Genetic Algorithm Based Approach for Mobile Robot Path Planning

FNU Koustubh
dir id: koustubh@umd.edu

Keyur Borad
dir id: kborad@umd.edu

Aryan Mishra
dir id: amishr17@umd.edu

***Abstract*—We propose the development of a genetic algorithm-based path planner for autonomous mobile robots, capitalizing on the principles of evolutionary computation to enhance path optimization. Genetic algorithms simulate natural evolutionary processes such as selection, crossover, and mutation, thereby enabling the iterative improvement of solutions across generations. In this study, we aim to integrate a GA-based path planner as an adaptive layer atop existing probabilistic sampling-based path planning methods, such as Rapidly-exploring Random Trees (RRT) or Probabilistic Roadmaps (PRM). The proposed approach is designed to refine initial path solutions generated by these conventional methods, optimizing path characteristics such as length, safety, and navigational efficiency. By employing a genetic algorithm, our planner iteratively evolves a population of path solutions, effectively adapting and enhancing route selection in complex environments. The integration promises to mitigate common limitations of probabilistic methods, including sub-optimality and computational inefficiency in dense or dynamic settings. Results indicate that the genetic algorithm layer not only refines paths to higher degrees of optimality but also introduces robustness against environmental changes and obstacles, making it a promising enhancement for autonomous navigation systems. This paper details the algorithmic framework, results, implementation challenges, and the potential impact of this hybrid approach on the future of robotic path planning.**

***Index Terms*—Genetic Algorithm, Path Planning, PRM, RRT, Fitness Function, Crossover, Mutation**

I. Introduction

Genetic Algorithms (GA) are a class of evolutionary algorithms that mimic the process of natural selection, embodying the survival of the fittest concept to solve optimization and search problems. These algorithms are particularly effective in complex optimization scenarios where traditional methods might struggle due to the vastness of the search space or the non-linearity of the problem.

In the context of path planning for autonomous mobile robots, the objective is often to find the most efficient route from a start point to a destination. Efficiency can be defined in various terms such as the shortest distance, the least time, or the safest path, depending on specific application requirements. Traditional path planning methods, such as A* or Dijkstra's algorithm, provide precise solutions by systematically exploring the possible paths. However, in highly complex or dynamic environments, these methods can become computationally expensive or may fail to find an optimal path due to their deterministic nature.

This is where Genetic Algorithms come into play. A GA-based path planner does not work to find a single best path in one go but rather evolves the solution over many iterations, making it well-suited for applications where the environment may change or where there are numerous local optima that a conventional algorithm might mistakenly consider as the global optimum.

II. Literature Review

A. Aim

To significantly improve the efficiency of genetic algorithms (GAs) in solving path optimization problems, it is essential to refine their operational mechanics and selection processes. By optimizing the crossover and mutation strategies can achieve more precise and quicker convergence, leading to enhanced overall performance in finding optimal paths.

B. Description

This paper introduces a novel crossover operator specifically engineered to prevent premature convergence in genetic algorithms while ensuring the generation of feasible paths. This operator is designed to produce offspring with superior fitness values compared to their parent solutions, thereby enhancing the efficiency and effectiveness of path optimization tasks.

It also proposes a new fitness function that integrates multiple critical considerations—distance, safety, and energy usage. This multi-faceted approach ensures that the paths not only optimize for shortness but also maintain high standards of safety and energy efficiency, making the algorithm suitable for complex environments where these factors are crucial.

C. Methodology involved

The approach modifies genetic algorithms' (GAs) crossover operators to handle variable-length chromosomes, essential for maintaining feasible paths in path optimization. By adapting crossover techniques, it ensures that the resulting paths are both feasible and optimized, accommodating changes in path lengths dynamically during the optimization process.

Introduces a sophisticated algorithm designed to enhance genetic diversity within the population while maintaining the feasibility of paths. This algorithm strategically adjusts mutation rates and incorporates diverse genetic material, ensuring robust solutions without compromising the practical applicability of the paths, thereby optimizing performance and solution diversity in genetic algorithms.

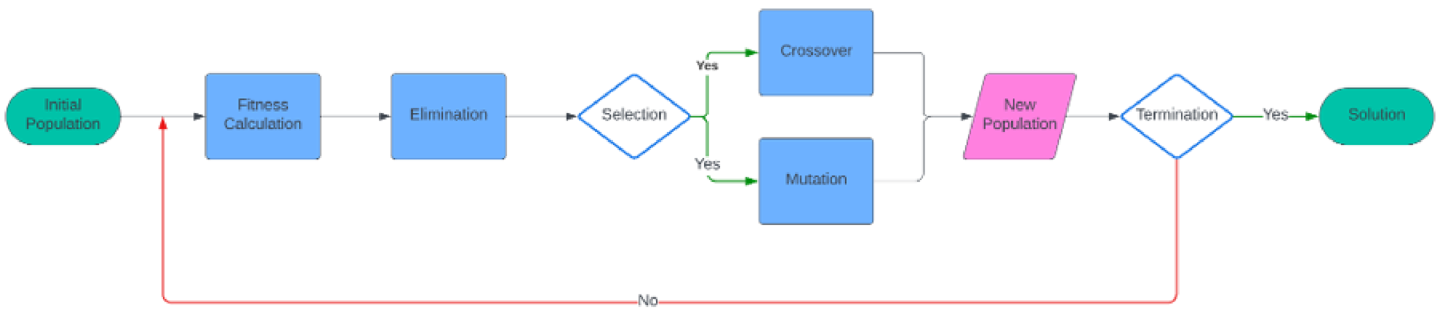

Fig. 1. Algorithm Workflow

III. Algorithm

A. Initial Population

The initial population is a collection of paths from the start to the goal node, in terms of natural selection here, they represent the chromosomes that crossover with each other. In this paper, the initial population is generated using the RRT: Rapidly-Exploring Random Trees even though other methods such as PRM, DAG, etc can be used.

