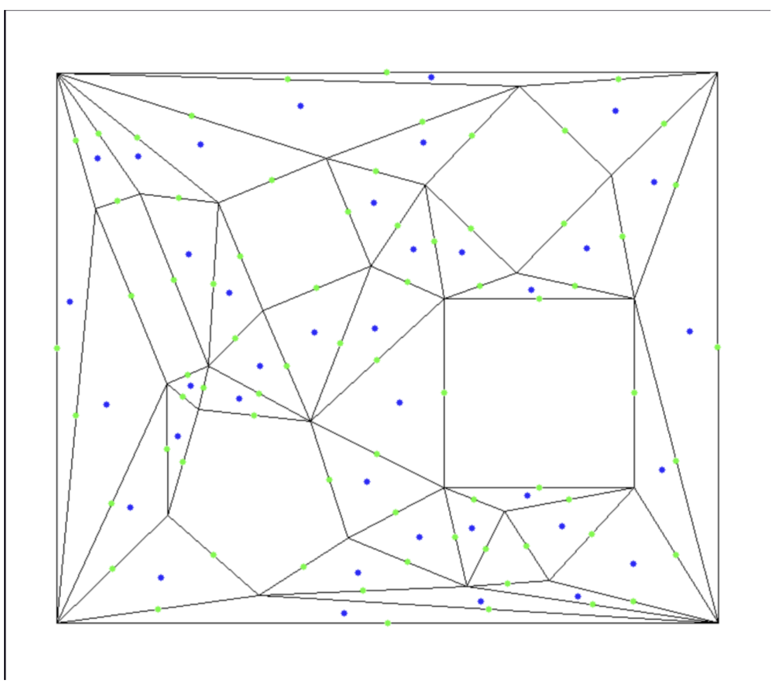


Figure 5: Centroids and Edge Midpoints of sub-polygons

Optimized A* Algorithm

Building upon the initial processing of the map image to detect boundaries and obstacles and performing constrained triangulation to decompose the area into convex sub-polygons, we further extended the methodology to include path planning. This extension involves:

- Reading the Decomposed Sub-polygons
- Constructing a Visibility Graph
- Implementing the A* Pathfinding Algorithm
- Path Smoothing using Bezier Curves
- User Interaction for Start and Goal Selection
- Visualization and Analysis of the Path

1. Reading Decomposed Sub-polygons

To import the convex sub-polygons (triangles) generated from the earlier constrained triangulation step. These sub-polygons represent the navigable space and obstacles within the map. From the file 'sub_polygons.txt' containing the list of sub-polygons from previous process we parsed each line to extract the coordinates of the vertices. Parsing is done by splitting the string representations to retrieve numerical coordinate pairs to store the sub-polygons as lists of vertex tuples. The sub-polygons are used to reconstruct the obstacle geometry in the map. They are converted into Shapely 'Polygon' objects for geometric operations.

2. Constructing the Visibility Graph

We created a graph where nodes represent points (vertices) in the environment, and edges represent direct lines of sight between these points that are not obstructed by obstacles. This graph serves as the foundation for the A* pathfinding algorithm. Firstly, we extracted all unique vertices from the sub-polygons (obstacles) and added the start and end points provided by the user. Then we created a dictionary where each key is a point, and the value is a list of adjacent points (neighbors) with corresponding edge costs. Then we created a unified geometric representation of all obstacles using 'unary_union' from Shapely where we generated the edges of sub-polygons and checked if this line intersects any obstacle. If the line does not cross any obstacles, we calculated the distance (edge weight) using the Euclidean distance heuristic and added each point to the other's adjacency list in the graph. The graph is undirected, and edges are bidirectional so to avoid redundant calculations, only unique pairs are considered and ensured that the edges in the visibility graph represent unobstructed paths.

3. Implementing the A* Pathfinding Algorithm

To implement the A* pathfinding algorithm on the visibility graph, a custom 'Node' class is used to represent each point in the graph with attributes such as coordinates, costs ('g', 'h', 'f'), and a reference to the parent node for path reconstruction. The nodes are ordered in a priority queue based on their total cost ('f'). The algorithm calculates the heuristic cost ('h') as the Euclidean distance between two points. During execution, the algorithm initializes the open list (a priority queue) with the start node and iteratively processes the node with the lowest 'f' cost. For each node, it updates the costs and parent references of its neighbors if a lower-cost path is found, adding them to the open list. Nodes are moved to a closed set once processed. The search ends when the goal node is reached, reconstructing the path by tracing back through parent nodes. If no path is found, the algorithm returns 'None'. This approach ensures efficient and optimal pathfinding while avoiding obstacles. The path is reconstructed in reverse order by following the 'parent' references from the goal node back to the start node. The path is then reversed to present it from start to goal.

4. Path Analysis

To calculate the path length and variance, we computed the Euclidean distance between each pair of consecutive points in the path. These distances are summed to determine the total path length. The variance of the distances is then calculated to assess the uniformity of the path segments, with a lower variance indicating more consistent segment lengths. This analysis is useful for evaluating the smoothness and uniformity of the generated path, which may be critical for certain applications.

5. Visualization of the Initial Path

Then we overlayed the computed path onto the original image for visualization. It drew lines between consecutive points in the path and highlights the start and end points with distinct colored circles. The resulting image is displayed in a window with an appropriate title, such as "Initial Path," and remains visible until a key press closes the window. This process helps in visually assessing the accuracy and feasibility of the generated path.

