Autonomous Robot Navigation Using a Genetic Algorithm: Implementations in C and Python and Their Relevance to AGVs

Furqan Yousaf*

Department of Mechatronics and Control Engineering, University of Engineering and Technology, Lahore, Pakistan
* Corresponding author: 2021mc64@student.uet.edu.pk | ORCID: https://orcid.org/0009-0005-5151-9085

Abstract- A mobile robot must travel from a start cell to a goal cell without touching obstacles, ideally along a short and smooth route. This paper presents a genetic algorithm (GA) that solves this path-planning problem on grid maps and mazes, implemented twice: in C for square N × N obstacle grids and in Python for rectangular mazes of any size generated with the pyamaze library. A route is encoded as a short chromosome whose genes give the column (or row) the robot occupies in each row (or column), so a route across a 20 × 20 grid needs only 20 numbers. The fitness function multiplies a feasibility factor by normalized length and turn factors, which drives the population first to collision-free routes and then to short, smooth ones; rank selection, block crossover and single-gene mutation breed each new generation. The C program solved the original 8 × 8 map in 4 to 8 generations in all seven runs, a random 20 × 20 map with 21.5% obstacles in 59 to 1,947 generations in all twelve runs, and a 40 × 40 map in the original project. The Python program solved 18 × 18 and 20 × 26 mazes in 252 and 3,953 generations. Unchanged, the C program also routed a vehicle through a warehouse-style layout of racks, pallets and a charger in all three runs. The paper relates these results to automated guided vehicles (AGVs) and autonomous mobile robots, where GAs are studied for route optimization and multi-vehicle planning, sets out the limits of the method against graph-search planners such as A*, and lists concrete improvements to both programs.

Index Terms- Automated guided vehicles, genetic algorithm, mobile robot navigation, obstacle avoidance, path planning

I. INTRODUCTION

Mobile robots suit jobs that are dangerous or tedious for people, such as bomb disposal, handling hazardous material, and carrying goods around factories and warehouses. Autonomous navigation lets such a robot travel from a starting point to a target point without human help. The area may contain obstacles, so the robot needs a route that lies entirely in free space; such a route is called feasible. Path planning is the task of finding a feasible route and then improving it against further criteria such as length and smoothness [1], [2].

The same problem lies at the heart of industrial intralogistics. Automated guided vehicles (AGVs) and autonomous mobile robots (AMRs) now move material in manufacturing plants, warehouses, cross-docks, terminals and hospitals [3]; one large Kiva warehouse installation, for example, may use 500 or more drive units [4]. Each of these vehicles has to reach its destination without running into racks, pallets, people or other vehicles.

Genetic algorithms (GAs) are a population-based search method modelled on natural selection [5]–[7]. Instead of exploring a map cell by cell, a GA keeps many complete candidate routes, scores them, and breeds better routes from the best ones. This paper describes a GA navigation program written for the Computer Programming (MCT-242) course at UET Lahore, which follows the row-wise route encoding and the feasibility, length and turn factors described by Manikas and co-workers [8], [9], and its later extension to a second language and to larger maps. The objectives were to:

1) encode a robot route as a short chromosome and design a fitness function that rewards feasibility, short length and few turns;
2) implement the GA in C for square obstacle grids and in Python for rectangular mazes;
3) measure how quickly each version finds a collision-free route on maps from 8 × 8 to 40 × 40; and
4) relate the method to AGVs and AMRs, including where a GA is, and is not, the right tool.

Section II reviews related work, Section III states the problem and Section IV describes the GA. Sections V and VI cover the C and Python implementations, Section VII reports the results and Section VIII discusses AGVs and other mobile robots. Section IX lists limitations and improvements, and Section X concludes the paper.

II. Related Work

A. Path-Planning Methods

Global planners for mobile robots fall broadly into graph-search methods, sampling-based methods and nature-inspired optimizers. A* search adds a heuristic estimate of the remaining cost to the cost already travelled, and finds a minimum-cost path on a graph when that estimate never overestimates [10]. A review of the literature found GAs, particle swarm optimization, artificial potential fields and ant colony optimization to be the most used approaches to mobile-robot path planning [1]. A later survey concluded that, for a given map, A* is a strong-performing option, and noted that bio-inspired algorithms do not give excellent results in real-time path planning [2].

B. Genetic Algorithms for Robot Paths

Holland introduced GAs as a model of adaptation [5], and Goldberg and Mitchell describe the selection, crossover and mutation operators used here, including rank selection [6], [7]. For robot navigation, the Tulsa group represented a route through a map of N rows by a genotype of N genes, one column per row, and scored it by feasibility, length and number of turns [8], [9]. Hu and Yang added problem-specific operators that use knowledge of the map, and showed their GA planning in both static and dynamic environments [11].

C. Genetic Algorithms for AGVs

GAs have been applied widely to AGV problems. Han et al. planned paths for several AGVs at once with an improved GA and a three-exchange crossover, shortening both the total distance and the longest single-vehicle distance [12]. Li et al. treated multi-AGV path planning in a warehouse, an NP-hard problem, with a GA [13]. Cai et al. designed a GA with a turning-cost function for two AGVs carrying one load together; in simulation and real-vehicle tests it planned safe, collision-free paths and outperformed a traditional GA and the A* and Dijkstra planners used in ROS, including a 30.1% shorter path than Dijkstra [14]. In AGV task scheduling, GAs are the most common heuristic, and Liu et al. used adaptive GAs to schedule sorting AGVs while accounting for battery charging [15].