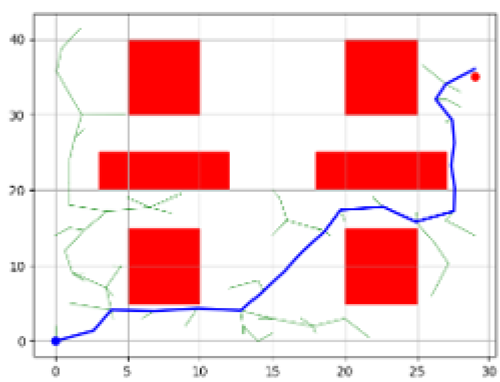

Fig. 2. Initial Population Generated via RRT

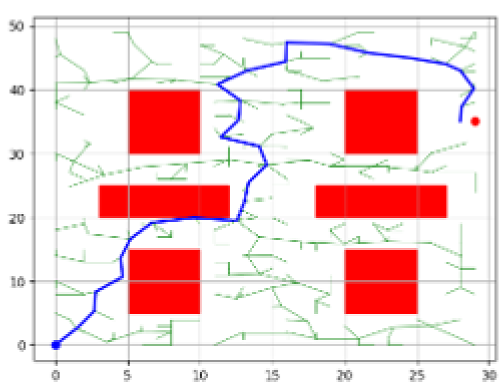

Fig. 3. Initial Population Generated via RRT

1) Probabilistic Road-maps Algorithm:

- Developed as a multi-query approach, effective in static environments where the obstacle configuration does not change.
- The essence of PRM lies in constructing a roadmap of randomly sampled configurations in the free space, which are nodes not obstructed by obstacles.
- During the pre-processing phase:
  - The algorithm randomly generates points in the environment.
  - Connects these points with simple paths, typically straight lines, to form a graph.
  - Connections are made only if the path between points does not intersect any obstacles, ensuring all paths on the roadmap are valid.
- Once the roadmap is built, path planning between any start and end points can be quickly accomplished by:
  - Connecting these points to the nearest nodes on the roadmap.
  - Searching for the shortest path between them using standard graph search algorithms like Dijkstra's or A*.
- This preprocessing makes PRM highly effective for scenarios where the environment is known and remains unchanged, allowing for rapid queries once the initial roadmap is established.

2) Rapidly Exploring Random Trees:

- RRT is favoured for scenarios where the environment is highly dynamic or contains numerous obstacles, as it efficiently searches non-convex, high-dimensional spaces.
- Core mechanism:
  - Involves selecting random points in the search space.
  - Connecting these points to the nearest vertex in the tree.
  - Each iteration of the algorithm extends the tree from the nearest vertex towards the randomly chosen point until a maximum allowed distance is reached.
- The algorithm's performance and efficiency stem from its ability to cover the search space extensively while

naturally avoiding obstacles by effectively steering clear of them during the extension of the branches.

Algorithm 1 Rapidly-exploring Random Trees (RRT)

Initialize nodes with starting node at start coordinates
while length of nodes $<$ max_nodes **do**
  Generate random_point within the environment size
  if random_point is within obstacles **then**
    Continue
  end if
  nearest $\leftarrow$ nearest_node(nodes, random_point)
  new_node $\leftarrow$ steer(nearest, random_point, step_size)
  if not is_within_obstacles(new_node) and is_valid_path(nearest, new_node) **then**
    new_node.parent_node $\leftarrow$ nearest
    nodes.append(new_node)
    if distance(new_node, goal) $\leq$ step_size **then**
      return nodes, new_node
    end if
  end if
end while
return nodes, None =0

B. Fitness Calculation

The fitness evaluation of each path generated in the algorithm is calculated based on three key parameters: total distance, energy consumption, and safety. The following equation represents the fitness function:

$$F = \frac{1}{w_1 \cdot l} + \frac{1}{w_2 \cdot H} + \frac{1}{w_3 \cdot SFL} \tag{1}$$

where:

$$\begin{aligned} F &= \text{Fitness of the path,} \\ l &= \text{Total length of the path,} \\ H &= \text{Sum of change in heading in the path} \\ SFL &= \text{Sum of points near and inside the path,} \\ w_1, w_2, w_3 &= \text{Weights to tune the fitness function.} \end{aligned}$$

- **Total Distance**: calculated using the Euclidean distance between each successive node from start to goal.

$$l = \sum_{i=0}^{N-1} \sqrt{(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2} \tag{2}$$

  where N is the total number of nodes
- **Energy Consumption**: The energy consumption depends on the frequency of direction changes.
  An optimal path will typically feature fewer, less intense turns compared to a non-optimal path, which may cover a greater distance and exhibit more variations in heading as the robot moves from one node to another.
  To consider the smoothness of the path, we calculate the difference in heading when the robot moves from one node to another. This calculation is performed as follows:

$$\begin{aligned} H = \sum_{i=0}^{N-2} atan2(y_{i+2} - y_{i+1}, x_{i+2} - x_{i+1})... \\ ... - atan2(y_{i+1} - y_i, x_{i+1} - x_i) \end{aligned} \tag{3}$$

  where N is the total number of nodes
- **Safety First Level (SFL)**: assesses the proximity of solution paths to potential obstacles. A greater distance from obstacles reduces the likelihood of collisions, enhancing the safety and reliability of the path. It allows for deviations without increased collision risks. The calculation involves selecting a node, defining a safety radius around it, and counting how many points within this radius intersect with obstacles. Higher counts indicate greater proximity to obstacles, as shown in Figure 6.

