University of Victoria

ECE 470: Machine Learning Algorithm for PCB Trace Routing

Team Name: Trace

Recipient: Dr. Kin Fun Li

Submission Date: August 1st, 2025

Team Members:

Iain Graham V00792466

Alex Rockson V00979147

Kevin Zhou V00961727

Table of Contents

Executive Summary

This report presents a genetic algorithm-based approach to solve a path finding and color connection game by using Python. This challenge arises in areas such as printed circuit board (PCB) routing [1], traffic flow optimization [2], and puzzle-solving applications [3] where path intersections can lead to signal interference, congestion, or invalid configurations. The genetic algorithm is used as an approach and compares its performance against other three heuristic strategies: 1) user-defined order, 2) pre-defined order and 3) greedy search. The implementation uses an A* path search with Manhattan heuristic to find each path segment while treating existing lines as obstacles. The performance of the approaches is evaluated by the number of successfully connected color pairs and total path length. This report details the genetic algorithm design and provides empirical results, background on pathfinding and evolutionary computation is discussed.

1. Introduction

1.1 Background

The game considered here involves an 10x10 grid with 5 pairs of colored endpoints. A valid solution draws a continuous path of the same color between each pair of endpoints, and no two paths intersect with each other. Traditional pathfinding like A* search can connect a single pair optimally, but choosing the order to connect multiple pairs is complex due to global interactions like paths block each other.

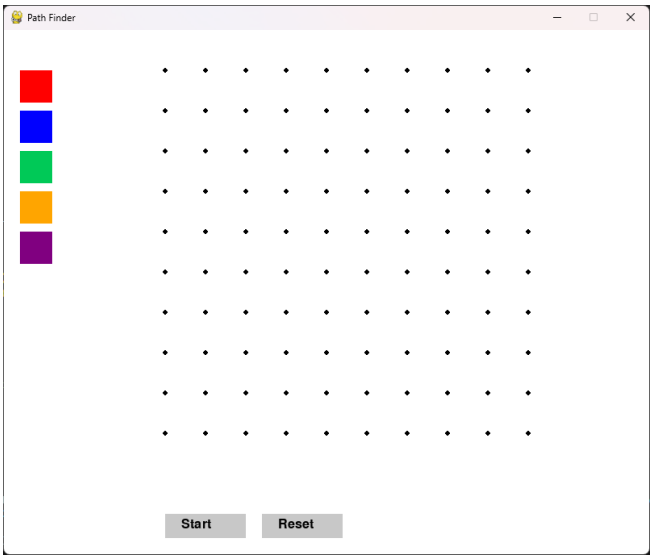


Figure 1: Path finding UI

A common way is to use the A* algorithm [4] for individual pairs, as it guarantees an optimal path on a weighted grid given an admissible heuristic. In practice, when several paths are drawn sequentially, early paths create obstacles that A* must work around, so the chosen connection order strongly affects feasibility and total length. Heuristic strategies can perform well on easy instances but might fail to find the best global solution on harder problems.

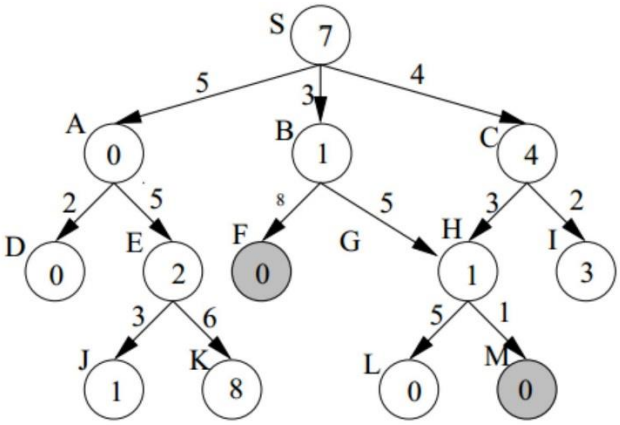


Figure 2: A algorithm*

Genetic algorithms are an alternative approach that evolve a population of candidate solutions over time. Gas mimic biological evolution: a population of solution-encodings is iteratively updated by selection, crossover, and mutation [5]. Over many generations, the population can converge to good solutions. In our context, each chromosome encodes an order to connect the colors. By evaluating each ordering via simulated pathfinding and assigning a fitness, the GA attempts to evolve an ordering that yields a well-performed connection pattern.

1.2 Motivation

This project is driven by both educational and practical goals. It demonstrates how genetic algorithms (GAs) can solve a concrete path-planning problem that is easy to visualize and test. By comparing GA-based methods with simpler heuristics, the study highlights the advantages and limitations of evolutionary search when tackling NP-hard challenges. Similar planning tasks appear in fields such as VLSI routing [6], robotic routing and network design, making the approach broadly relevant. The interactive Python implementation, with a Pygame interface [7], facilitates hands-on experimentation and clear demonstrations of key ideas. Performance is measured not only by the number of color pairs successfully connected, but also by the extra path length required.