Compared with these studies, the present work is an introductory implementation: a plain GA with a compact staircase encoding, written in two languages and tested on maps up to 40 × 40. Its value lies in showing every step of the method in short, readable code, and in measuring how that code behaves.

III. Problem Formulation

The arena is a grid of cells. In the C version it is an N × N matrix of 0 (free) and 1 (obstacle); the robot starts at the top-left cell (0, 0) and the goal is the bottom-right cell (N − 1, N − 1). In the Python version the arena is a maze of R rows and C columns whose walls lie between cells; the robot starts at cell (1, 1) and the goal is cell (R, C). The robot moves one cell at a time, horizontally or vertically. Three quantities judge a route:

Obstacle strikes, S: the number of route cells that lie on an obstacle (C), or the number of steps that pass through a closed wall (Python). A feasible route has S = 0.

Path length, L: the number of moves from start to goal.

Turns, T: the number of changes of direction along the route.

The planner must find a route with S = 0 and, among such routes, prefer small L and T. Short, smooth routes save travel time and reduce wear on a real vehicle [1].

IV. Genetic Algorithm Design

A. Chromosome Encoding

A chromosome is a list of genes, one for each row of the map (Fig. 1). In the row-wise encoding, gene i is the column the robot occupies in row i: the route sweeps along row i from the column given by gene i to the column given by gene i + 1, then steps down into row i + 1. In the column-wise encoding the roles of rows and columns are swapped. Decoding always yields a staircase route made only of horizontal and vertical moves, and a route across a 40 × 40 grid needs only 40 numbers. The first gene is fixed to the start and the last to the goal, so only the middle genes evolve. In C, each chromosome carries a direction bit (0 for row-wise, 1 for column-wise); the Python program keeps two separate populations, one of each kind, and stops when either finds a route.

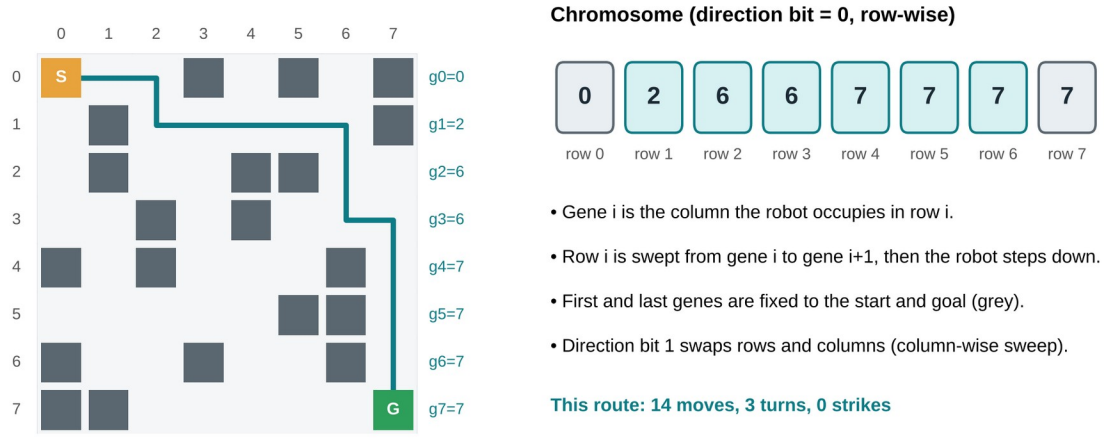


Fig. 1. Chromosome encoding on the original 8 × 8 map. The row-wise chromosome 0, 2, 6, 6, 7, 7, 7, 7, found by the C program, decodes into a 14-move route with three turns and no strikes.

B. Fitness Function

Each quantity is normalized against the rest of the population, so that 1 is ideal and 0 is the worst [8], [9]. With S, L and T for one chromosome and their maxima and minima over the population, the feasibility, length and turn factors are

$$f_S = 1 - (S - S_{min}) / (S_{max} - S_{min}), \quad \text{with } S_{min} \text{ taken as } 0 \tag{1}$$

$$f_L = 1 - (L - L_{min}) / (L_{max} - L_{min}) \tag{2}$$

$$f_T = 1 - (T - T_{min}) / (T_{max} - T_{min}) \tag{3}$$

and the C program combines them, with weights $w_S = 3$ and $w_L = w_T = 2$, as

$$F_C = 100\, w_S\, f_S\, (w_L\, f_L + w_T\, f_T) / (w_L + w_T), \quad 0 \le F_C \le 300 \tag{4}$$

while the Python program uses

$$F_P = 100\, f_S\, (3\, f_L + 1.5\, f_T) / 3, \quad 0 \le F_P \le 150 \tag{5}$$

Because f_S multiplies the whole score, any strike pulls the fitness down, so feasibility dominates length and turns. The normalization is relative, however: a route with one strike can still rank first when the rest of the population has many, which matters when the search is nearly finished (Section VII-B).