6. Path Smoothing using Bezier Curves

Then we generated a smoother path for real-world navigation while ensuring obstacle avoidance. It converts path points into NumPy arrays, parameterizes the path with a 't' value from 0 to 1, and fits a Bezier curve using polynomial fitting, typically with a degree of at least 3. A finer parameter 't_smooth' is used to create high-resolution smoothed 'x' and 'y' coordinates. The smoothed path is checked for collisions by iterating through its segments and ensuring no path crosses any obstacle. If collisions are detected, the smoothing is halted, and the original path is returned. This process balances smoothness with feasibility, considering the degree of the polynomial and collision avoidance to maintain realistic and usable paths.

7. User Interaction for Start and Goal Selection

This process allows users to specify start and goal points either manually or interactively, improving usability. Users can choose between manual coordinate input or interactive selection. For manual input, the program prompts for integer coordinates and validates them. For interactive selection, the image is displayed`, and a mouse callback function captures clicks. On a left mouse button click, the point is recorded, and a circle is drawn at the location for visual feedback. Once two points are selected, the pathfinding proceeds. Error handling ensures invalid inputs or choices are addressed by notifying the user and either exiting or prompting again, ensuring a seamless experience.

8. Integration with Previous Steps

The sub-polygons derived from the initial processing are utilized as obstacles for pathfinding, ensuring accurate navigation constraints. These obstacles are represented as Shapely 'Polygon' objects, enabling efficient geometric operations such as collision detection. For visualization, the paths are overlaid on the original map image, offering clear context and a comprehensive view of the navigation environment. This approach integrates obstacle representation and path visualization seamlessly to enhance the pathfinding process.

Workflow

1. Firstly, the map image is converted to grayscale, then detect edges, and extract contours.
2. Then we identified the map boundary as the largest contour and detect obstacles using hierarchical contour information.
3. Then the contours are simplified and sorted vertices for consistent polygon representation.
4. Then we determined convex or concave nature of vertices.
5. Then we ensured whether the polygons have the correct vertex order for triangulation.
6. Then the map is decomposed into convex sub-polygons using Delaunay triangulation, treating obstacles as holes.
7. Then centroids and edge midpoints are calculated and displayed.
8. Then a visibility graph is constructed and implemented the A* algorithm for optimal pathfinding.
9. Then Bezier curves are applied to smooth the path while ensuring collision avoidance.
10. Allowed users to select start and goal points manually or interactively.
11. Overlayed initial and smoothed paths on the map for analysis.

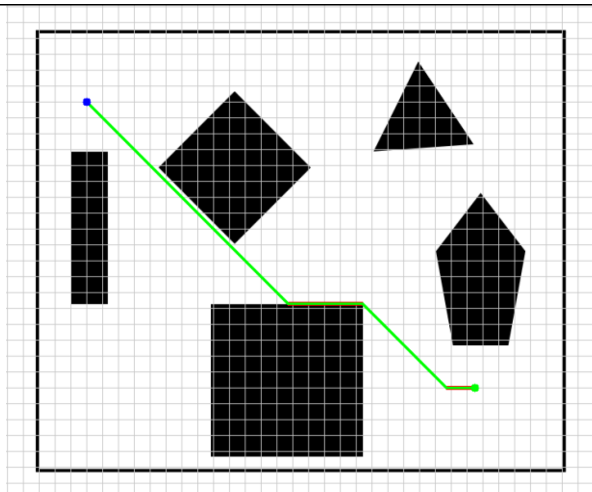
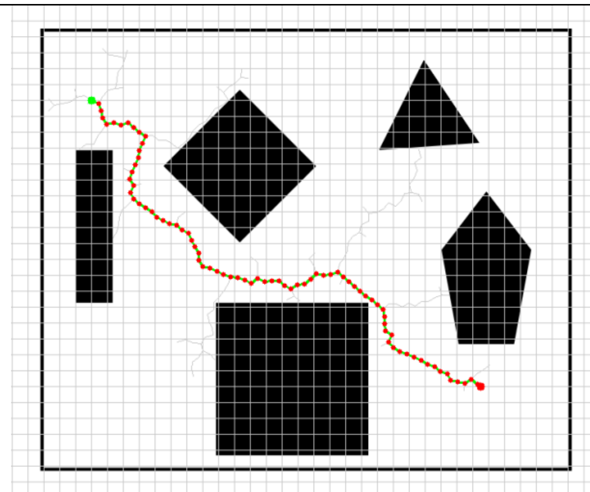
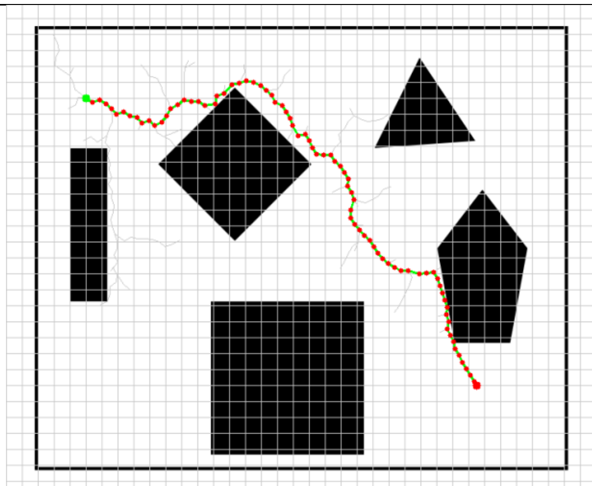
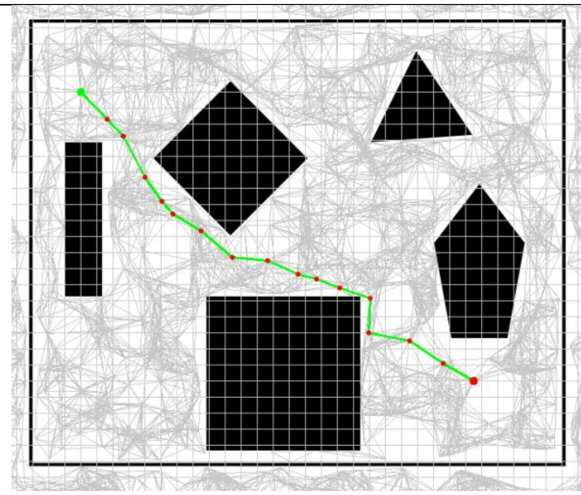
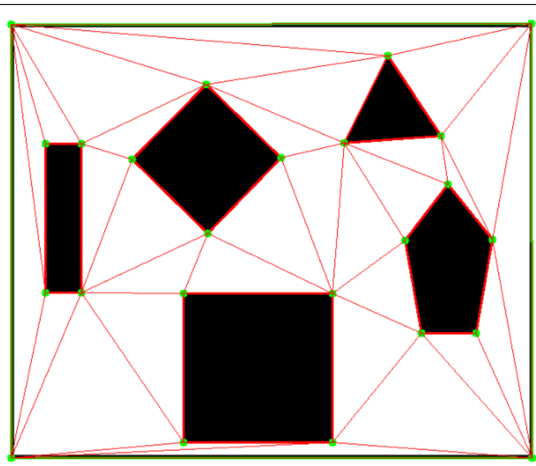
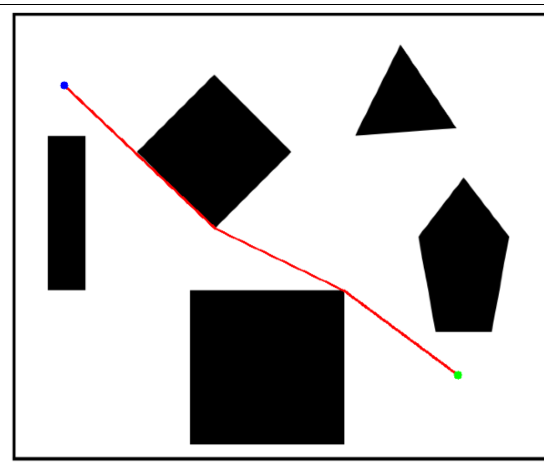
EXPERIMENTATION AND RESULTS