Algorithm 2 Fitness Function Evaluation for a Path

coordinates $\leftarrow$ list of (node.x, node.y) for each node in path
Define weights: $w_1 = 3$, $w_2 = 1$, $w_3 = 2$
Initialize $euc\_dist = 0$, $angle\_sum = 0$, $interference = 0$
Define $safe\_rad = 8$
for i in range(0, length of coordinates - 1) **do**
  $x_1, y_1 \leftarrow$ coordinates[i]
  $x_2, y_2 \leftarrow$ coordinates[i+1]
  $dist \leftarrow \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$
  $euc\_dist \mathrel{+}= dist$
  if $i \neq$ length of coordinates - 2 **then**
    $x_3, y_3 \leftarrow$ coordinates[i+2]
    $heading1 \leftarrow$ degrees(atan2($y_2 - y_1$, $x_2 - x_1$))
    $heading2 \leftarrow$ degrees(atan2($y_3 - y_2$, $x_3 - x_2$))
    Normalize headings to be between 0 and 360
    $angle \leftarrow |heading2 - heading1|$
    $angle\_sum \mathrel{+}= angle$
  end if
end for
for each $(center\_x, center\_y)$ in coordinates **do**
  for each (ox, oy, ex, ey) in obstacles **do**
    for i in range(ox, ex) **do**
      for j in range(oy, ey) **do**
        if $((i - center\_x)^2 + (j - center\_y)^2) < safe\_rad^2$ **then**
          $interference \mathrel{+}= 1$
        end if
      end for
    end for
  end for
end for
$F \leftarrow w_1 \cdot (1/euc\_dist) + w_2 \cdot (1/angle\_sum) + w_3 \cdot (1/interference)$
return euc_dist, F =0

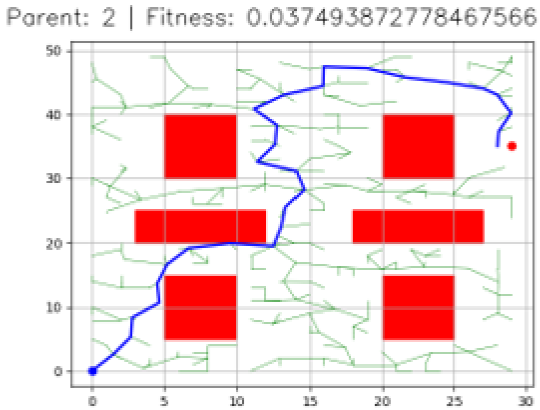

Fig. 4. Fitness value of Parent 2

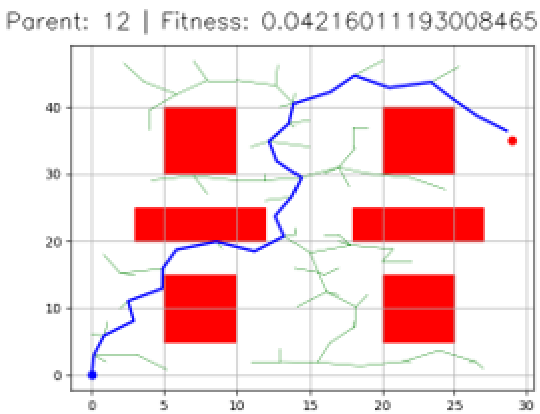

Fig. 5. Fitness value of Parent 12

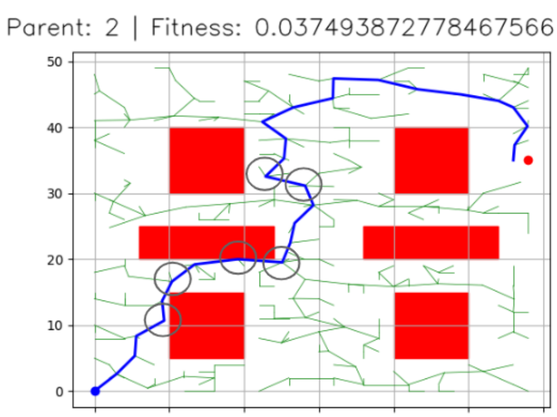

Fig. 6. Visualization of safety circles on each node

C. Selection and Crossover

1) Selection: The selection step is one of the most important steps in the genetic algorithm, it ensures that the best solutions which have a high fitness continue to pass their traits to the next generations and the solutions that are poor with a low level of fitness are eliminated. There are various methods to perform the selection procedure, some of which are listed below:

- Elitism
- Tournament Selection
- Roulette Selection
- Random Selection
- Stochastic Universal Sampling
- Linear Rank Selection
- Exponential Rank Selection
- Truncation Selection

This approach uses Elitism, a selection technique in genetic algorithms (GAs), to ensure the preservation of the best individuals from one generation to the next. By doing so, it prevents the loss of optimal solutions due to the random nature of selection, crossover, and mutation processes. The population is ranked by fitness score and divided into two groups; individuals are alternately assigned to each group based on their rank. For instance, the highest-ranked individual is placed in the first group, the second highest in the second group, and so forth. This setup leads to pairs of parents from which new generations are created through crossover.

Algorithm 3 Selection Procedure

```
Initialize P1 as an empty list
Initialize P2 as an empty list
while fitness list is not empty do
    max_fitness ← maximum value in fitness
    index ← index of max_fitness in fitness
    Remove max_fitness from fitness
    Append population[index] to P1
    Remove population[index] from population
    l ← length of fitness
    i ← random integer between 0 and l − 1
    Remove fitness[i] from fitness
    Append population[i] to P2
    Remove population[i] from population
end while
return (P1, P2) =0
```

2) Crossover: Crossover functions are pivotal in genetic algorithms, facilitating the convergence to optimal solutions by generating offspring with superior traits from their parents. Various advanced crossover techniques have been developed, such as Order Crossover (OX), Partially-Mapped Crossover (PMX), Cycle Crossover (CX), and the widely-used Same Point (SP) Crossover. Munemoto's implementation, for example, features a multiple-point crossover that targets multiple identical genes. Enhancements to the SP Crossover include the

Same Adjacent Crossover, which focuses on nodes adjacent to identical ones instead of each node directly.

In our simpler approach, we employ a single-point crossover. We compare nodes generated by probabilistic methods (like RRT, PRM) from two parents, selecting the crossover point at the node with the smallest inter-node distance, excluding start and end nodes to avoid negligible changes in fitness. Typically, a midpoint crossover results in more significant alterations in fitness values compared to those near the endpoints.