C. Selection, Crossover and Mutation

After scoring, the population is sorted so that the fittest chromosomes come first, and the best half act as parents (rank selection [7]). Each pair of parents produces two children, which overwrite the weaker half of the population. Crossover swaps blocks of genes between the two parents at a cut point: the C program always cuts at the middle of the chromosome, while the Python program picks a random cut point each generation. Mutation then overwrites one random middle gene with a random value, which keeps the search exploring; the C program mutates every chromosome, and the Python program protects the top quarter of the population. After mutation the first and last genes are restored, so every route still starts at the start and ends at the goal. Fig. 2 shows the cycle.

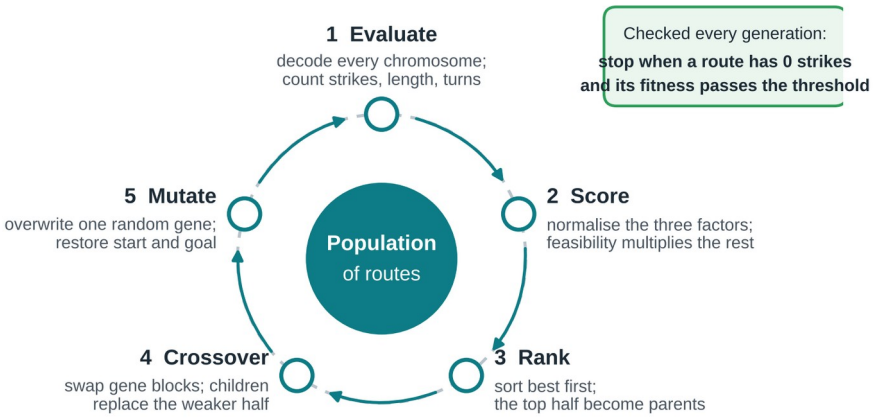


Fig. 2. One generation of the GA. The cycle repeats until the stop rule is met or the generation limit is reached.

D. Termination

The C program stops when a chromosome has zero strikes and a fitness of at least 265 (of 300), or after 25,000 generations. The Python program stops when its best chromosome has zero strikes and a fitness above 110 (of 150), or after 30,000 generations.

V. Implementation in C

The C program, about 450 lines, stores the arena as an N × N matrix whose size is fixed at compile time through GRIDSIZE, so the grid must have the same number of rows and columns: 8 × 8, 20 × 20, 30 × 30, 40 × 40 and so on. The population holds 1,000 chromosomes. Table I lists the functions. One generation decodes every chromosome, counts its strikes, turns and length, computes the fitness, bubble-sorts the population best first, applies crossover and mutation, and then evaluates and sorts again before checking the stop rule. The solution is printed in the terminal with block characters for obstacles and dots for the route (Fig. 6).

TABLE I
Functions of the C Program

Function (as named in the source)	**Role**
random_population	Creates 1,000 random chromosomes with fixed first and last genes and a random direction bit.
path_plannning	Decodes one chromosome into the list of cells of its route.
parameters_of_fitness	Counts the strikes, turns and path length of every chromosome.
fitness_function	Normalizes the three factors and computes the fitness, (1)–(4).
buble_sorting	Sorts chromosomes and direction bits by fitness, best first.
cross_over	Builds two children from each pair of parents by swapping halves.
mutation	Overwrites one random middle gene of every chromosome.
Display_Solution	Prints the grid with the obstacles and the route.

VI. Implementation in Python

The Python program builds an R × C maze with the pyamaze library [16], with a loop percentage of 65 so that the maze has several possible routes. Because the maze is described by walls between cells, a strike is a step of the route that crosses a closed wall, checked against the east, west, north and south wall flags of each cell. The numbers of rows and columns may differ, so a 20 × 26 maze runs without any change to the code. The program keeps 52 row-wise and 52 column-wise chromosomes. Two details shape its behaviour: the populations are sorted by strike count with a tuple sort, and the top quarter is never mutated, so the best strike count can only improve from one generation to the next. Table II compares the two versions.

TABLE II
THE TWO IMPLEMENTATIONS COMPARED

Aspect	C program	Python program
Arena	Square N × N grid of obstacle cells	Any R × C maze with walls (pyamaze)
Population	1,000 chromosomes, each with a direction bit	52 row-wise and 52 column-wise chromosomes
What counts as a strike	A route cell on an obstacle	A route step through a closed wall
Fitness range	0 to 300, (4)	0 to 150, (5)
Sorting	Bubble sort by fitness	Tuple sort by strike count
Crossover cut	Fixed at the middle	Random each generation
Mutation	Every chromosome, one gene	All but the top 25%
Stop rule	Fitness ≥ 265 and 0 strikes	Fitness > 110 and 0 strikes
Generation limit	25,000	30,000

VII. RESULTS

All runs used the student programs with their search logic unchanged; only logging was added, and the C program had its unused Windows header removed so that it would compile on Linux. Times were measured on one core of an Intel Xeon cloud machine (GCC 13.3 with -O2; Python 3.13) and will differ on other computers. Path length is counted in moves, and generations are counted from 1. The C program reseeds its random-number generator from the clock, so each run differs; the Python runs seed the generator once, which makes them repeatable. Table III summarizes every run.

A. Original 8 × 8 Map

On the 8 × 8 map of the original project, all seven runs found a collision-free route in 4 to 8 generations (median 5), each in a few hundredths of a second. Every solution had 14 moves, the shortest possible on this grid, three turns and a fitness of 270. Fig. 3 follows one run: the top-ranked route of generation 1 crossed two obstacles, that of generation 3 crossed one, and in generation 5 a route with no strikes appeared. Fig. 4, from the original 2022 project report, shows the same trend over a longer run: path length fell from 50 to about 30 cells and turns from 10 to 5 as the strikes fell.