Performance Metrics:

- Path Length: The total distance traveled from the start to the goal point.
- Variance: The consistency of segment lengths in the path; lower variance indicates more uniform segment lengths.
- Time Taken: The computational time required to complete the pathfinding process.
- Nodes Expanded: The total number of nodes explored during the pathfinding, reflecting the algorithm's efficiency in searching the solution space.

We conducted comprehensive experiments on all the maps by first applying convex decomposition to simplify and accurately represent the environment. Utilizing our improved A* algorithm, we then found optimal paths within these decomposed maps. To evaluate the performance and effectiveness of our approach, we tested it against other AI path planning methods, including the traditional grid-based A*, RRT, RRT*, and PRM algorithms. These comparisons allowed us to assess computational efficiency, path optimality, and solution stability. We took start point as (120, 140) and goal point as (610, 500).

For Map 1

 *Figure 6: A**	 *Figure 7: RRT*
 *Figure 8: RRT**	 *Figure 9: PRM*
 *Figure 10: Convex Decomposition*	 *Figure 11: Our A**

For Map 2

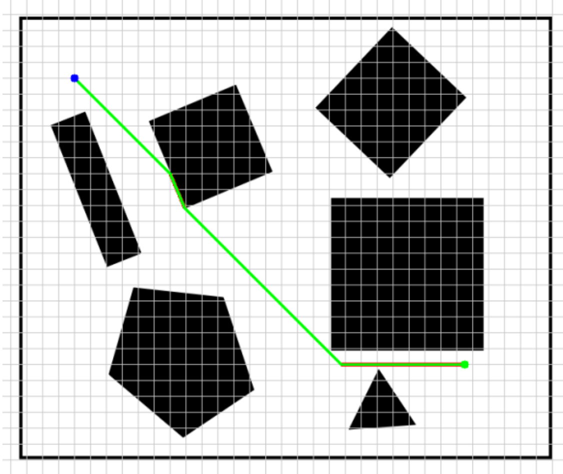


*Figure 13: A**

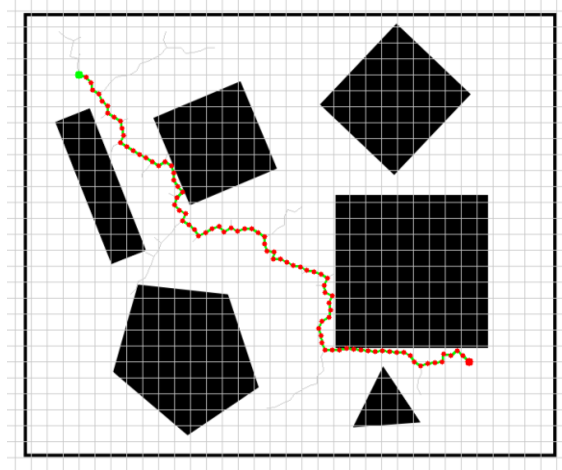


Figure 14: RRT

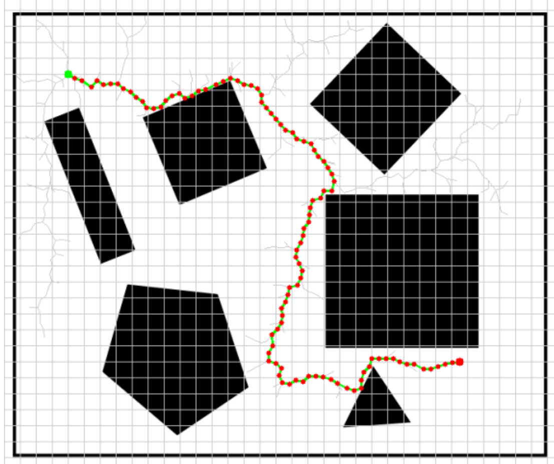


*Figure 15: RRT**

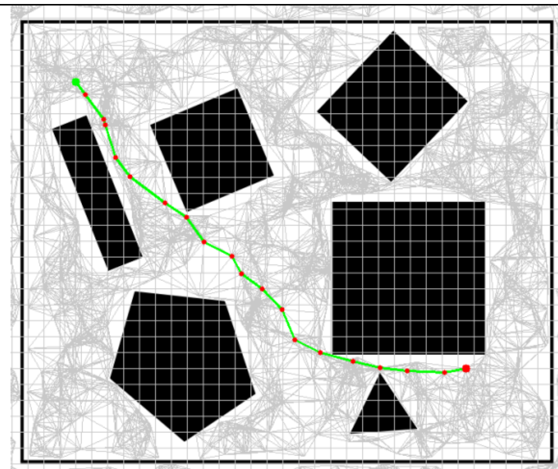


Figure 16: PRM

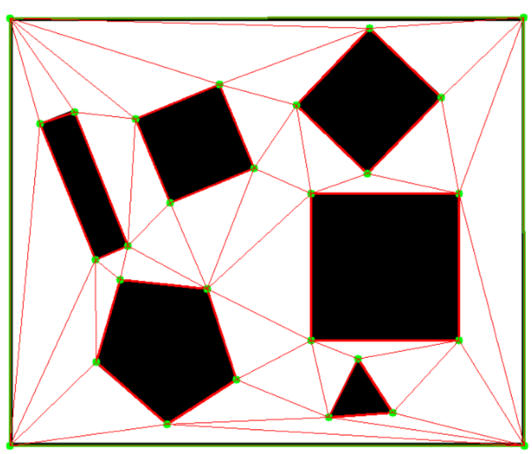


Figure 17: Convex Decomposition

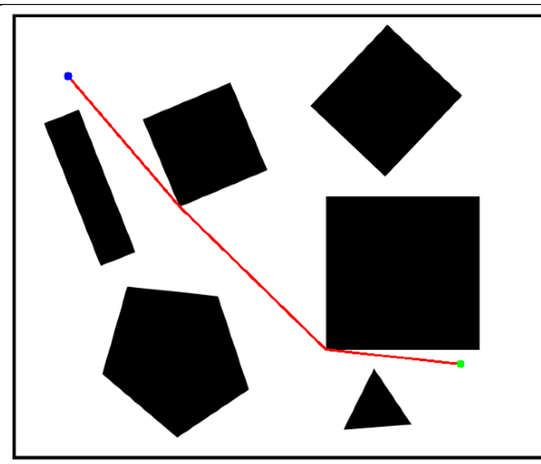


*Figure 18: Our A**

Map 1

Algorithm	Path Length	Variance	Nodes Expanded	Time Taken (sec)
A*	639.12	0.0335	491	0.00
RRT	754.48	0.33	217	0.02
RRT*	782.90	1.62	209	0.02
PRM	654.04	100.30	176	0.54
Our A*	613.07	1487.5899	5	0.00

Map 2

Algorithm	Path Length	Variance	Nodes Expanded	Time Taken (sec)
A*	653.76	0.0391	516	0.00