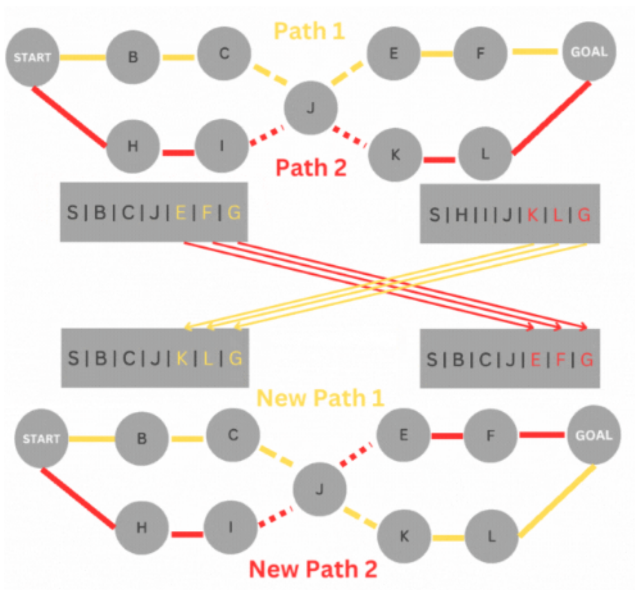

Fig. 7. Crossover

In Figure 6, we can observe the following:

- **Path 1:** Highlighted in yellow, this route progresses from the starting point to the goal. It traverses through a sequence of waypoints: B, C, J, E, and F. This path is designed to illustrate a specific trajectory within the scenario.
- **Path 2:** Highlighted in red, this path charts a different route from the start to the goal, passing through waypoints H, I, J, K, and L. It represents an alternative approach, offering a contrasting path through the same environment.
- **Common Crossover Point:** Both paths intersect at waypoint J, which serves as a pivotal crossover point. This commonality suggests a significant junction within the navigational framework of the scenario.

  Path Modification Strategy:
  - **Construction of New Path 1:** The segment of Path 1 leading up to the crossover point J (including points B to J) is merged with the segment from Path 2 that extends beyond point J (including points J to L). This hybrid path is then highlighted in yellow, illustrating a new, potentially more efficient route.
  - **Construction of New Path 2:** In a reciprocal manner, the segment of Path 2 leading up to point J (including points H to J) is combined with the segment from Path 1 that extends beyond point J (including points J to F). The resulting new path is highlighted in red, depicting another innovative route configuration.

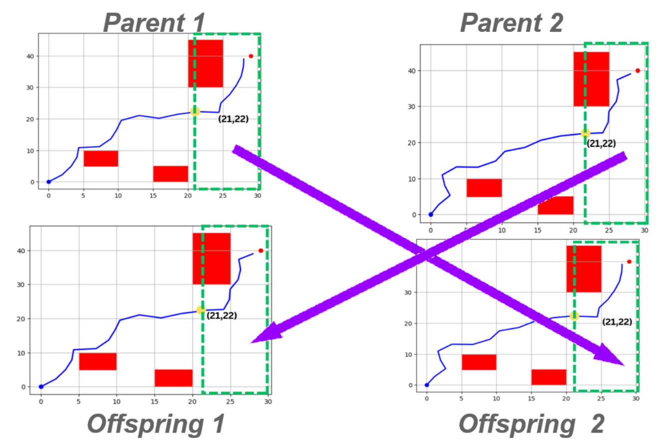

Fig. 8. Crossover Path Representation

Algorithm 4 Crossover Point Selection

Input: parent1, parent2
Output: offspring1, offspring2
Initialize minval to 2
Initialize offspring1, offspring2 as empty lists
Declare parent1_idx, parent2_idx
for pt1 $= 3$ to $\text{len}(parent1) - 3$ **do**
  for pt2 $= 3$ to $\text{len}(parent2) - 3$ **do**
    dist $\leftarrow \sqrt{(parent1[pt1].x - parent2[pt2].x)^2 + (parent1[pt1].y - parent2[pt2].y)^2}$
    if dist $<$ minval **then**
      minval $\leftarrow$ dist
      parent1_idx $\leftarrow$ pt1
      parent2_idx $\leftarrow$ pt2
    end if
  end for
end for
for i $= 0$ to parent1_idx **do**
  offspring1.append(Node(parent1[i].x, parent1[i].y))
end for
for i $=$ parent2_idx to $\text{len}(parent2)$ **do**
  offspring1.append(Node(parent2[i].x, parent2[i].y))
end for
for i $= 0$ to parent2_idx **do**
  offspring2.append(Node(parent2[i].x, parent2[i].y))
end for
for i $=$ parent1_idx to $\text{len}(parent1)$ **do**
  offspring2.append(Node(parent1[i].x, parent1[i].y))
end for
return offspring1, offspring2 =0

Figure 8 demonstrates the single-point crossover of two-parent paths at the coordinate values (21, 22). Following this coordinate, the path nodes are swapped between the two parent paths, resulting in the generation of two new offspring.

D. Mutation

Mutation in the genetic algorithm is a mechanism designed to maintain genetic diversity within the population, critical for the success of these algorithms in solving complex optimization problems.

Mutation allows the algorithm to explore a wider range of potential solutions and helps prevent the population from converging at sub-optimal solutions.

To maintain diversity in the paths generated and achieve mutation, the best two paths from each generation are selected. The mutation operator is then applied to these paths, by selecting a random node from the path and displacing it in a random direction at an arbitrary distance. This results in a new mutated version of the original path. The mutated paths are then added to the next generation.

In Figure 9, a random node is selected and in Figure 10, that node is moved in a down-left direction, resulting in a mutation of the original path.