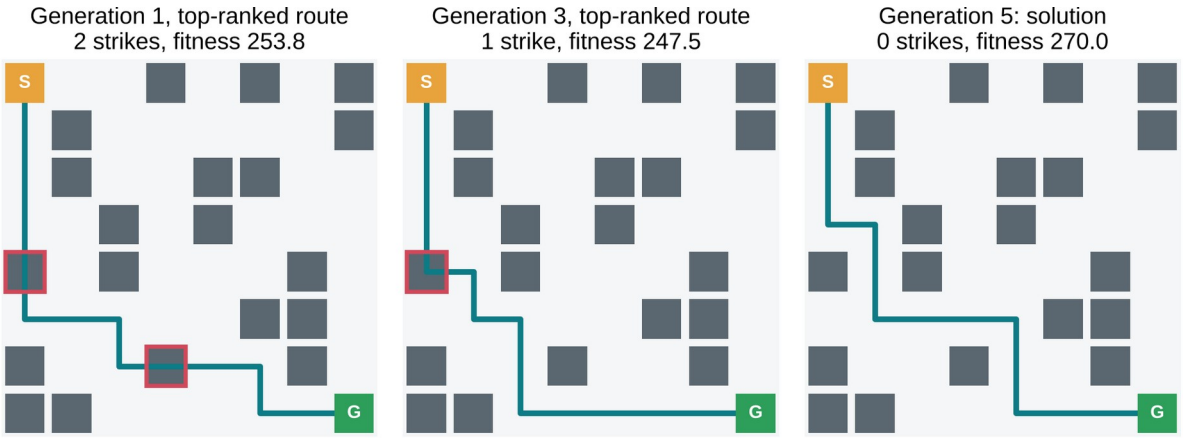


Fig. 3. One C run on the 8 × 8 map. Red outlines mark obstacle strikes.

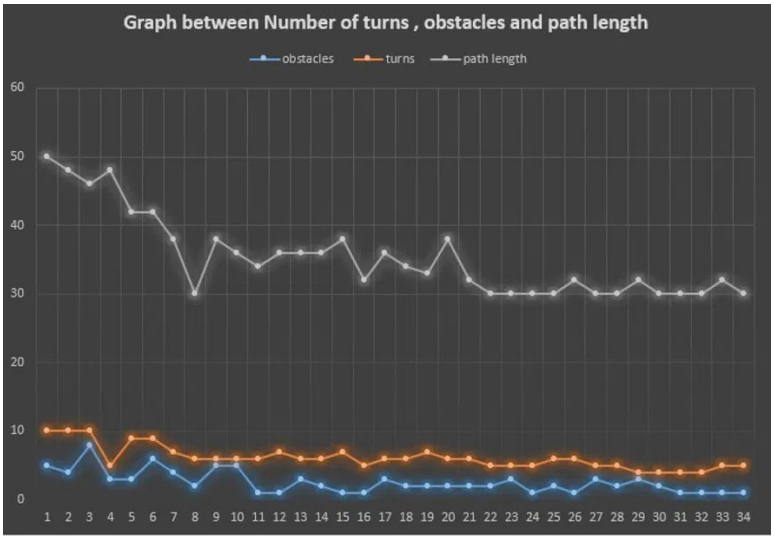


Fig. 4. Obstacle strikes, turns and path length of the best route at 34 sampled points of a run, reproduced from the original project report (2022).

B. Random 20 × 20 Map

For a larger test, a 20 × 20 map was generated at random with 86 obstacles (21.5%) and checked to contain at least one route that the encoding can represent. In twelve runs the C program always found a collision-free route, after 59 to 1,947 generations (median 857) and 0.35 to 11.1 s. The solutions had 38 to 44 moves, where 38 is the shortest possible, and seven to nine turns. Fig. 5 shows one run. The fewest strikes in the population fell from 13 to 1 by generation 43, and a zero-strike route first appeared in generation 95; its fitness of 233 was below the threshold of 265, so the search went on until generation 688, when a zero-strike route with 44 moves and a fitness of 280.5 was found. The stop rule therefore trades search time for route quality.