RRT	815.11	0.82	173	0.01
RRT*	1001.68	1.15	366	0.06
PRM	660.17	101.79	263	0.53
Our A*	639.17	1262.9521	8	0.00

Our results demonstrated that our improved A* algorithm consistently outperformed the other methods. By reducing the search space through map segmentation and leveraging the visibility graph, our algorithm achieved faster computation times and generated more optimal paths. The convex decomposition ensured that the environment's complexity was managed effectively, allowing for efficient navigation even in maps with irregular and complex obstacles. This experimental validation confirms the advantages of our approach in complex environments, highlighting its potential for applications requiring efficient and reliable path planning.

CONCLUSION

Our project successfully enhanced path planning in environments with complex obstacles by integrating map segmentation, visibility graphs, and an improved A* algorithm. Through image processing and constrained Delaunay triangulation, we decomposed the map into convex sub-polygons, enabling efficient navigation by reducing computational complexity. A visibility graph streamlined the search space, allowing the improved A* algorithm to find optimal paths with fewer nodes expanded and minimal computation time compared to traditional A*, RRT, and PRM algorithms. Bezier curve smoothing produced smoother, collision-free paths, making them practical for real-world applications like robotics and autonomous vehicles. User interaction and visualizations enhanced usability and process transparency. Experimental results demonstrated the approach's superiority in path length, computational efficiency, and adaptability, underscoring its potential for real-time applications. Future work could focus on dynamic environments, 3D spaces, machine learning integration, and real-time updates to extend the methodology's versatility further.