Algorithm 5 Mutation

Input: path
Output: new_path
num_nodes ← length of path
random_node ← random integer from 0 to num_nodes - 1
del_x ← random float from -1 to 1
del_y ← random float from -1 to 1
new_path ← deep copy of path
new_path[random_node].x ← new_path[random_node].x + del_x
new_path[random_node].y ← new_path[random_node].y + del_y
return new_path =0

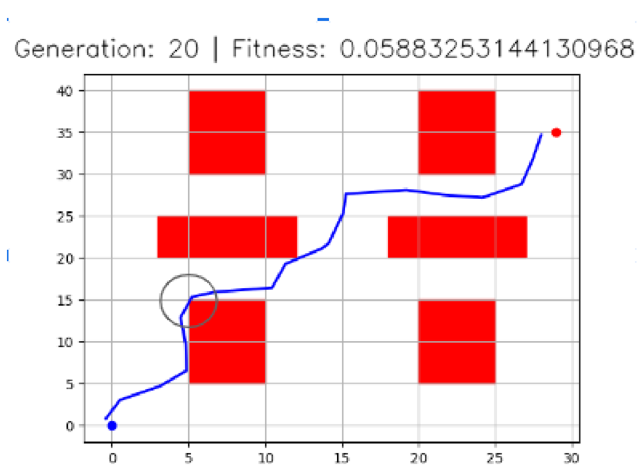

Fig. 9. Random selection of a node on a valid path

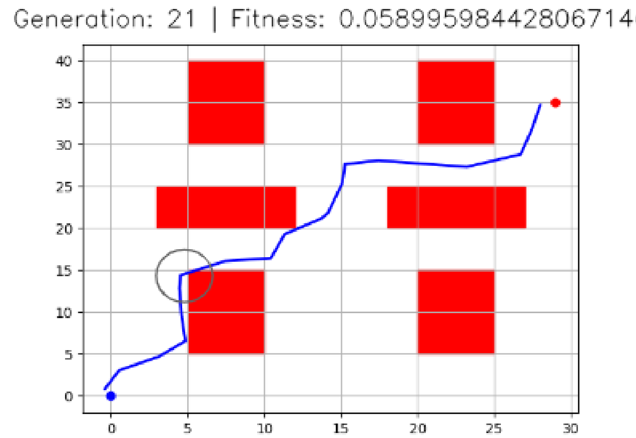

Fig. 10. Moving the node in a random direction

IV. Results and Discussion

A. Results

1) Basic Setup: For our experimentation, we have chosen a 30 x 50 grid size for simpler, faster implementation. In our grid, we have made rectangle-shaped obstacles with gaps in between so that the rrt algorithm has multiple and diverse approaches to the goal position. In our grid, obstacles are visualised in red colour. Furthermore, to maximise the exploration, we have selected the start and goal points as (0,0) and (29,35), respectively.

2) Initial population generation: We have generated a population size of 20 randomly generated paths for a given start and goal coordinates. The figure shown below demonstrates the initial population using a Rapidly explored Random Tree. This sample population can be termed as generation zero as well. Some of the sample initial population generations are shown in Figure 11.

3) Applying Genetic Algorithm: After applying the genetic algorithm step by step, as mentioned in section III, that is:

- Applying Fitness Function
- Elimination
- Selection
- Crossover between selected parent paths
- Mutation over selected paths
- Appending paths to new population
- Termination

Over multiple iterations, we get successive generations with improved fitness function values. Below shown in figure 12 shows different generations with fitness values.

B. Problems Faced

The following problems have been faced us while implementing the project:

- **Fitness Calculation for Safety Second Level**
  Calculating fitness for the "Safety Second Level" (SSL) presents a challenge because it involves assessing the safety of a path not just based on distance from obstacles