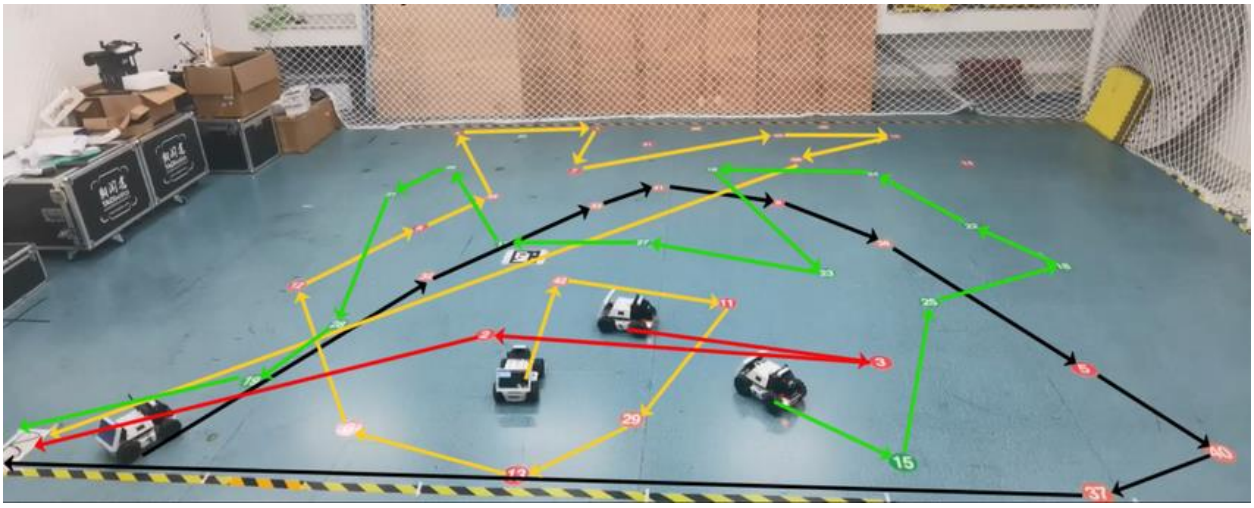


Figure 3: Robotic routing

2. Related Work

Pathfinding and connection puzzles have been widely studied. One approach encodes the entire grid as a graph of cells and uses satisfiability or constraint solving: Torvaney maps the puzzle to a SAT problem [8].

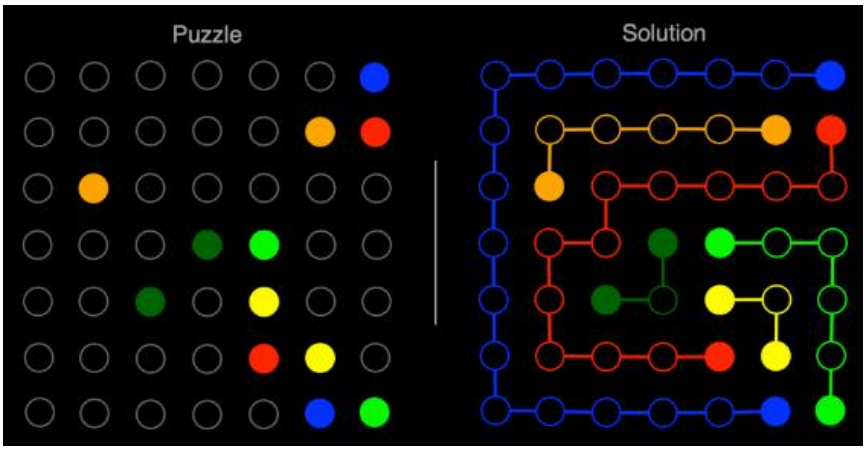


Figure 4: Torvaney SAT problem

Heuristic search methods are commonly used. A* search is the standard algorithm for shortest-path problems in similar projects. Simply connecting pairs in the order given by the user requires no computation but can be suboptimal. Greedy strategies sort colors by the straight-line distance between endpoints, connecting the shortest first, to prioritize

easier paths. Fixed priority lists are another heuristic. These methods are effective for simple layouts but may leave some pairs unconnected on more complex maps.

Genetic algorithms have been applied to routing and path planning in related domains. Sourabh and his team used a GA to explore variations of the A* algorithm in a game simulation, finding that evolved heuristics could yield near-optimal paths more quickly [5].

3. Problem Formulation

The problem is formalized as follows. The grid is a 10x10 lattice of cells. There are 5 distinct colors (red, blue, green, orange, purple). For each color that is used, exactly two cells. For each color, there are two endpoints assigned to that color. The task is to draw a path connecting the two endpoints of each color but without sharp turn and no intersection with other paths.

The objective is: maximize the number of colors successfully connected and minimize the total length of each path. Therefore, the fitness measure: F = 1000*C – L where C is the connected path count and L is total length, which strongly prioritizes connecting more colors and then shorter length. This reflects the assumption that covering all color pairs is the primary goal. Once all colored endpoints have been placed on the 10×10 grid and the user presses the “Start” button, the program immediately launches into its solution phase. As each strategy runs, real-time progress indicators update to show how many connections have been made and the current total length for that approach.

There are three heuristic approaches that have been used for comparison for completeness vs. simplicity. The baseline strategies are:

- **User-Order Strategy:** Connect colors in the order the user placed the endpoints. The solution order is predetermined by user input, with A* searches performed sequentially.
- **Priority-Order Strategy:** Connect colors in a fixed priority list. This ignores the user’s placement order and uses a pre-defined order.
- **Greedy Strategy:** Compute the Manhattan distance between the two endpoints of each color. Then connect colors in ascending distance order: shortest first, this aims to connect the easiest pair first.

Each strategy attempts to run A* for each color pair in its order, checking if the resulting path intersects any already drawn path. If an intersection occurs, that color is skipped. Thus, each method returns a mapping of some subset of colors to their connecting paths.

- **Genetic Algorithm:** The GA Strategy encodes a candidate solution as a permutation of the valid colors. For instance, the order of five colors will be represented in a list. The GA proceeds as follows:
    - **Initialization:** Generate a population of fifty random individuals, each a random shuffle of all colors.
    - **Fitness Evaluation:** For each individual, the program tries to connect colors in that order using the same A* algorithm, accumulating a temporary solution. Suppose *i* colors are connected and the total path length is *L_i*. The fitness function will be f = 1000* *i* - *L_i*.
    - **Selection:** Parents are chosen by roulette-wheel selection proportionally to fitness. To handle negative fitness, all fitness scores are adjusted to at least 1. If all fitness is equal, the parents are chosen randomly.
    - **Crossover:** Two parents are paired, and with probability CROSS_RATE = 0.8, two cut points are chosen and copy the sub-order between them from parent 1 into the child in the same positions. Then fill the remaining slots with the colors from parent 2 in their original order but skip the duplicate ones. If crossover is not applied, the child is basically a copy of parent 1.
    - **Mutation:** With a small probability MUT_RATE = 0.3, the swap mutation is performed on the child: pick two positions randomly and exchange the colors at those positions.
    - **Replacement and Elitism:** Each generation, replace the population entirely with the new children. After evaluating the population, keep track of the best solution found so far. Then terminate either after a fixed number of generations or achieve a solution that connects all colors.