ANNEX

Convex Decomposition of the Map

```python
import cv2
import numpy as np
import matplotlib.pyplot as plt
from shapely.geometry import Polygon, MultiPolygon, LineString
from shapely.ops import triangulate, unary_union, polygonize
import triangle as tr
import math

def read_image(image_path):
    image = cv2.imread(image_path)
    if image is None:
        raise FileNotFoundError("Error: Image not found!")
    return image

def find_contours(image):
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    edges = cv2.Canny(gray, 50, 150)
    contours, hierarchy = cv2.findContours(
        edges, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE
    )
    if not contours or hierarchy is None:
        raise ValueError("No contours found!")
    return contours, hierarchy

def get_largest_contour(contours, hierarchy):
    max_area = 0
    map_boundary = None
    map_boundary_index = None
    for i, contour in enumerate(contours):
        if hierarchy[0][i][3] == -1:
            area = cv2.contourArea(contour)
            if area > max_area:
                max_area = area
                map_boundary = contour
                map_boundary_index = i
    if map_boundary is None:
        raise ValueError("Map boundary not found!")
    return map_boundary, map_boundary_index

def approximate_contour(contour, epsilon_factor=0.005):
    epsilon = epsilon_factor * cv2.arcLength(contour, True)
    approx = cv2.approxPolyDP(contour, epsilon, True)
    return [tuple(pt[0]) for pt in approx]

def sort_vertices_clockwise(vertices):
    centroid = np.mean(vertices, axis=0)
    return sorted(
        vertices, key=lambda v: np.arctan2(v[1] - centroid[1], v[0] - centroid[0])
    )

def is_descendant(contour_index, ancestor_index, hierarchy):
    parent_index = hierarchy[0][contour_index][3]
    while parent_index != -1:
        if parent_index == ancestor_index:
            return True
        parent_index = hierarchy[0][parent_index][3]
    return False

def remove_duplicate_vertices(vertices, threshold=10):
    if not vertices:
        return []

    cleaned_vertices = [vertices[0]]
    for vertex in vertices[1:]:
        last = cleaned_vertices[-1]
        distance = math.hypot(vertex[0] - last[0], vertex[1] - last[1])
        if distance >= threshold:
            cleaned_vertices.append(vertex)
        else:
            cleaned_vertices[-1] = vertex

    if len(cleaned_vertices) > 1:
        first = cleaned_vertices[0]
        last = cleaned_vertices[-1]
        distance = math.hypot(first[0] - last[0], first[1] - last[1])
```

```python
        if distance < threshold:
            cleaned_vertices.pop()

    return cleaned_vertices

def detect_obstacles(
    contours, hierarchy, map_boundary_index, map_boundary_area, vertices_dict
):
    obstacle_areas = []
    obstacle_count = 1
    for i, contour in enumerate(contours):
        if i == map_boundary_index:
            continue
        if is_descendant(i, map_boundary_index, hierarchy):
            area = cv2.contourArea(contour)
            if abs(area - map_boundary_area) < 20000:
                continue
            if any(abs(area - stored_area) < 1000 for stored_area in obstacle_areas):
                continue
            obstacle_areas.append(area)
            obstacle_vertices = approximate_contour(contour)
            obstacle_vertices = sort_vertices_clockwise(obstacle_vertices)
            obstacle_vertices = remove_duplicate_vertices(obstacle_vertices, threshold=10)
            if len(obstacle_vertices) < 3:
                print(f"Warning: Obstacle {obstacle_count} has less than 3 vertices after cleaning and will be skipped.")
                continue
            vertices_dict[f"Obstacle {obstacle_count}"] = obstacle_vertices
            obstacle_count += 1
    return vertices_dict

def classify_vertex_convexity(vertices):
    n = len(vertices)
    classifications = []
    for i in range(n):
        prev = np.array(vertices[(i - 1) % n])
        curr = np.array(vertices[i])
        next_v = np.array(vertices[(i + 1) % n])
        v1 = curr - prev
        v2 = next_v - curr
        cross_product = np.cross(v1, v2)
        classification = "Convex" if cross_product > 0 else "Concave"
        classifications.append((tuple(curr), classification))
    return classifications

def detect_polygon_orientation(vertices):
    area = 0.0
    n = len(vertices)
    for i in range(n):
        x1, y1 = vertices[i]
        x2, y2 = vertices[(i + 1) % n]
        area += (x1 * y2) - (x2 * y1)
    return area >= 0

def perform_constrained_triangulation(map_boundary_vertices, vertices_dict):
    if not detect_polygon_orientation(map_boundary_vertices):
        map_boundary_vertices = map_boundary_vertices[::-1]
        print("Reversed map boundary vertex order for positive orientation.")

    holes = []
    for key in vertices_dict:
        if key.startswith("Obstacle"):
            obstacle_vertices = vertices_dict[key]
            if detect_polygon_orientation(obstacle_vertices):
                obstacle_vertices = obstacle_vertices[::-1]
                print(f"Reversed {key} vertex order for negative orientation.")
            holes.append(obstacle_vertices)

    segments = []
    points = []
    point_marker_dict = {}
    point_index = 0

    for i, vertex in enumerate(map_boundary_vertices):
        points.append(vertex)
```

```python
            point_marker_dict[vertex] = point_index
            point_index += 1
    num_boundary_points = len(map_boundary_vertices)
    for i in range(num_boundary_points):
        idx1 = i
        idx2 = (i + 1) % num_boundary_points
        segments.append([idx1, idx2])

    for hole_vertices in holes:
        hole_point_indices = []
        for vertex in hole_vertices:
            if vertex not in point_marker_dict:
                points.append(vertex)
                point_marker_dict[vertex] = point_index
                point_index += 1
            hole_point_indices.append(point_marker_dict[vertex])
        num_hole_points = len(hole_vertices)
        for i in range(num_hole_points):
            idx1 = hole_point_indices[i]
            idx2 = hole_point_indices[(i + 1) % num_hole_points]
            segments.append([idx1, idx2])

    A = dict(vertices=np.array(points), segments=np.array(segments))

    hole_points = []
    for hole_vertices in holes:
        hole_polygon = Polygon(hole_vertices)
        x, y = hole_polygon.representative_point().coords[0]
        hole_points.append([x, y])