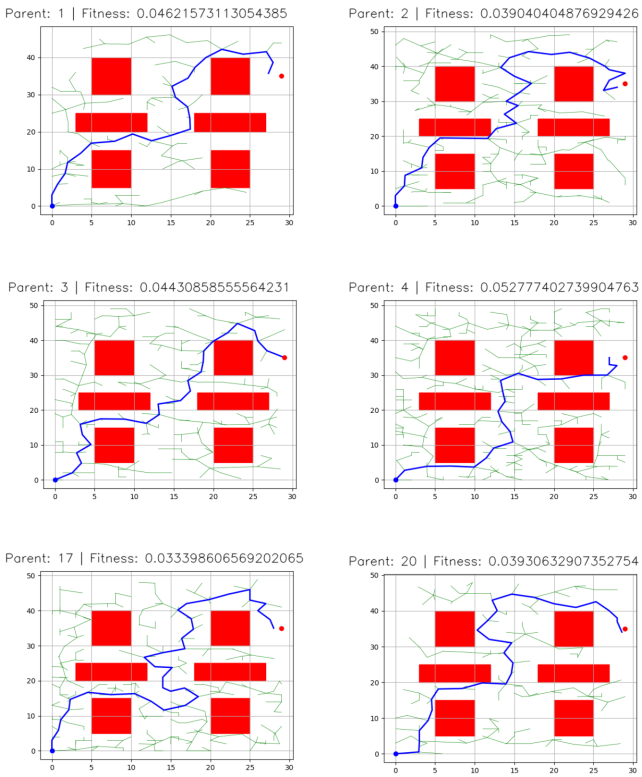

Fig. 11. Initial population generated using RRT

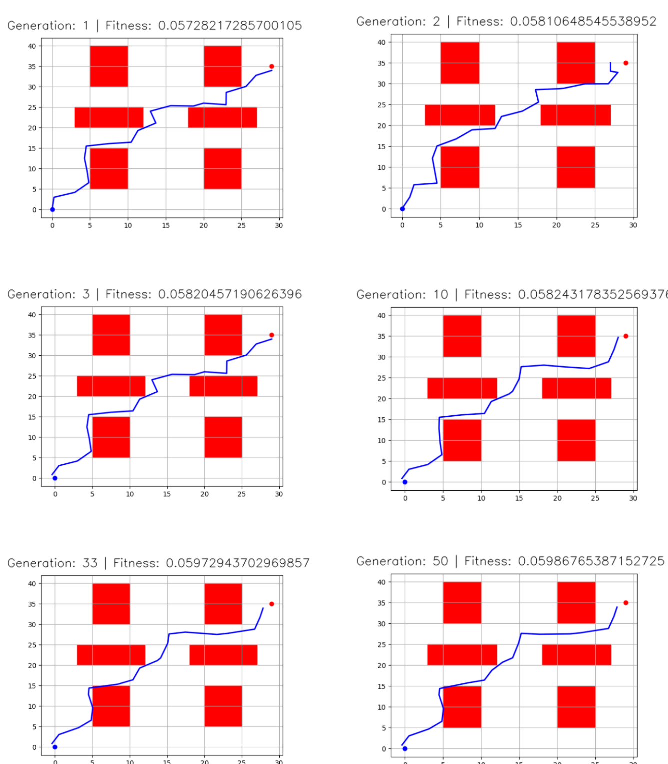

Fig. 12. Best solutions of each generation generated by GA

(as in the first level, SFL), but also considering additional factors such as the robot's speed, manoeuvrability, and the dynamic changes in the environment. This level might include evaluating how well a path adapts to sudden obstacles appearing or how effectively it utilizes safe stopping and evasion manoeuvres.

- **Inadequacies in Mutation Function**
  Mutation functions in evolutionary algorithms are crucial for introducing variability into the population, helping to escape local optima and explore new areas of the solution space. Inadequacies in the mutation function can lead to insufficient exploration, premature convergence, or excessive diversity that fails to refine towards an optimal solution. For instance, a mutation function might not adequately account for the specific constraints
- **Initial Population and Number of Generations**
  An initial population that is not diverse enough or poorly represents the solution space can hinder the algorithm's ability to find the best solutions. Similarly, the number of generations must be balanced to allow sufficient time for convergence without leading to unnecessary computational expense. Too few generations can result in underfitting, while too many might lead to overfitting or wasted computational resources.

C. *Conclusions*

In this technical report, we endeavoured to implement path optimization for given start and goal positions using sampling-based methods. We observed that genetic algorithms based on natural selection tend to converge to the optimal solution more rapidly and with fewer iterations compared to other sampling-based approaches. The choice of crossover strategy notably impacts the convergence toward minima/maxima. Additionally, selecting an appropriate mutation strategy aids in identifying global minima/maxima. Lastly, the elimination strategy ensures that our solution remains focused on the desired minima/maxima. While the computational time difference may not be readily apparent in smaller maps tested during our experiments, it becomes significant in larger maps/canvases.

The overall average fitness of the solutions generated by **RRT** was **0.03976487298743** and the average fitness of solutions after applying the additional layer of **genetic algorithm** was **0.05986876299423**. Which results in an improvement of **50.5569073819 %** in the fitness of the path generated.

D. *Future Works*

- **Robust Fitness Function based on Terrain**
  function that takes into account the characteristics of the terrain could significantly enhance path planning algorithms. This fitness function would evaluate paths not only based on distance and safety margins but also considering the terrain's impact on the robot's mobility. For instance, it could factor in terrain roughness, slope, and stability, which affect the robot's speed and energy consumption. This would help in generating more

practical and efficient routes in varied environmental conditions.

- **Potent Mutation Function and Different Types for Better Generations**
  Enhancing the mutation function in genetic algorithms could lead to better exploration of the solution space and generation of more optimal paths. Implementing a variety of mutation types could tailor the evolutionary process to the specific needs of the path planning, such as adaptive mutation rates or context-sensitive mutations that change based on the type of terrain or the stage of the evolution process. This approach could help in maintaining diversity in the population and preventing premature convergence.
- **Initial Population Calculated with Simpler or Faster Path Planning Algorithm**
  Using a simpler or faster path-planning algorithm to generate the initial population could streamline the evolutionary process. Algorithms like A* or Dijkstra's could be used to quickly generate a set of feasible paths that cover different areas of the solution space. These initial paths would then be refined through the evolutionary process, combining the benefits of both deterministic and stochastic approaches to achieve both efficiency and thoroughness in the search process.
- **Path Smoothing**
  Smoothing would reduce abrupt changes in direction and speed, which are critical for the practical implementation of robot navigation. Techniques such as cubic splines or Bezier curves could be used to create smoother, more naturally flowing paths that are easier for robots to follow and more efficient in terms of energy consumption and time management.

Acknowledgments

We would like to express my gratitude to our Professor, Dr. Reza Monfaredi, for the invaluable support and guidance throughout the course and broadening the horizon of knowledge.

[1] [2] [3] [4] [5] [6] [9] [8] [7]

References

[1] Maram Alajlan, Anis Koubaa, Imen Chaari, Hachemi Bennaceur, and Adel Ammar. Global path planning for mobile robots in large-scale grid environments using genetic algorithms. In *2013 International Conference on Individual and Collective Behaviors in Robotics (ICBR)*, pages 1–8. IEEE, 2013.

[2] Arakil Chentoufi, Abdelhakim El Fatmi, Ali Bekri, Said Benhlima, and Mohamed Sabbane. Genetic algorithms and dynamic weighted sum method for rna alignment. In *2017 Intelligent Systems and Computer Vision (ISCV)*, pages 1–5. IEEE, 2017.

[3] Yanrong Hu and Simon X Yang. A knowledge based genetic algorithm for path planning of a mobile robot. In *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA'04. 2004*, volume 5, pages 4350–4355. IEEE, 2004.

[4] Chaymaa Lamini, Said Benhlima, and Ali Elbekri. Genetic algorithm based approach for autonomous mobile robot path planning. *Procedia Computer Science*, 127:180–189, 2018. PROCEEDINGS OF THE FIRST INTERNATIONAL CONFERENCE ON INTELLIGENT COMPUTING IN DATA SCIENCES, ICDS2017.