In code, these steps are implemented in `run_genetic_algorithm()`. The function iterates GEN_MAX times, and each time computing fitness for all individuals, then updating the best solution, after that creating a new population by repeatedly selecting parents, applying crossover/mutation and collecting children.

The performance on test instances by counting how many colors were connected by each method and summing their path lengths was recorded. Compared GA with the heuristics, the results show that GA might take more time, but it can always find a best solution at the end.

- **Grid Size:** For the grid size selection, by having a 10x10 grid could achieve an ideal balance between complexity and clarity: it's large enough to produce intersecting paths for multiple color pairs and also small enough for GA and A* converge within seconds. A uniform 10x10 layout ensures fair, reproducible comparisons across

different algorithms under consistent conditions. During preliminary testing, population size, crossover/mutation rates and iteration counts optimized for 10x10 delivered stable, high-quality solutions, scaling up would demand extensive parameter retuning and significantly increase computational cost. Therefore, this choice guarantees clear visualization, manageable runtime, and repeatable experiments.

4. Methodology and Evaluation

4.1 Algorithms and Implementation

The GUI uses Pygame to allow interactive placement of colored endpoints and visualization of solutions, the core algorithms are independent of the interface. The key components are as follows:

- **A*:** A standard A* grid planner (`GridPathFinder.find_path`) to connect two given endpoints, treating existing path cells as obstacles. Each pixel coordinate is mapped to a grid cell by rounding `(x-spacing)/spacing` [10]. The cost g(n) is the distance traveled, and the Manhattan distance `h(n) = |x_goal - x| + |y_goal - y|` as the heuristic. After A* finding an optimal path for each pair, convert the resulting grid path back to pixel coordinates for drawing.
- **Obstacle Handing:** Once a path segment is drawn (a sequence of grid cells), it will be marked as obstacles for subsequent paths. The Bresenham algorithm [9] is used to add all intervening cells between consecutive points on the path. Additionally, all non-endpoint-colored dots are treated as obstacles. This ensures paths do not overlap or pass through other colors' endpoints.

```
while True:
    self.obstacles.add((current_x, current_y))
    if current_x == end_grid_x and current_y == end_grid_y:
        break
    error_double = 2 * error
    if error_double > -delta_y: # update x
        error -= delta_y
        current_x += step_x
    if error_double < delta_x: # update y
        error += delta_x
        current_y += step_y
```

Figure 5: Bresenham algorithm

- **User-Order Strategy:** The program records the order in which the user clicked to place colors. The function `user_order_connect()` iterate over the list and then apply A* algorithm for the colors.

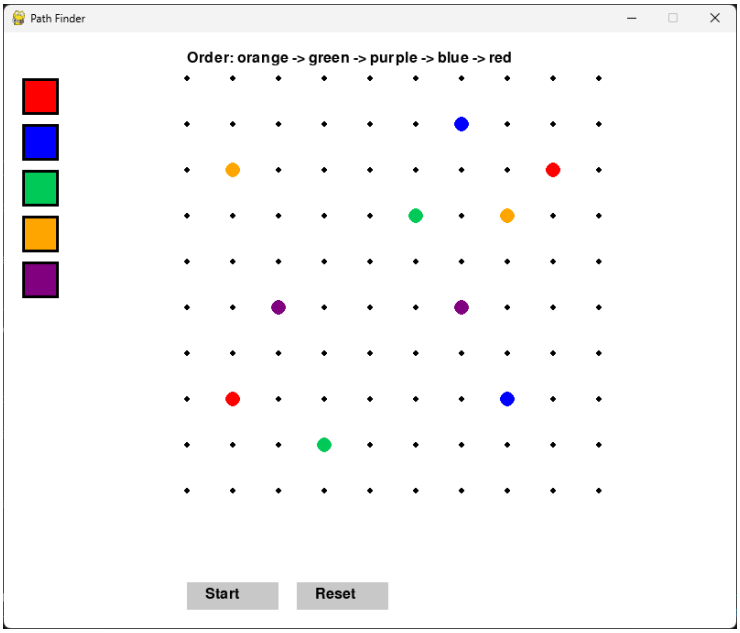


Figure 6: User-order on the UI

```
Color orange is added into the placement order, current order is: ['orange']
Color green is added into the placement order, current order is: ['orange', 'green']
Color purple is added into the placement order, current order is: ['orange', 'green', 'purple']
Color blue is added into the placement order, current order is: ['orange', 'green', 'purple', 'blue']
Color red is added into the placement order, current order is: ['orange', 'green', 'purple', 'blue', 'red']
```

Figure 7: User-Order added into Color Order List

- **Priority-Order Strategy:** There is also a pre-defined order of colors for the program to apply A* algorithm.

```
def priority_connect():
    "Connect based on the priority"
    PRIORITY_ORDER = ["red", "blue", "green", "orange", "purple"]
```

Figure 8: Pre-defined Color Order

- **Greedy Search:** The Manhattan distance is the core parameter for this strategy; after calculating the distance, the colors are sorted by their distance and apply A* algorithm on the color with the shortest distance first under the hope they are easier. This is a common heuristic in path puzzles, though not guaranteed optimal.

```
for color in valid_colors:
    point1, point2 = points[color]
    # Conver to grid coordinate
    grid_x1, grid_y1 = path_finder._screen_to_grid(point1)
    grid_x2, grid_y2 = path_finder._screen_to_grid(point2)
    # Find the Manhattan distance
    distance = abs(grid_x1 - grid_x2) + abs(grid_y1 - grid_y2)
    color_distances.append((distance, color))
```