    A['holes'] = np.array(hole_points)

    B = tr.triangulate(A, 'p')

    sub_polygons = []
    for tri_indices in B.get('triangles', []):
        coords = [tuple(B['vertices'][idx]) for idx in tri_indices]
        sub_polygons.append(coords)
    return sub_polygons

def visualize_results(
    image, vertices_dict, convexity_results, sub_polygons, save_dir="Output", show_convexity=True
):
    import os

    os.makedirs(save_dir, exist_ok=True)

    output_image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB).copy()

    for key, vertices in vertices_dict.items():
        color = (0, 255, 0) if key == "Map Boundary" else (255, 0, 0)
        pts = np.array(vertices, np.int32).reshape((-1, 1, 2))
        cv2.polylines(output_image, [pts], True, color, 2)

        if show_convexity:
            for vertex, classification in convexity_results[key]:
                marker_color = (0, 255, 0) if classification == "Convex" else (255, 0, 0)
                cv2.circle(output_image, vertex, 5, marker_color, -1)

    boundary_image_path = os.path.join(save_dir, "Boundary_and_Obstacles.png")
    boundary_bgr = cv2.cvtColor(output_image, cv2.COLOR_RGB2BGR)
    cv2.imwrite(boundary_image_path, boundary_bgr)
    print(f"Saved boundary and obstacles image to {boundary_image_path}")

    plt.figure(figsize=(12, 10))
    plt.imshow(output_image)
    plt.title("Detected Map Boundary and Obstacles with Convex/Concave Vertices")
    plt.axis("off")
    plt.show()

    decomp_image = output_image.copy()
    for idx, poly in enumerate(sub_polygons):
        color = (255, 0, 0)
        pts = np.array(poly, np.int32).reshape((-1, 1, 2))
```

```python
            cv2.polylines(decomp_image, [pts], True, color, 1)

        decomposed_image_path = os.path.join(save_dir, "Convex_Decomposition.png")
        decomposed_bgr = cv2.cvtColor(decomp_image, cv2.COLOR_RGB2BGR)
        cv2.imwrite(decomposed_image_path, decomposed_bgr)
        #print(f"Saved convex decomposition image to {decomposed_image_path}")

        plt.figure(figsize=(12, 10))
        plt.imshow(decomp_image)
        plt.title("Convex Sub-polygons after Decomposition (Edges Only)")
        plt.axis("off")
        plt.show()

        height, width, _ = output_image.shape
        sub_polygons_image = np.ones((height, width, 3), dtype=np.uint8) * 255  # White background

        centroids = []
        all_edge_midpoints = []

        for poly in sub_polygons:
            pts = np.array(poly, np.int32).reshape((-1, 1, 2))
            cv2.polylines(sub_polygons_image, [pts], True, (0, 0, 0), 1)

            M = cv2.moments(pts)
            if M["m00"] != 0:
                cX = int(M["m10"] / M["m00"])
                cY = int(M["m01"] / M["m00"])
                centroids.append((cX, cY))
                cv2.circle(sub_polygons_image, (cX, cY), 3, (0, 0, 255), -1)
            else:
                continue

            for i in range(len(poly)):
                x1, y1 = poly[i]
                x2, y2 = poly[(i + 1) % len(poly)]
                mid_x = int((x1 + x2) / 2)
                mid_y = int((y1 + y2) / 2)
                all_edge_midpoints.append((mid_x, mid_y))
                cv2.circle(sub_polygons_image, (mid_x, mid_y), 3, (0, 255, 0), -1)  # Green dot

        sub_polygons_image_path = os.path.join(save_dir, "Sub_Polygons_Edges_Centroids_Midpoints.png")
        sub_polygons_bgr = cv2.cvtColor(sub_polygons_image, cv2.COLOR_RGB2BGR)
        cv2.imwrite(sub_polygons_image_path, sub_polygons_bgr)
        #print(f"Saved sub-polygons image with centroids and edge midpoints to {sub_polygons_image_path}")

        plt.figure(figsize=(12, 10))
        plt.imshow(sub_polygons_image)
        plt.title("Convex Sub-polygons with Centroids (Red) and Edge Midpoints (Green)")
        plt.axis("off")
        plt.show()

        centroids_file_path = os.path.join(save_dir, "Centroids.txt")
        with open(centroids_file_path, 'w') as f:
            for idx, (cX, cY) in enumerate(centroids, start=1):
                f.write(f"Centroid {idx}: ({cX}, {cY})\n")
        #print(f"Saved centroids to {centroids_file_path}")

        edge_midpoints_file_path = os.path.join(save_dir, "Edge_Midpoints.txt")
        with open(edge_midpoints_file_path, 'w') as f:
            for idx, (mid_x, mid_y) in enumerate(all_edge_midpoints, start=1):
                f.write(f"Edge Midpoint {idx}: ({mid_x}, {mid_y})\n")
        #print(f"Saved edge midpoints to {edge_midpoints_file_path}")

def detect_and_decompose_map(image_path):
    try:
        image = read_image(image_path)
        contours, hierarchy = find_contours(image)

        map_boundary, map_boundary_index = get_largest_contour(contours, hierarchy)
        map_boundary_vertices = approximate_contour(map_boundary)
        map_boundary_vertices = sort_vertices_clockwise(map_boundary_vertices)

        vertices_dict = {"Map Boundary": map_boundary_vertices}
        map_boundary_area = cv2.contourArea(map_boundary)
```

```python

        vertices_dict = detect_obstacles(
            contours, hierarchy, map_boundary_index, map_boundary_area, vertices_dict
        )

        convexity_results = {}
        for key, vertices in vertices_dict.items():
            convexity_results[key] = classify_vertex_convexity(vertices)

        print("\n=== Map Boundary and Obstacles Vertices ===")
        for key, vertices in vertices_dict.items():
            print(f"\n{key} Vertices ({len(vertices)} points):")
            for idx, vertex in enumerate(vertices, start=1):
                print(f"  Vertex {idx}: {vertex}")

        sub_polygons = perform_constrained_triangulation(map_boundary_vertices, vertices_dict)
        with open('Output/sub_polygons.txt', 'w') as f:
            print("\n=== Decomposed into the following convex sub-polygons (triangles) ===", file=f)
            for idx, poly in enumerate(sub_polygons, start=1):
                line = f"Sub-polygon {idx}: {poly}"
                print(line)
                f.write(line + '\n')

        visualize_results(
            image, vertices_dict, convexity_results, sub_polygons, save_dir="output", show_convexity=True
        )

        print("\n=== Decomposed into the following convex sub-polygons (triangles) ===")
        for idx, poly in enumerate(sub_polygons, start=1):
            print(f"Sub-polygon {idx}: {poly}")

    except Exception as e:
        print(str(e))

image_path = "Images/S1.png"
detect_and_decompose_map(image_path)
```