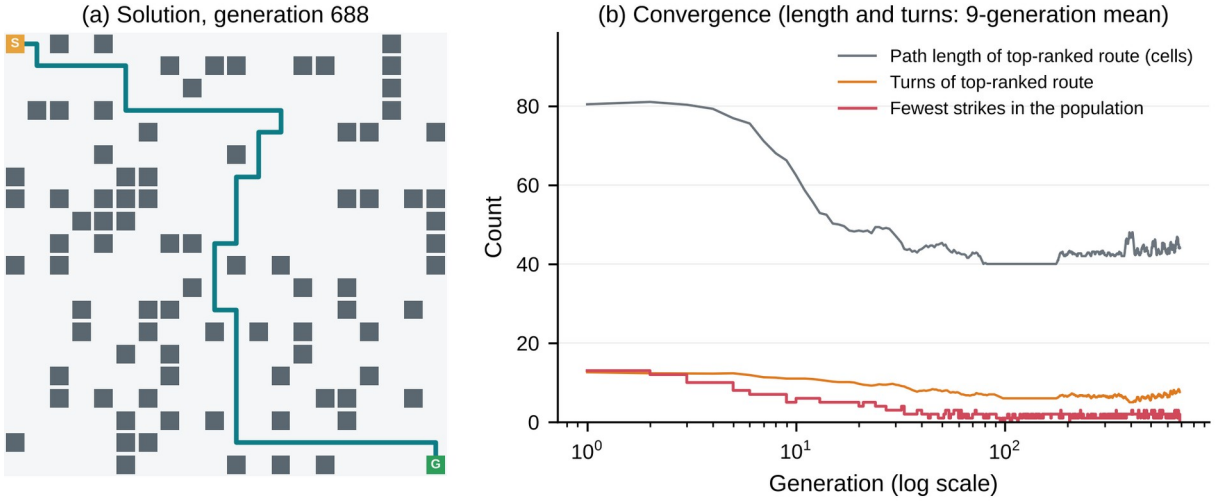


Fig. 5. C program on the random 20 × 20 map: (a) the route found in generation 688; (b) the fewest strikes in the population, and the length and turns of the top-ranked route, in every generation.

The length and turns of the top-ranked route jump about late in the run (Fig. 5b) because the C program mutates every chromosome, including the best one, and because fitness is relative: a short route with one strike can outrank a longer feasible one. A second random 20 × 20 map, on which only row-wise routes exist, was not solved: after 25,000 generations (143 s) the top-ranked route had never had fewer than one strike.

C. 40 × 40 Map

In the original project the same C program solved a 40 × 40 map (Fig. 6). It reported a route with no strikes and the maximum fitness of 300 at loop index 18,851, that is, in generation 18,852.

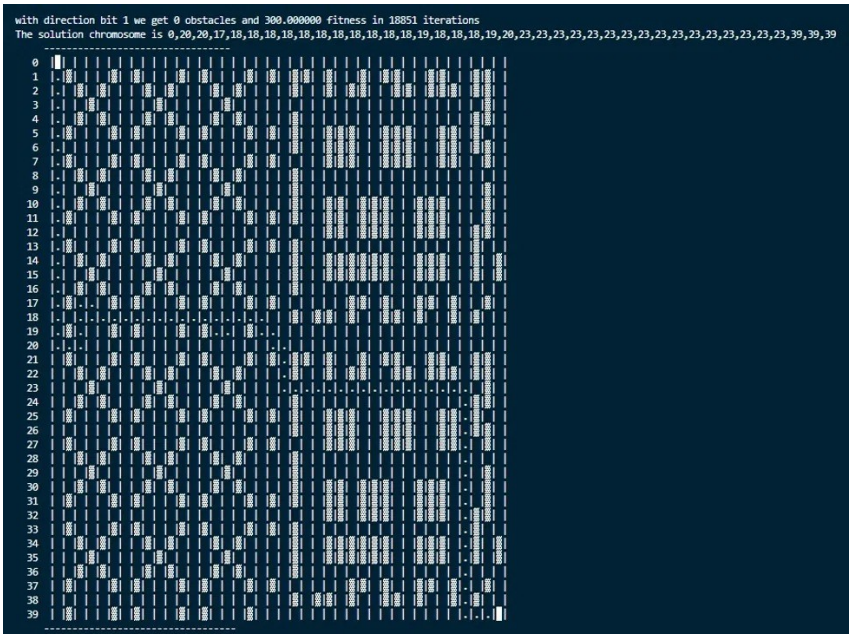


Fig. 6. Terminal output of the C program for the 40 × 40 map, from the original project; dots mark the route.

D. Python Mazes

With random seed 1, the Python program solved the 18 × 18 maze in 252 generations (1.1 s) and the 20 × 26 maze in 3,953 generations (19.6 s), both through its column-wise population (Fig. 7). The 18 × 18 route had 42 moves and 14 turns, with a fitness of 113.2 of 150; the 20 × 26 route had 58 moves and 15 turns and reached the maximum fitness of 150. Fig. 8 shows the strike count on the 20 × 26 maze: it fell from 36 to 2 by generation 111, held at 1 from generation 235 to 3,951, and reached 0 in generation 3,952. The last strike took most of the time, as is typical of a GA: early progress is fast and the final refinement is slow. Other seeds give other mazes and change the effort: of four more 18 × 18 mazes, three were solved in 101 to 6,578 generations and one was not solved within 30,000 generations.