Figure 9: Greedy search

- **Genetic Algorithm:**
  - **Population:** There are POP_SIZE random permutations of the valid colors are generated. This ensures each individual is in complete ordering of all colors.
  - **Fitness:** The `evaluate(individual)` function simulates connection in the order given by the individual. It attempts each color sequency exactly as in the user-order method. It counts the number of connected color pairs and the Euclidean length of the paths. Therefore, the fitness is: `1000 * connected_count – total_length` since the connected count is more significant.
  - **Selection:** In `select_parent()`, the roulette-wheel selection is implemented. Also adjust all possible negative fitness to at least 1. This favors high-fitness solutions but still allows exploration.
  - **Crossover:** Two cut points are randomly picked and then copy the middle segment from parent1 to the child and fill in the remaining slots with the colors from parent2 in order. If the random check fails, the child is just a copy of parent1.
  - **Mutation:** In `mutate()`, simply swap two positions and their colors are exchanged. This introduces small random changes to help escape local optimal.

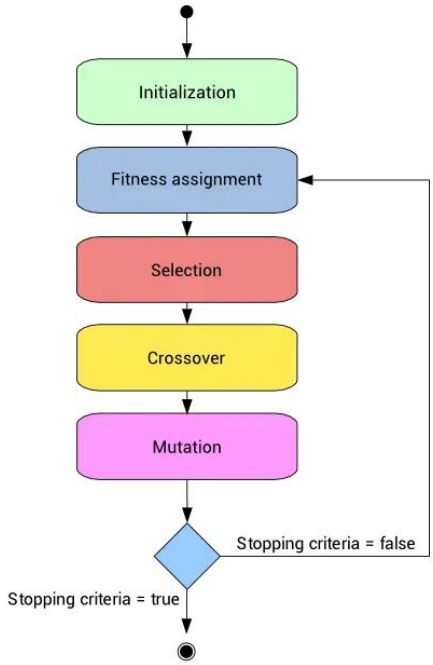


Figure 10: Genetic Algorithm

5. Results and Discussion

There are multiple tests being implemented. On hard tests where color points are either close to each other or at a location where it is very hard to connect, the heuristic methods are not as good as the GA method, the GA always can connect more color pairs than other methods, but GA takes longer time to complete the search for an optimal solution. On simple tests all methods performed equally. For instance, with a layout where each color pair is isolated, all algorithms connected 4-5 colors in a short time. In such cases the GA has no advantage, and often the solution has nearly the same length as greedy. This suggests that the GA's benefit is most clear on harder instances.

These results support several conclusions:

Effectiveness in Connection Maximization: The GA is more effective at maximizing connected colors. In the experiments it often found a solution connecting an additional color when simpler strategies could not. This aligns with the GA's fitness emphasis on connection counts. The GA can effectively search different color orders to overcome local traps.

Efficiency: When the GA connects more colors, it typically yields a higher total length because it is forced to route around existing obstacles. This is expected: the connection count is emphasized in fitness by a large weight, so the GA is willing to incur extra distance. In applications where minimizing path length is also critical, one could adjust the fitness weights.

Trade-offs in path length: Since the GA prioritizes connection count (C) heavily, it will willingly make detours and extra turns to secure additional connections. As a result, GA-derived routes typically have greater total length than those produced by length-focused heuristics when comparing solutions with the same connection count. This trade-off is predictable from the fitness formulation and highlights the need to recalibrate the weight when path efficiency is equally critical.

Adaptability through parameter tuning: The balance between completeness and compactness is directly adjustable by modifying the fitness weights or genetic operators. Increasing the penalty on length (L) or introducing a secondary selection pressure for shorter genotypes can steer the GA toward more efficient. This tunability makes the GA framework adaptable to a spectrum of real-world requirements, from time-sensitive robotics to PCB layout design.

Scalability and Robustness: While the current experiments focus on a 10x10 grid with five color pairs, the GA scales naturally to larger grids and more endpoints. Population-

based research retains robustness against increasing problem complexity, provided that mutation and crossover rates are appropriately adjusted.

Educational and Visualization Benefits: This interactive Pygame implementation not only facilitates performance evaluation but also serves as a valuable teaching tool. By visualizing how different strategies evolve or get trapped, students can gain intuitive insight into evolutionary search, constraint handling, and objective trade-offs.

A key observation is that user-order and priority can also differ in performance. The user-order strategy may perform worse than a well-chosen fixed priority. Which is acceptable sometimes since the early-chosen colors are more important for some scenarios.

Finally, the run-time can be different for each strategy: the GA took longer than the heuristics but still completed within a few seconds on these small grids. For educational projects this is acceptable. In a larger grid, one might optimize by caching paths or reducing population.