Optimized A* Algorithm

```python
import cv2
import numpy as np
import math
import heapq
import time
import os
from shapely.geometry import Polygon, Point, LineString
from shapely.ops import unary_union
from scipy.interpolate import CubicSpline
from functools import lru_cache
import itertools

class Node:
    """A node class for A* Pathfinding
    def __init__(self, position, g=0, h=0):
        self.position = position
        self.g = g
        self.h = h
        self.f = g + h
        self.parent = None

    def __lt__(self, other):
        return self.f < other.f

@lru_cache(maxsize=None)
def heuristic(a, b):
    """Euclidean distance heuristic"""
    return math.hypot(a[0] - b[0], a[1] - b[1])

def read_sub_polygons(sub_polygons_file_path):
    """Read sub-polygons from the file"""
    sub_polygons = []
    with open(sub_polygons_file_path, 'r') as f:
        for line in f:
            if line.startswith('Sub-polygon'):
                parts = line.strip().split(':', 1)
                coords_str = parts[1].strip().strip('[]')
                coords = []
```

```python
                for coord_pair in coords_str.split('), ('):
                    coord_pair = coord_pair.strip('() ')
                    if coord_pair:
                        x_str, y_str = coord_pair.split(',')
                        x, y = float(x_str.strip()), float(y_str.strip())
                        coords.append((int(x), int(y)))
                sub_polygons.append(coords)
    return sub_polygons

def build_visibility_graph(sub_polygons, start_point, end_point):
    """
    Build a visibility graph using obstacle vertices, start, and end points.
    """
    obstacle_polygons = [Polygon(polygon) for polygon in sub_polygons]
    obstacle_union = unary_union(obstacle_polygons)
    obstacle_vertices = set()
    for polygon in sub_polygons:
        obstacle_vertices.update(polygon)
    points = list(obstacle_vertices)
    points.append(start_point)
    points.append(end_point)

    graph = {point: [] for point in points}

    for i, point1 in enumerate(points):
        for j, point2 in enumerate(points):
            if i >= j:
                continue

            line = LineString([point1, point2])

            # Check if the line intersects any obstacle
            if not obstacle_union.crosses(line):
                distance = heuristic(point1, point2)
                graph[point1].append((point2, distance))
                graph[point2].append((point1, distance))

    return graph

def astar_visibility_graph(graph, start, end):
    """A* pathfinding algorithm on the visibility graph."""
    counter = itertools.count()
    open_heap = []
    open_entry_finder = {}
    closed_set = set()

    nodes_expanded = 0

    start_node = Node(start, g=0, h=heuristic(start, end))
    entry = (start_node.f, next(counter), start_node)
    heapq.heappush(open_heap, entry)
    open_entry_finder[start] = entry

    while open_heap:
        _, _, current_node = heapq.heappop(open_heap)
        nodes_expanded += 1

        if current_node.position in closed_set:
            continue

        closed_set.add(current_node.position)

        if current_node.position == end:
            path = []
            while current_node:
                path.append(current_node.position)
                current_node = current_node.parent
            return path[::-1], nodes_expanded

        for neighbor_pos, cost in graph[current_node.position]:
            if neighbor_pos in closed_set:
                continue

            g_cost = current_node.g + cost
```

```python
            h_cost = heuristic(neighbor_pos, end)
            f_cost = g_cost + h_cost

            if neighbor_pos in open_entry_finder:
                existing_entry = open_entry_finder[neighbor_pos]
                existing_node = existing_entry[2]
                if g_cost < existing_node.g:
                    neighbor_node = Node(neighbor_pos, g=g_cost, h=h_cost)
                    neighbor_node.parent = current_node
                    entry = (neighbor_node.f, next(counter), neighbor_node)
                    heapq.heappush(open_heap, entry)
                    open_entry_finder[neighbor_pos] = entry
            else:
                neighbor_node = Node(neighbor_pos, g=g_cost, h=h_cost)
                neighbor_node.parent = current_node
                entry = (neighbor_node.f, next(counter), neighbor_node)
                heapq.heappush(open_heap, entry)
                open_entry_finder[neighbor_pos] = entry
    return None, nodes_expanded

def calculate_path_length_and_variance(path):
    """Calculate path length and variance"""
    if not path or len(path) < 2:
        return 0, 0
    distances = [
        heuristic(path[i], path[i + 1])
        for i in range(len(path) - 1)
    ]
    return sum(distances), np.var(distances) if len(distances) > 1 else 0

def visualize_sparse_path(img, path, title="Path Found"):
    """Visualize the path on the image"""
    img_color = img.copy()
    for i in range(len(path) - 1):
        cv2.line(img_color, (int(path[i][0]), int(path[i][1])), (int(path[i + 1][0]), int(path[i + 1][1])), (0, 0, 255), 2)
    cv2.circle(img_color, (int(path[0][0]), int(path[0][1])), 5, (255, 0, 0), -1)  # Blue start
```

```python
    cv2.circle(img_color, (int(path[-1][0]), int(path[-1][1])), 5, (0, 255, 0), -1)  # Green end
    cv2.imshow(title, img_color)
    cv2.waitKey(0)
    cv2.destroyAllWindows()
    return img_color

def smooth_path_bezier(path, obstacle_polygons):
    """Smooth the path using Bezier curves while avoiding obstacles."""
    path = np.array(path)
    num_points = max(100, len(path) * 10)

    t = np.linspace(0, 1, len(path))