[5] Masaharu Munetomo, Yoshiaki Takai, and Yoshiharu Sato. A migration scheme for the genetic adaptive routing algorithm. In *SMC'98 Conference Proceedings. 1998 IEEE International Conference on Systems, Man, and Cybernetics (Cat. No. 98CH36218)*, volume 3, pages 2774–2779. IEEE, 1998.

[6] Zhang Qiongbing and Ding Lixin. A new crossover mechanism for genetic algorithms with variable-length chromosomes for path optimization problems. *Expert Systems with Applications*, 60:183–189, 2016.

[7] Noraini Mohd Razali, John Geraghty, et al. Genetic algorithm performance with different selection strategies in solving tsp. In *Proceedings of the world congress on engineering*, volume 2, pages 1–6. International Association of Engineers Hong Kong, China, 2011.

[8] Thomas Tometzki and Sebastian Engell. Systematic initialization techniques for hybrid evolutionary algorithms for solving two-stage stochastic mixed-integer programs. *IEEE Transactions on Evolutionary Computation*, 15(2):196–214, 2010.

[9] Yong Zhang, Lin Zhang, and Xiaohua Zhang. Mobile robot path planning base on the hybrid genetic algorithm in unknown environment. In *2008 eighth international conference on intelligent systems design and applications*, volume 2, pages 661–665. IEEE, 2008.