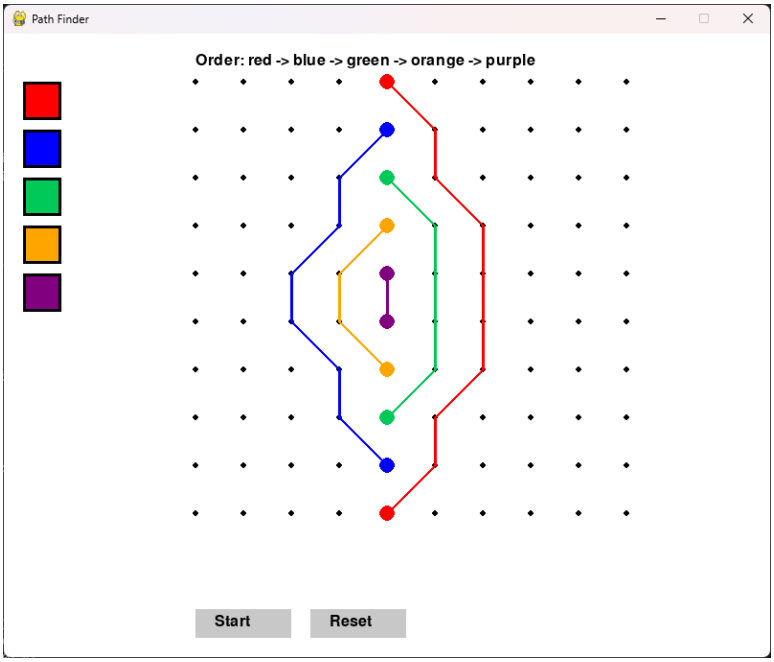


Figure 11: Path finding result

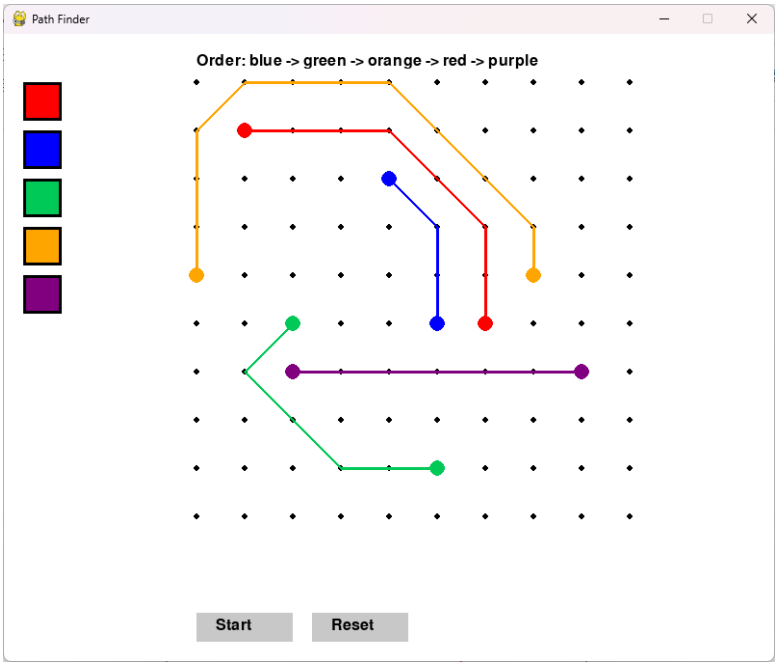


Figure 12: Path finding result 2

A comparative evaluation based on five trials per algorithm: User-Defined, Priority-Order, Greedy Search and Genetic Algorithm, reveals that the Genetic Algorithm incurs substantially greater execution time than its counterparts while achieving the highest average number of connected color pairs. Notably, it is the sole method to identify a five-color connection. In contrast, the remaining algorithms demonstrate rapid performance but are generally limited to three or four connected color pairs.

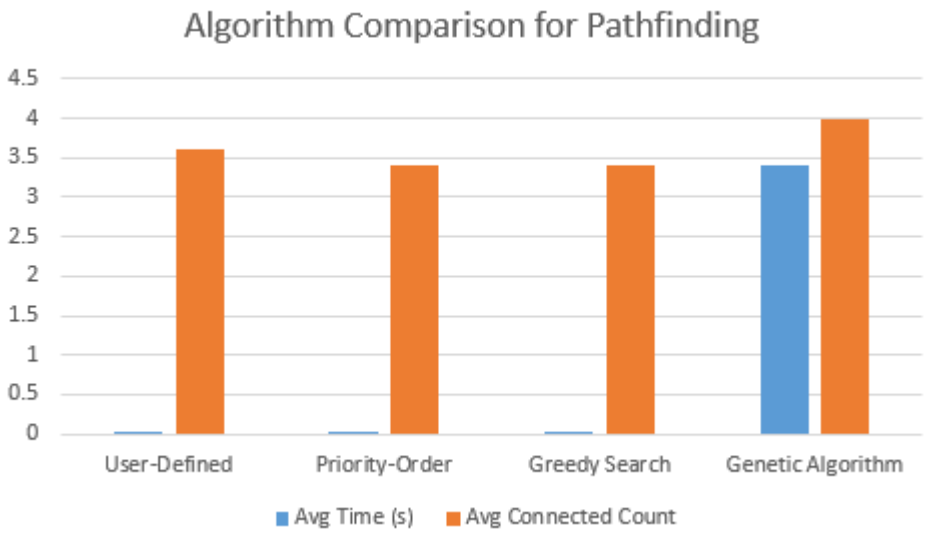


Figure 13: Algorithm Comparison for Pathfinding

Table 1: User-Defined Algorithm

User-Defined	Time (s)	Connected Count
No.1	0.00155	3
No.2	0.0014	4
No.3	0.00127	4
No.4	0.00116	3
No.5	0.0021	4

Table 2: Priority-Order Algorithm

Priority-Order	Time (s)	Connected Count
No.1	0.00136	3
No.2	0.00139	4
No.3	0.00131	3
No.4	0.00151	3
No.5	0.00112	4

Table 3: Greedy Search Algorithm

Greedy Search	Time (s)	Connected Count
No.1	0.00145	4
No.2	0.0012	3
No.3	0.00158	4
No.4	0.00123	2
No.5	0.00147	4

Table 4: Genetic Algorithm

Genetic Algorithm	Time (s)	Connected Count
No.1	3.85688	4
No.2	2.43208	3
No.3	4.09896	4
No.4	3.59162	4
No.5	3.03054	5

Table 5: Average Comparison of All Algorithms

Algorithm	Avg Time (s)	Avg Connected Count
User-Defined	0.001496	3.6
Priority-Order	0.001338	3.4
Greedy Search	0.001386	3.4
Genetic Algorithm	3.402016	4

6. Conclusion

A genetic algorithm has been designed, implemented and evaluated for a color-connection pathfinding game. The GA encodes solutions as color-orderings and optimizes for maximum connected pairs with minimal total length. In the Python-based experiments, the GA generally matched or outperformed simpler heuristics in terms of the number of colors connected. Confirming that evolutionary search can improve global solution quality [5]. Specifically, on challenging instances the GA connected more pairs than greedy or fixed-order methods, at the expected cost of longer paths. This demonstrates the GA's ability to overcome local-choice pitfalls by exploring many orderings.