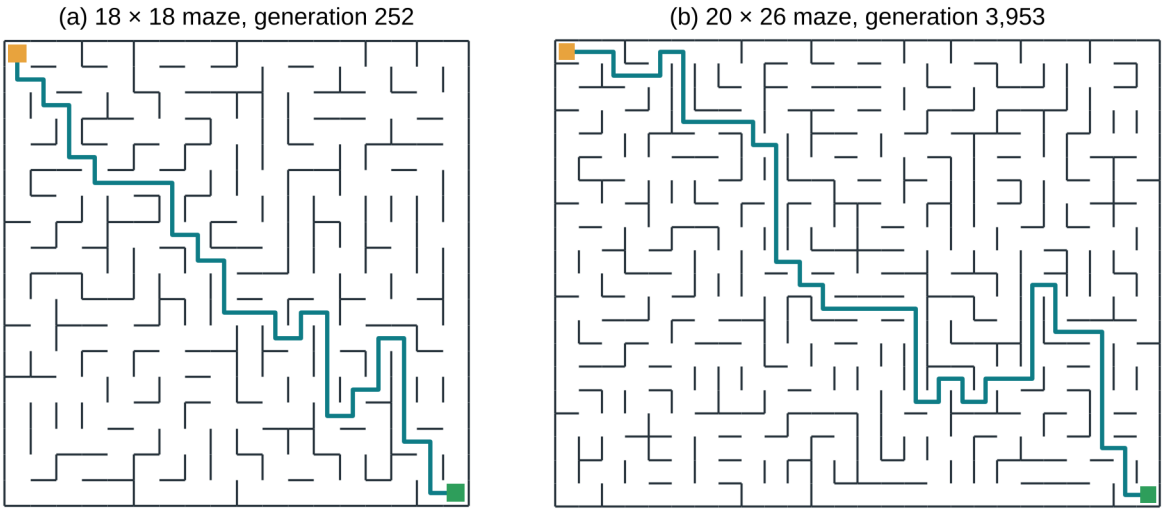


Fig. 7. Routes found by the Python program (seed 1): (a) 18 × 18 maze; (b) 20 × 26 maze.

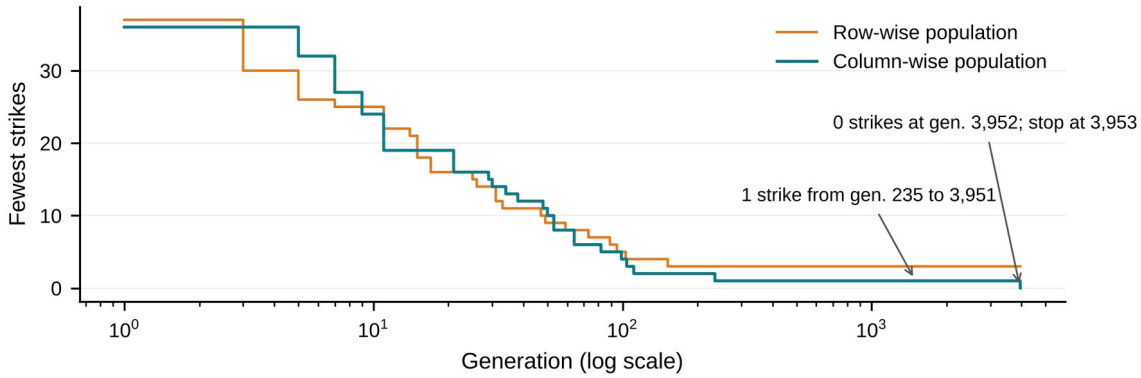


Fig. 8. Fewest strikes in each population per generation on the 20 × 26 maze.

TABLE III
SUMMARY OF RUNS

Program	Arena	Solved	Generations	Time (s)	Moves	Turns	Fitness
C	8 × 8, original map	7 / 7	4–8 (median 5)	< 0.05	14	3	270
C	20 × 20, random, 21.5% obstacles	12 / 12	59–1,947 (median 857)	0.35–11.1	38–44	7–9	266–288
C	20 × 20, random, row-wise routes only	0 / 1	stopped at 25,000	143	—	—	—
C	20 × 20, warehouse layout	3 / 3	1,385–1,567	8.1–10.1	42–48	5–6	267–270
C	40 × 40, original project	1 / 1	18,852	not recorded	—	—	300
Python	18 × 18 maze, seed 1	1 / 1	252	1.1	42	14	113.2
Python	18 × 18 mazes, seeds 2–5	3 / 4	101–6,578	0.4–24.6	42–48	10–13	138–150
Python	20 × 26 maze, seed 1	1 / 1	3,953	19.6	58	15	150
Python	24 × 30 maze, seed 1	0 / 1	stopped at 30,000	178	—	—	—

Fitness is out of 300 for C and 150 for Python. Moves, turns and fitness are those of the reported routes; ranges cover all solved runs. Unsolved runs are counted in the Solved column.

VIII. Relevance to AGVs and Other Mobile Robots

A. From a Grid to a Warehouse Floor

To see the method in an intralogistics setting, the unchanged C program was run on a 20 × 20 warehouse-style layout: four double rows of storage racks with a cross-aisle, plus pallets, a parked cart and a charging point in the aisles (Fig. 9). The start cell stands for a dock and the goal for a pick-and-drop station. In all three runs the GA found a collision-free route, after 1,385 to 1,567 generations (8 to 10 s), with 42 to 48 moves and five or six turns. Table IV shows how the parts of this project map onto an AGV or AMR system.