    x = path[:, 0]
    y = path[:, 1]

    bezier_x = np.polyfit(t, x, deg=min(3, len(path)-1))
    bezier_y = np.polyfit(t, y, deg=min(3, len(path)-1))
    bezier_curve_x = np.poly1d(bezier_x)
    bezier_curve_y = np.poly1d(bezier_y)

    t_smooth = np.linspace(0, 1, num_points)
    x_smooth = bezier_curve_x(t_smooth)
    y_smooth = bezier_curve_y(t_smooth)

    smoothed_path = []
    collision = False
    for i in range(len(x_smooth) - 1):
        point1 = (x_smooth[i], y_smooth[i])
        point2 = (x_smooth[i + 1], y_smooth[i + 1])
        line = LineString([point1, point2])

        intersects = False
        for obstacle in obstacle_polygons:
            if line.crosses(obstacle):
                intersects = True
```

```
                collision = True
                break
        if not intersects:
            smoothed_path.append(point1)
        else:
            break

    if collision:
        print("Collision detected during path smoothing with Bezier. Using original path.")
        return [tuple(p) for p in path]
    else:
        smoothed_path.append((x_smooth[-1], y_smooth[-1]))
        smoothed_path = [tuple(p) for p in smoothed_path]
        return smoothed_path

def main():
    img_path = "Images/S1.png"
    sub_polygons_file_path = "Output/sub_polygons.txt"

    img = cv2.imread(img_path)
    if img is None:
        print("Error: Unable to read the image. Check the path.")
        return

    sub_polygons = read_sub_polygons(sub_polygons_file_path)
    if not sub_polygons:
        print("Error: No sub-polygons found. Ensure sub_polygons.txt is properly formatted.")
        return

    obstacle_polygons = [Polygon(polygon) for polygon in sub_polygons]
    obstacle_union = unary_union(obstacle_polygons)

    img_color = img.copy()

    choice = input("Enter '1' to input coordinates manually or '2' to select on the image: ").strip()
    if choice == '1':
        try:
            start_x = int(input("Enter start x-coordinate: "))
            start_y = int(input("Enter start y-coordinate: "))
            end_x = int(input("Enter end x-coordinate: "))
            end_y = int(input("Enter end y-coordinate: "))
            start_point = (start_x, start_y)
            end_point = (end_x, end_y)

            graph = build_visibility_graph(sub_polygons, start_point, end_point)
            start_time = time.time()
            path, nodes_expanded = astar_visibility_graph(graph, start_point, end_point)  # Adjusted to receive nodes_expand
            end_time = time.time()

            if path:
                path_length, path_variance = calculate_path_length_and_variance(path)
                print(f"Path Length: {path_length:.2f}, Variance: {path_variance:.4f}, "
                      f"Nodes Expanded: {nodes_expanded}, Time Taken: {end_time - start_time:.2f}s")
                visualize_sparse_path(img_color, path, title="Initial Path")

                smoothed_path = smooth_path_bezier(path, obstacle_polygons)
                smoothed_length, smoothed_variance = calculate_path_length_and_variance(smoothed_path)
                print(f"Smoothed path! Length: {smoothed_length:.2f}, Variance: {smoothed_variance:.4f}")

                visualize_sparse_path(img_color, smoothed_path, title="Smoothed Path")
            else:
                print(f"No path found! Nodes Expanded: {nodes_expanded}")

        except ValueError:
            print("Invalid coordinates. Please enter valid integers.")
    elif choice == '2':
        points = []

        def mouse_callback(event, x, y, flags, param):
            nonlocal points
            if event == cv2.EVENT_LBUTTONDOWN:
                points.append((x, y))
                color = (255, 0, 0) if len(points) == 1 else (0, 255, 0)
                cv2.circle(img_color, (x, y), 5, color, -1)
```

```python
            cv2.imshow("Select Start and Goal", img_color)

            if len(points) == 2:
                start_point, end_point = points

                graph = build_visibility_graph(sub_polygons, start_point, end_point)
                start_time = time.time()
                path, nodes_expanded = astar_visibility_graph(graph, start_point, end_point)  # Adjusted to receive node:
                end_time = time.time()

                if path:
                    path_length, path_variance = calculate_path_length_and_variance(path)
                    print(f"Initial path found! Length: {path_length:.2f}, Variance: {path_variance:.4f}, "
                          f"Nodes Expanded: {nodes_expanded}, Time Taken: {end_time - start_time:.2f}s")
                    visualize_sparse_path(img_color, path, title="Initial Path")

                    smoothed_path = smooth_path_bezier(path, obstacle_polygons)
                    smoothed_length, smoothed_variance = calculate_path_length_and_variance(smoothed_path)
                    print(f"Smoothed path! Length: {smoothed_length:.2f}, Variance: {smoothed_variance:.4f}")

                    visualize_sparse_path(img_color, smoothed_path, title="Smoothed Path")
                else:
                    print(f"No path found! Nodes Expanded: {nodes_expanded}")

        cv2.imshow("Select Start and Goal", img_color)
        cv2.setMouseCallback("Select Start and Goal", mouse_callback)
        cv2.waitKey(0)
        cv2.destroyAllWindows()
    else:
        print("Invalid choice. Exiting.")

if __name__ == '__main__':
    main()
```

REFERENCES

[1] Zhaoying L, Ruoling S, Zhao Z. A new path planning method based on sparse A* algorithm with map segmentation. Transactions of the Institute of Measurement and Control. 2022;44(4):916-925. doi:10.1177/01423312211046410