The report emphasized the GA's design: permutation encoding, roulette selection, ordered crossover, swap mutation, and a compound fitness function.

The findings suggest that hybrid approaches might be fruitful: for instance, using the GA to determine a good initial ordering and then allowing a greedy refinement, or vice versa. In practice, one might also incorporate domain-specific heuristics or constraint programming to further prune the search [11]. Nonetheless, this project results clearly illustrate how a GA can solve a complex problem more effectively than naïve strategies.

7. Future Work

The grid currently measures 10×10; expanding it to 100×100 does increase connection possibilities but renders the interface overcrowded and difficult to use on smaller displays. To address this scalability problem while preserving readability, a zoom-in/zoom-out feature could be introduced, allowing users to magnify specific grid regions for precise placement of color points.

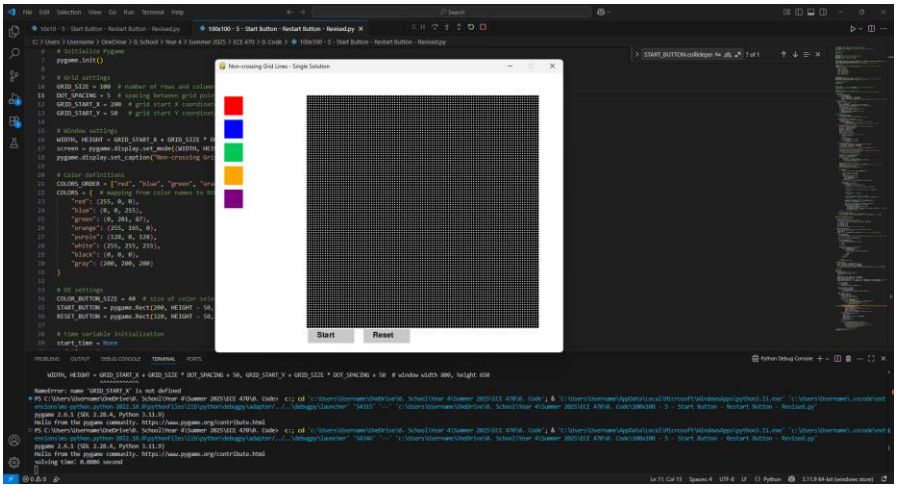


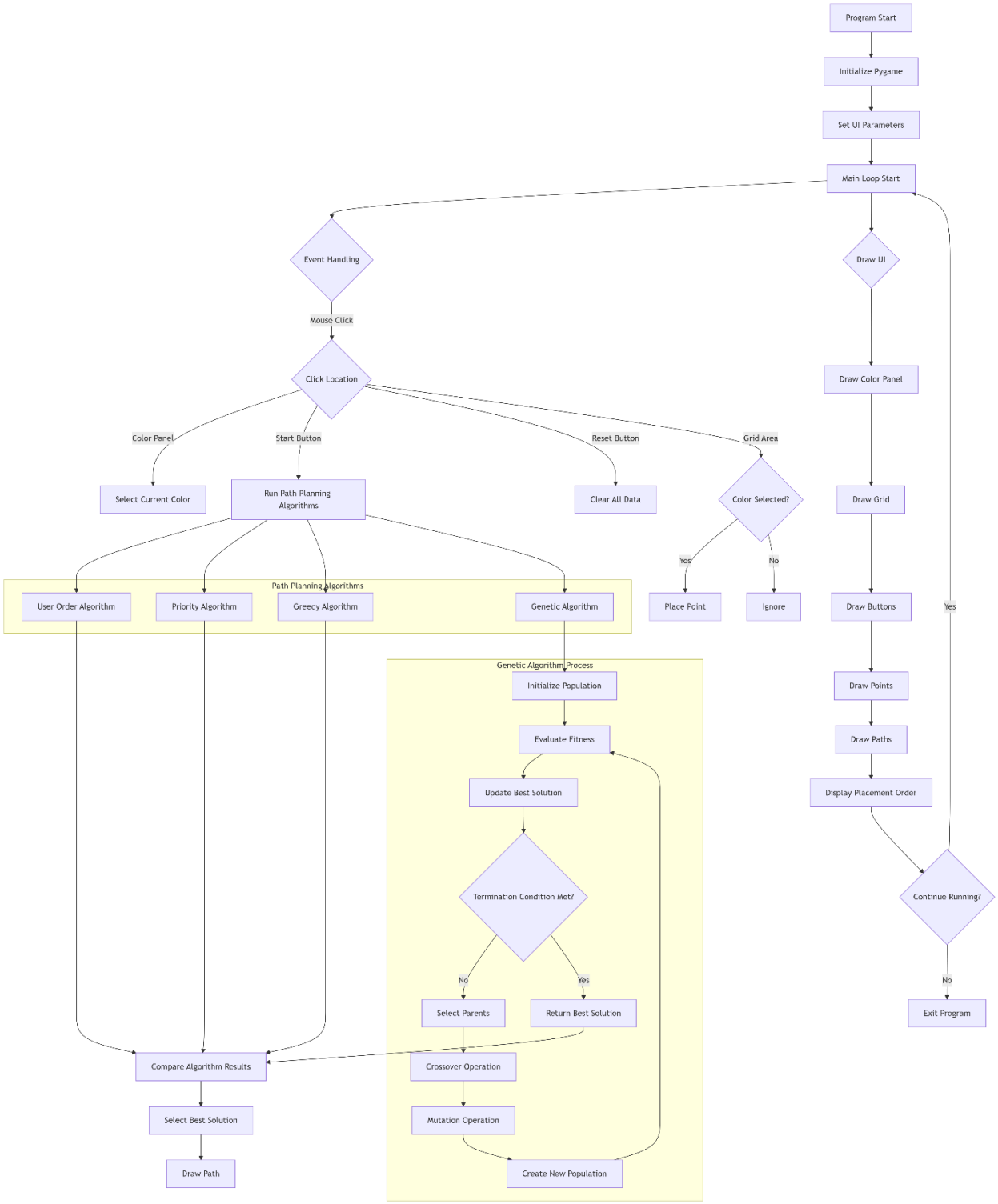
Figure 14: 100x100 Grid