Appendix

A. Python Code for Path Planning Algorithm

```
import matplotlib.pyplot as plt
import numpy as np
import random
import math
import cv2
import copy

size = (30, 50)
obstacles = [(5, 5, 10, 15), (20, 5, 25, 15), (5,
    30, 10, 40), (20, 30, 25, 40), (3, 20, 12, 25),
    (18, 20, 27, 25)]
start = (0, 0)
plot_num = 0
goal = (29, 35)
INITIAL_POPULATION_SIZE = 20
NUMBER_OF_GENERATIONS = 50
STEP_SIZE = 3
FPS = 2
generation_best_fitness = []

class Node:
  def __init__(self, x, y):
    self.x = x
    self.y = y
    self.parent_node = None

def is_within_obstacles(point, obstacles):
  for (x1, y1, x2, y2) in obstacles:
    if x1 <= point[0] <= x2 and y1 <= point[1] <= y2
    :
      return True
  return False

def distance(point1, point2):
  return np.sqrt((point1[0]-point2[0])**2 + (point1
    [1]-point2[1])**2)

def nearest_node(nodes, random_point):
  return min(nodes, key=lambda node: distance((node.
    x, node.y), random_point))

def steer(from_node, to_point, step_size=1):
  if distance((from_node.x, from_node.y), to_point)
    < step_size:
    return Node(to_point[0], to_point[1])
  else:
    theta = np.arctan2(to_point[1]-from_node.y,
    to_point[0]-from_node.x)
```

```
    return Node(from_node.x + step_size * np.cos(theta), from_node.y + step_size * np.sin(theta))

def is_valid_path(node1, node2, obstacles):
  steps = int(distance((node1.x, node1.y), (node2.x, node2.y)) / 0.5)  # Smaller steps for more accuracy
  for i in range(1, steps + 1):
    inter_x = node1.x + i * (node2.x - node1.x) / steps
    inter_y = node1.y + i * (node2.y - node1.y) / steps
    if is_within_obstacles((inter_x, inter_y), obstacles):
      return False
  return True

def plot(nodes=None, path=None):
    global plot_num
    fig, ax = plt.subplots()
    if nodes:
      for node in nodes:
          if node.parent_node:
              plt.plot([node.x, node.parent_node.x], [node.y, node.parent_node.y], "g-", linewidth=0.5)
    for (ox, oy, ex, ey) in obstacles:
        ax.add_patch(plt.Rectangle((ox, oy), ex-ox, ey-oy, color="red"))
    if path:
        plt.plot([node.x for node in path], [node.y for node in path], "b-", linewidth=2)  # Highlight path in blue
    plt.plot(start[0], start[1], "bo")  # Start
    plt.plot(goal[0], goal[1], "ro")  # Goal
    plt.grid(True)
    plt.savefig(f'plot_{plot_num + 1}.png')  # Save plot as PNG
    plot_num += 1

def rrt(step_size=1, max_nodes=10000):
    nodes = [Node(start[0], start[1])]
    while len(nodes) < max_nodes:
        random_point = (random.randint(0, size[0] - 1), random.randint(0, size[1] - 1))
        if is_within_obstacles(random_point, obstacles):
            continue
        nearest = nearest_node(nodes, random_point)
        new_node = steer(nearest, random_point, step_size)
        if not is_within_obstacles((new_node.x, new_node.y), obstacles) and is_valid_path(nearest, new_node, obstacles):
            new_node.parent_node = nearest
            nodes.append(new_node)
            if distance((new_node.x, new_node.y), goal) <= 2:
                return nodes, new_node
    return nodes, None

def fitness_function(path):
    coordinates = [(node.x, node.y) for node in path]
    w1 = 3
    w2 = 1
    w3 = 2
    euc_dist = 0
    angle_sum = 0
    interference=0
    safe_rad=8
    for i in range(len(coordinates)-1):
        x1, y1 = coordinates[i]
        x2, y2 = coordinates[i+1]
        dist_ = math.sqrt((x2-x1)**2 +(y2-y1)**2)
        euc_dist += dist_
        if i != len(coordinates)-2:
          x3, y3 = coordinates[i+2]
          heading1 = math.degrees(math.atan2((y2-y1),(x2-x1)))
          heading2 = math.degrees(math.atan2((y3-y2),(x3-x2)))
          if heading1 < 0:
            heading1 = 360 + heading1
          if heading2 < 0:
            heading2 = 360 + heading2
          angle = abs(heading2-heading1)
          angle_sum += angle

    for center_x,center_y in coordinates:
      for (ox , oy, ex, ey) in obstacles:
        for i in range(ox,ex):
          for j in range(oy,ey):
            if ((i-center_x)**2+(j-center_y)**2)<safe_rad**2:
              interference+=1

    F = w1*(1/(euc_dist))+ w2*1/(angle_sum)+w3*(1/interference)

    F = w1*(1/(euc_dist))+ w2*1/(angle_sum)
    return euc_dist, F

def calculate_fitness_of_population(population):
  fitness = []
  for index, path in enumerate(population):
    euc_dist, F = fitness_function(path)
    fitness.append(F)
  return fitness

def selection(fitness, population):
  P1 = []
  P2 = []
  while(fitness):
    max_fitness = max(fitness)
    index = fitness.index(max_fitness)
    fitness.pop(index)
    P1.append(population.pop(index))

    l = len(fitness)
    i = random.randint(0, l-1)
    fitness.pop(i)
    P2.append(population.pop(i))
  return P1, P2

def best_selection(fitness, population):
  P1 = []
  P2 = []
  while(fitness):
    max_fitness = max(fitness)
    index = fitness.index(max_fitness)
    fitness.pop(index)
    P1.append(population.pop(index))

    max_fitness = max(fitness)
    index = fitness.index(max_fitness)
    fitness.pop(index)
    P2.append(population.pop(index))
  return P1, P2

def plot_best_solution(fitness, population, plot_graph=False):
  max_fitness = max(fitness)
  index = fitness.index(max_fitness)
  best_path = population[index]
  print(f"Fitness: {max_fitness}")
  if plot_graph:
    plot(path=best_path)
```

```
  return max_fitness

#CrossOver points
def crossoverpt(parent1,parent2):
    minval=2
    offspring1=[]
    offspring2=[]
    #range defined in such a way that it ignores inital and final points
    for pt1 in range(3, len(parent1)-3):
        for pt2 in range(3, len(parent2)-3):
            #Calculating  distance between every nodes of 2 paths and selecting the min distance path points
            if (math.sqrt((parent1[pt1].x-parent2[pt2].x)**2+(parent1[pt1].y-parent2[pt2].y)**2))<minval:
                minval=math.sqrt((parent1[pt1].x-parent2[pt2].x)**2+(parent1[pt1].y-parent2[pt2].y)**2)
                #storing path points indexes so to use them to crossover
                parent1_point_idx=pt1
                parent2_point_idx=pt2

    # Pruning the path and making crossover based on the indexes calculated
    for i in range(parent1_point_idx+1):
        offspring1.append(Node(parent1[i].x,parent1[i].y))
    for i in range(parent2_point_idx,len(parent2)):
        offspring1.append(Node(parent2[i].x,parent2[i].y))

    for i in range(parent2_point_idx+1):
        offspring2.append(Node(parent2[i].x,parent2[i].y))
    for i in range(parent1_point_idx,len(parent1)):
        offspring2.append(Node(parent1[i].x,parent1[i].y))

    #Returning offsprings
    return offspring1,offspring2

def mutate_path(path):
  num_nodes = len(path)
  random_node = random.randint(0,num_nodes-1)
  del_x = random.random()*2 - 1
  del_y = random.random()*2 - 1
  new_path = copy.deepcopy(path)
  new_path[random_node].x += del_x
  new_path[random_node].y += del_y
  return new_path

def elimination(fitness, population):
  num_eliminations = 2
  for i in range(0, num_eliminations):
    min_fitness = min(fitness)
    index = fitness.index(min_fitness)
    fitness.pop(index)
    population.pop(index)
  return fitness, population

def create_initial_population(population_size=20, step_size=1, plot_paths = True):
  population = []
  for i in range(population_size):
    nodes, final_node = rrt(step_size=step_size)
    path = []
    if final_node:
        while final_node.parent_node:
            path.append(final_node)
            final_node = final_node.parent_node
        path.reverse()
    if plot_paths:
      plot(nodes, path)
    population.append(path)
  return population

def create_opencv_visualisation(parent_gen_size=INITIAL_POPULATION_SIZE, generation_gen_size=100):
  fourcc = cv2.VideoWriter_fourcc(*'mp4v')
  video = cv2.VideoWriter("genetic.mp4", fourcc, FPS, (640, 480))
  for i in range(0, parent_gen_size+generation_gen_size):
    image = cv2.imread(f'plot_{i+1}.png')
    if i < parent_gen_size:
      fitness = parent_gen_fitness[i]
      cv2.putText(image, f"Parent: {i+1} | Fitness: {fitness}",(10,40),cv2.FONT_HERSHEY_SIMPLEX,0.8, (0,0,0), 1, cv2.LINE_AA)
    else:
      fitness = generation_best_fitness[i-parent_gen_size]
      cv2.putText(image, f"Generation: {i-parent_gen_size+1} | Fitness: {fitness}",(10,40),cv2.FONT_HERSHEY_SIMPLEX,0.8, (0,0,0), 1, cv2.LINE_AA)
    video.write(image)
  for i in range(20):
    video.write(image)

population = create_initial_population(population_size=INITIAL_POPULATION_SIZE, step_size=STEP_SIZE, plot_paths=True)
parent_gen_fitness = calculate_fitness_of_population(population)

for gen in range (0,NUMBER_OF_GENERATIONS):
  print(f"Generation: {gen+1}")
  if population:
    fitness = calculate_fitness_of_population(population)
    best_fitness = plot_best_solution(fitness=fitness, population=population, plot_graph=True)
    generation_best_fitness.append(best_fitness)
    P1, P2 = best_selection(fitness, population)
    mutation_1 = mutate_path(P1[0])
    mutation_2 = mutate_path(P2[0])
    population.append(mutation_1)
    population.append(mutation_2)
    for i in range(0,len(P1)):
      try:
        offspring1 , offspring2 = crossoverpt(P1[i],P2[i])
        population.append(offspring1)
        population.append(offspring2)
      except:
        pass
  else:
    NUMBER_OF_GENERATIONS = gen
    print(f"Termination at generation: {gen+1}")

create_opencv_visualisation(parent_gen_size=INITIAL_POPULATION_SIZE, generation_gen_size=NUMBER_OF_GENERATIONS)
```