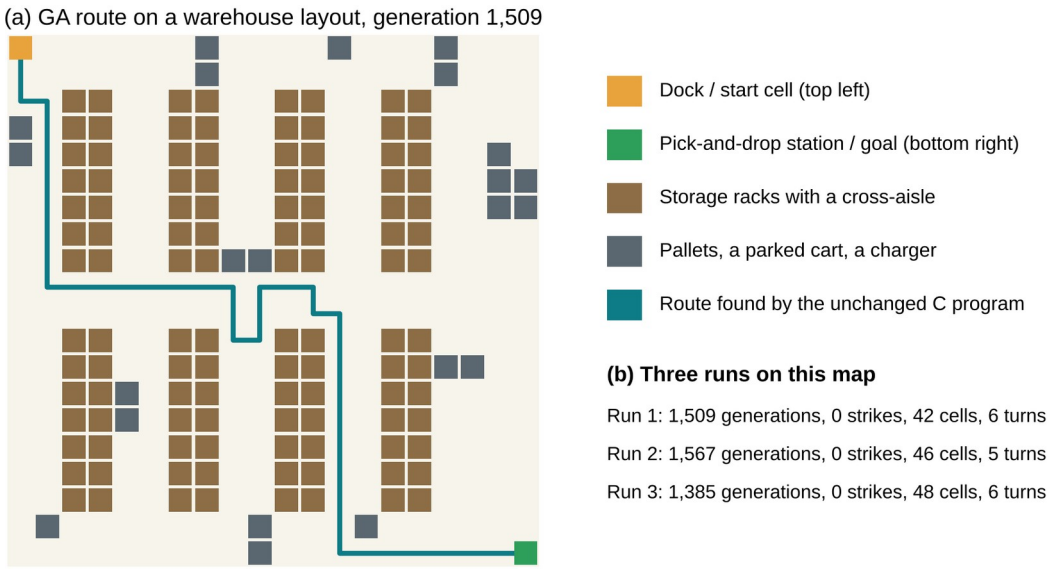


Fig. 9. The unchanged C program on a warehouse-style layout: (a) route of run 1; (b) the three runs.

TABLE IV
FROM THIS PROJECT TO AN AGV OR AMR SYSTEM

In this project	In an AGV or AMR system
Grid of free and blocked cells	Map of aisles, racks, stations and keep-out zones
Obstacle strike	Collision with a rack, pallet, person or another vehicle
Path length	Travel time and battery energy
Turns	Slowing down to turn; smoother, faster motion with fewer turns
Fitness weights	The operator's priorities: safety first, then time and smoothness
Population of routes	Several good alternative routes kept at once
Stop rule	The point at which the planner hands a route to the vehicle's controller

B. Where a GA Helps

A GA needs no gradient and no special structure in the problem: it only needs a way to score a route. That makes it easy to combine several objectives, such as length, smoothness and safety distance [14] or the total and longest distances of a whole fleet [12], and to plan for many vehicles and tasks together [13], [15]. Because a GA keeps a population, it also offers several good routes rather than one, which is useful when an aisle is blocked.

C. Where It Does Not

A GA is stochastic and gives no guarantee of finding the best route, or any route, within a fixed time; the unsolved 20 × 20 map and 18 × 18 maze in Section VII show this. For one vehicle on a known grid, A* finds a shortest path directly [10], and surveys rate bio-inspired algorithms as weak for real-time replanning [2]. In industry, AGV systems almost all operate under a central controller that takes the scheduling, routing and dispatching decisions for the whole fleet [3], [17]. A realistic role for a GA is therefore offline or global optimization, such as laying out routes, assigning tasks and tuning schedules, alongside a fast graph-search or local planner that reacts to people and vehicles in real time. A GA also does not learn from data: it searches each map afresh, so a new layout simply means a new run.

IX. LIMITATIONS AND IMPROVEMENTS

Division by zero. Both programs divide by the difference between the maximum and minimum when normalizing length and turns, (2)–(3). If every chromosome has the same value this divides by zero; a guard that returns 1 in that case removes the problem.

Sorting cost (C). Bubble-sorting 1,000 chromosomes costs about a million comparisons, twice per generation. Replacing it with qsort would speed the program up considerably.

Fitness type (C). The swap inside the bubble sort stores the fitness in an int, which truncates it. The fitness is recomputed afterwards, so results are not affected, but a float is the correct type.

Random seed (C). srand(time(0)) is called inside the mutation function every generation, so all generations within the same second reuse the same random numbers. Seeding once at the start gives more varied mutations.

Array bound (C). In its last step the decoding loop reads one element past the end of a chromosome. For every chromosome but the last this is the next chromosome's first gene, which is 0; for the last it lies outside the array. The runs here added one spare zero row so that the read behaves the same for all chromosomes; stopping the loop one step earlier is the proper fix.

Stop check (Python). The stop rule tests the strike count and fitness computed before the final sort, while the route drawn is the first one after it. In the runs reported here the drawn routes were checked and had no strikes, but the test should use the sorted lists.

Encoding. Routes are built from row or column sweeps, so a map whose only solution needs the robot to move back up or back to the left cannot be solved, and convergence depends strongly on the map: the 24 × 30 Python maze with seed 1 stalled at two strikes and was not solved within 30,000 generations (Table III).

Next steps. Add elitism to the C program so that its best fitness never drops, try an adaptive mutation rate, allow variable-length or waypoint chromosomes so that any route can be encoded, add moving obstacles and several robots, and compare every result against an A* baseline.

X. CONCLUSION

A genetic algorithm can find feasible, short and smooth routes for a mobile robot without being told how to search. Encoding a route as a list of row or column positions keeps the chromosomes short, and a fitness function that multiplies feasibility by length and turn factors steers the population toward collision-free routes first and efficient routes second. The C version runs quickly on square grids up to 40 × 40, and the Python version extends the method to rectangular mazes. The same unchanged program routes a vehicle through a warehouse-style layout, which connects this classroom project to the AGVs and AMRs of modern intralogistics, where GAs are used for route optimization and fleet planning alongside faster graph-search planners.