Due to the absence of established datasets in this project that is typical in AI project like computer vision research, a customize database could be constructed by manually wiring dot configurations and systematically vary their positions across numerous trials. This tailored dataset enables the algorithm to identify optimal paths that surpass the capabilities of existing methods.

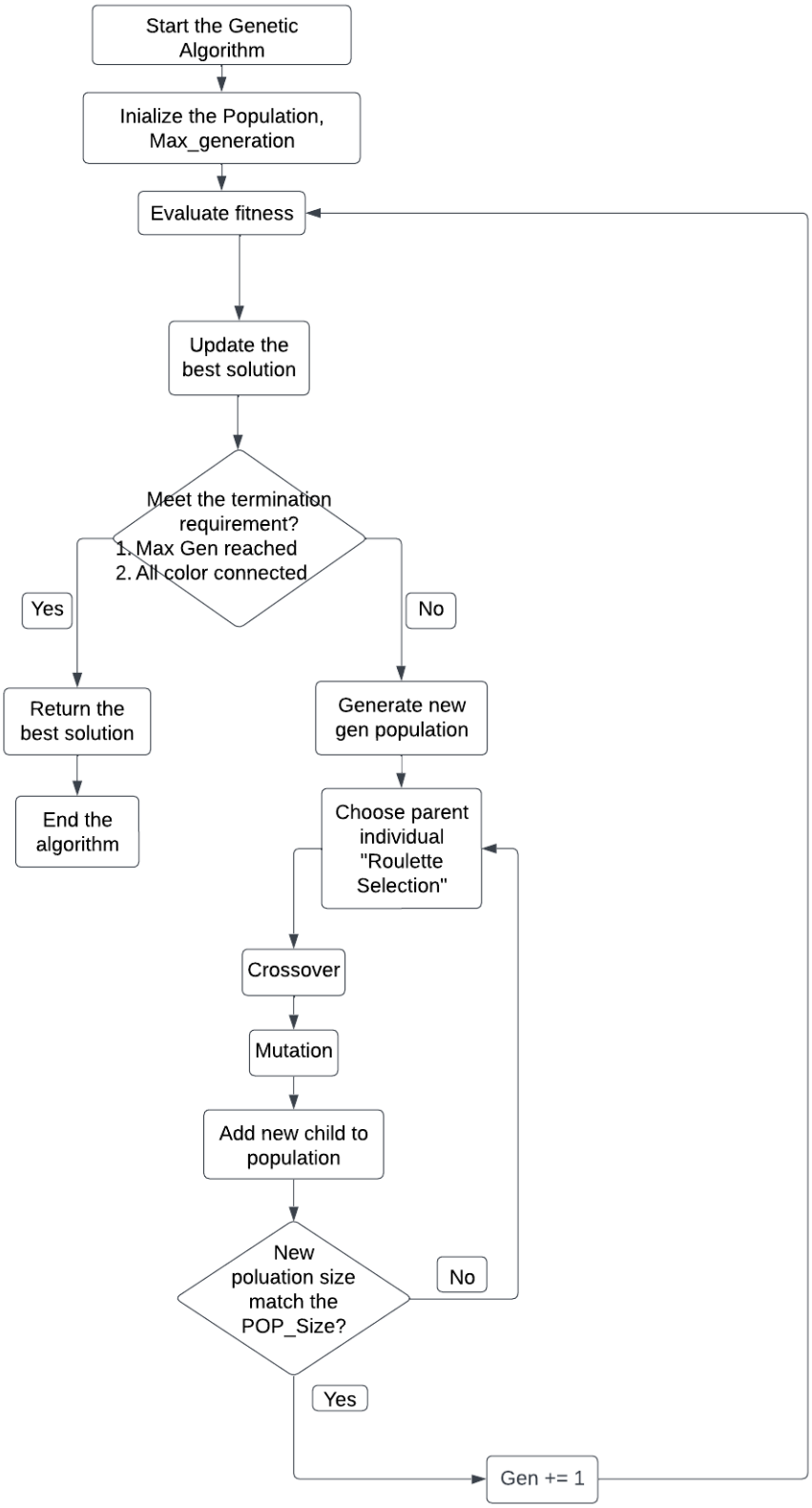
Other future enhancements could include integrating crossover at the path level, like combining partial solutions from two parents; experimenting with different fitness formulations or multi-objective optimization; or comparing with other metaheuristics. One could also study how performance scales with grid size and number of colors or incorporate real-time updating if the problem changes. Finally, a user study could evaluate whether this GA-based method actually produces “more satisfying” solutions in a game context.

Appendix:

Project Flow Chart



Genetic Algorithm Flow Chart



Project Poster

TRACE: PCB Pathfinding Project

Automated Grid Routing Using Genetic Algorithm & A* Pathfinding

ECE 470: Artificial Intelligence

Iain Graham | Alex Rockson | Kevin Zhou

What Problem Are We Solving

Printed circuit boards (PCBs) hold electronic components like resistors, capacitors, and chips, which are connected using copper traces. These traces link solder pads to form complete circuits.

Designers use computer-aided design (CAD) software to place components and draw traces on a PCB. As the number of connections increases, manually routing each trace becomes more complex and time-consuming. Careful planning is needed to avoid crossing traces and to ensure all connections are completed without causing electrical issues.

Project Overview

Our goal was to design a program that could automatically draw valid, non-crossing traces between pairs of points on a grid, just like on a real printed circuit board (PCB). We wanted to create a tool that not only performs this task using smart pathfinding algorithms, but also gives users a way to interact with and test the system themselves.

How We Built It

We created a custom Python program using the Pygame library to provide a graphical user interface (GUI). This interface allows users to select coloured endpoints, place them on a 10×10 grid, and then press a button to find a routing solution. The program explores multiple routing strategies, including a genetic algorithm that evolves optimal connection orders across generations, to compute non-crossing paths between solder pad pairs.

Design Constraints

To make the problem realistic and challenging, we applied several constraints to the trace routing:

- Traces must stay on the grid and connect only the placed endpoint pairs
- Traces must not cross over one another
- Turns are limited to 45 degrees, just like physical trace routing rules
- Once a trace is placed, it becomes an obstacle for all future routes.

These constraints simulate real-world design limitations in PCB layouts.

Key Python Functions

Several parts of the code were critical to making the system work:

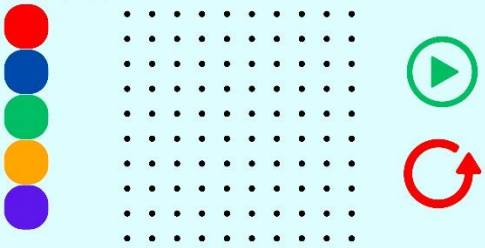
- Genetic Algorithm Engine evolves connection orders using selection, crossover, and mutation to discover optimal, non-intersecting paths
- A Pathfinding Core* finds individual paths between endpoint pairs while avoiding already placed traces and respecting 45° turns
- Fitness Evaluator prioritizes solutions with more connected pairs and shorter total trace length
- Obstacle Tracker ensures new paths avoid already occupied nodes
- Intersection Detector validates that no two traces overlap.

Together, these functions ensure that each connection is valid and respects the design rules.

Graphical User Interface

Our interface makes testing easy and visual. Users can:

- Select a colour from the sidebar
- Click two points on the grid to set a connection
- Selected colours will have a grey border
- Once two points of one colour are on the grid, the corresponding sidebar square will gain a black border
- Press the START button to generate a solution
- Use the RESET button to clear the board and try new configurations.

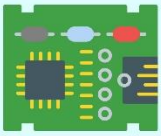


What We Achieved

We built a routing tool that automatically draws valid, non-overlapping connections between up to five colour-coded point pairs. It uses a genetic algorithm to explore different connection orders and A* pathfinding to efficiently route each pair. The system follows design rules like avoiding obstacles and limiting turn angles, and provides a simple visual interface for users to place points and view results instantly.

Future Iterations

This project lays the groundwork for more advanced PCB routing tools. Future versions could support larger grids, integrate real PCB layout features, optimize trace lengths, or run faster using parallel processing. With added complexity, this type of AI-assisted routing can be adapted to real-world electronics design and manufacturing.



Github Page

https://github.com/FengXue49/Uvic-ECE-470.git

References

[1] Chen J, Zhou Y, Liu Q, Zhang X. A Novel Global Routing Algorithm for Printed Circuit Boards Based on Triangular Grid. *Electronics*. 2023; 12(24):4942. https://doi.org/10.3390/electronics12244942

[2] Daniel, K., Nash, A., Koenig, S., & Felner, A. (2010). Theta*: Any-angle path planning on grids. *Journal of Artificial Intelligence Research*, *39*, 533-579.

[3] M. Daneshvaramoli, M. S. Kiarostami, S. K. Monfared, H. Karisani, A. Visuri, S. Hosio, D. Rahmati, and S. Gorgin, “A study on non-overlapping multi-agent pathfinding,” in *Advances in Information and Communication: FICC 2022*, K. Arai, Ed. Cham, Switzerland: Springer, 2022, vol. 439, pp. 1–18, doi: 10.1007/978-3-030-98015-3_1.

[4] D. Foead, A. Ghifari, M. B. Kusuma, N. Hanafiah and E Gunawan, “A systematic literature review of A* pathfinding,”

[5] Katoch S, Chauhan SS, Kumar V. A review on genetic algorithm: past, present, and future. Multimed Tools Appl. 2021;80(5):8091-8126. doi: 10.1007/s11042-020-10139-6. Epub 2020 Oct 31. PMID: 33162782; PMCID: PMC7599983.

[6] Rathinam, Vinoth & Ramasamy, Senthil Ganesh. (2019). Path Search Algorithms for VLSI Detailed Routing - A Survey. VIII. 355-363.

[7] Pygame Development Team, “pygame,” GitHub repository, https://github.com/pygame/pygame

[8] T. van Eyndhoven, "Solving Flow Free with SAT," 2015. [Online]. Available: https://torvaney.github.io/projects/flow-solver.html

[9] GeeksforGeeks, “Bresenham’s Line Generation Algorithm,” GeeksforGeeks, published Feb. ? 2024*, https://www.geeksforgeeks.org/bresenhams-line-generation-algorithm/

[10] O. Varaksin, “Convert world to screen coordinates and vice versa in WebGL,” Medium, Apr. 24, 2023. [Online]. Available: https://olegvaraksin.medium.com/convert-world-to-screen-coordinates-and-vice-versa-in-webgl-c1d3f2868086.

[11] N. Chandak, K. Chour, S. Rathinam, and R. Ravi, “Informed Steiner Trees: Sampling and Pruning for Multi-Goal Path Finding in High Dimensions,” https://www.contrib.andrew.cmu.edu/~ravi/ASE23.pdf