Acknowledgment

The author thanks his course instructor, Sir Shujaat, who supervised the original Computer Programming (MCT-242) project, and the Department of Mechatronics and Control Engineering, University of Engineering and Technology, Lahore, for its support.

Data Availability

The C and Python source code, the maps and the run logs used in this paper are available from the author on reasonable request.

Conflict of Interest

The author declares no conflict of interest.

References

[1] H.-y. Zhang, W.-m. Lin, and A.-x. Chen, "Path planning for the mobile robot: A review," *Symmetry*, vol. 10, no. 10, Art. no. 450, 2018, doi: 10.3390/sym10100450.

[2] K. Karur, N. Sharma, C. Dharmatti, and J. E. Siegel, "A survey of path planning algorithms for mobile robots," *Vehicles*, vol. 3, no. 3, pp. 448–468, 2021, doi: 10.3390/vehicles3030027.

[3] G. Fragapane, R. de Koster, F. Sgarbossa, and J. O. Strandhagen, "Planning and control of autonomous mobile robots for intralogistics: Literature review and research agenda," *Eur. J. Oper. Res.*, vol. 294, no. 2, pp. 405–426, 2021, doi: 10.1016/j.ejor.2021.01.019.

[4] P. R. Wurman, R. D'Andrea, and M. Mountz, "Coordinating hundreds of cooperative, autonomous vehicles in warehouses," *AI Mag.*, vol. 29, no. 1, pp. 9–20, 2008, doi: 10.1609/aimag.v29i1.2082.

[5] J. H. Holland, *Adaptation in Natural and Artificial Systems*. Ann Arbor, MI, USA: Univ. of Michigan Press, 1975.

[6] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading, MA, USA: Addison-Wesley, 1989.

[7] M. Mitchell, *An Introduction to Genetic Algorithms*. Cambridge, MA, USA: MIT Press, 1996.

[8] T. W. Manikas, K. Ashenayi, and R. L. Wainwright, "Genetic algorithms for autonomous robot navigation," *IEEE Instrum. Meas. Mag.*, vol. 10, no. 6, pp. 26–31, Dec. 2007, doi: 10.1109/MIM.2007.4428579.

[9] A. Hermanu, T. W. Manikas, K. Ashenayi, and R. L. Wainwright, "Autonomous robot navigation using a genetic algorithm with an efficient genotype structure," in *Intelligent Engineering Systems Through Artificial Neural Networks*, vol. 14 (Proc. ANNIE 2004). New York, NY, USA: ASME Press, 2004.

[10] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Trans. Syst. Sci. Cybern.*, vol. 4, no. 2, pp. 100–107, Jul. 1968, doi: 10.1109/TSSC.1968.300136.

[11] Y. Hu and S. X. Yang, "A knowledge based genetic algorithm for path planning of a mobile robot," in *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, New Orleans, LA, USA, 2004, pp. 4350–4355, doi: 10.1109/ROBOT.2004.1302402.

[12] Z. Han, D. Wang, F. Liu, and Z. Zhao, "Multi-AGV path planning with double-path constraints by using an improved genetic algorithm," *PLoS ONE*, vol. 12, no. 7, Art. no. e0181747, 2017, doi: 10.1371/journal.pone.0181747.

[13] C. Li, C. Cao, and Y. Gao, "Path planning for multiple AGV systems using genetic algorithm in warehouse," in *Proc. 3rd Int. Conf. Commun., Inf. Manage. Netw. Secur. (CIMNS)*, Atlantis Press, 2018, pp. 106–109, doi: 10.2991/cimns-18.2018.24.

[14] Y. Cai, H. Liu, M. Li, and F. Ren, "A method of dual-AGV-ganged path planning based on the genetic algorithm," *Appl. Sci.*, vol. 14, no. 17, Art. no. 7482, 2024, doi: 10.3390/app14177482.

[15] Y. Liu, S. Ji, Z. Su, and D. Guo, "Multi-objective AGV scheduling in an automatic sorting system of an unmanned (intelligent) warehouse by using two adaptive genetic algorithms and a multi-adaptive genetic algorithm," *PLoS ONE*, vol. 14, no. 12, Art. no. e0226161, 2019, doi: 10.1371/journal.pone.0226161.

[16] M. A. Naeem, *pyamaze*, ver. 1.0.1, Python Package Index, 2021. [Online]. Available: https://pypi.org/project/pyamaze/

[17] M. De Ryck, M. Versteyhe, and F. Debrouwere, "Automated guided vehicle systems, state-of-the-art control algorithms and techniques," *J. Manuf. Syst.*, vol. 54, pp. 152–173, 2020, doi: 10.1016/j.jmsy.2019.12.002.

Author

Author – Furqan Yousaf, Department of Mechatronics and Control Engineering, University of Engineering and Technology, Lahore, Pakistan. E-mail: 2021mc64@student.uet.edu.pk, engr.furqanshahid64@gmail.com. ORCID: https://orcid.org/0009-0005-5151-9085.

Correspondence Author – Furqan Yousaf, 2021mc64@student.uet.edu.pk; alternate e-mail: engr.furqanshahid64@gmail.com.