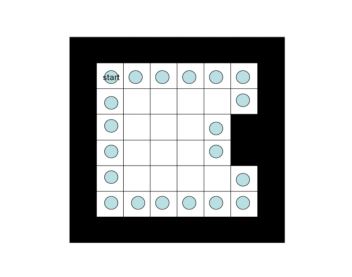
**A report on wall - following using genetic algorithms:**

**Problem: Wall-Following –** Use genetic algorithm to achieve a robot that follows the path of the marked boxes as efficiently as possible.

****

**Initialization**: We begin any genetic algorithm by randomly creating an initial population (solution space) from which we later evolve new generations or new solutions. In this problem, our population consists of n robots where each robot is represented as a bit-string of its moves. Each move of a robot is represented as the combination of two bits such that 00 means do nothing, 10 means turn left, 01 means turn right and 11 means move forward and thus the bit-string is just a concatenation of all of its moves. Since our goal requires for the robot to complete the task using 28 moves thus, the length of each string has been kept as 56. To generate our initial population, we thus use two for loops where the outer loop runs for our decided population size which in this case, we have set to be 20. The inner loop runs 56 times (our string length) and in each iteration it produces a randomly-generated 1 or 0 and concatenates the 56 bits into one string. Thus, our solution space consists of 20 random strings each of length 56 bits. We set this as our initial population or generation 1 and store it a list in python.

Example String: 11111001111101111011111101001111111101111110011111000011

Moves: 11-11-00-11-11-10-11-11-01-11-11-10-10-01-11-11-11-10-10-11-11-10-01-11-11-00-00-11

|  |  |
| --- | --- |
| Signal | Action |
| 00 | Do nothing |
| 01 | Turn right |
| 10 | Turn left |
| 11 | Move forward |

**Fitness**: For any problem, we have certain set criteria and depending on how well each individual of the population completes those criteria (quality of the solution), we award it points and call it the fitness of the robot. In this problem, our criterion is that the robot covers all the boxes next to the wall (marked in blue in the grid) in the restricted set of moves. So, for each robot we can score its fitness as the number of marked boxes it covers. To do this, we run each string move by move against our grid and every time the robot traverses over a new marked box, we award it one point. We do not award the robot any points if it traverses to an already visited box. So, for each robot, we must keep a track of which boxes the robot has already traversed. This can be done by creating another grid called IsVisited and initially set each box to 0. When the robot starts traversing and covers a box, update the 0 to a 1 for that box in the isVisited grid. We then set a condition that the robot is awarded points only if the box is a marked box and the box has a 0 in the isVisited grid. So, for each string we calculate the fitness and store it in a fitness list.

**Roulette Wheel Selection**: In genetic algorithm, we randomly select solutions from our current solution space/generation and pass their copies on to the next generation while keeping the population size the same. The roulette wheel selection is a method by which we can give solutions with higher fitness more chance to be selected and passed on to the next generation. Metaphorically, we can create a wheel in which the portions are divided based on the relative fitness of each solution (that means fitness of solution over the sum of all fitness of the population). We then spin the wheel n times where n is the size of the population (20) and select the one where it lands. This creates a greater probability of selecting the higher quality solution. The way I have implemented this in my code is shown by the table below:

Using an example to explain, I am considering this population of 5 with their fitness shown. Using their cumulative frequency, we develop certain ranges for each sample and then randomly generate a number between 0 and the total fitness (in this case 21). We then choose the solution whose range the randomly generated number fell into. Example: here if we get a 0 or 1, we choose sample A, if we get 2,3,4,5 or 6 we choose B, if we get 7,8,9 we choose C and so on. This allows for there to be a greater chance for those with higher fitness, since we have a greater range for D than for A.

|  |  |  |  |
| --- | --- | --- | --- |
| String Sample: | Fitness Value | Cumulative frequency | Range |
| A | 2 | 2 | 0 - 2 |
| B | 5 | 7 | 2 - 7 |
| C | 3 | 10 | 7 - 10 |
| D | 7 | 17 | 10 - 17 |
| E | 4 | 21 | 17 - 21 |

Thus, using roulette wheel selection, we are able to pass the higher quality solutions to the next generation.

**Genetic Operators**: Genetic operators refer to Mutation and Crossover. Crossover occurs more often than mutation because chances of mutation are very low.

**Mutation:** Mutation refers to a random flip in a bit inside the strings. For the parents we have selected from our current generation by roulette selection we can apply mutation on their strings. However, the chance/rate of mutation occurring are usually very less (About 1/pop-size). Using this formula (was given in the slides) I calculate the percentage chance of fitness which turns out to (1/20) \* 100 which is 5%. For each string I randomly generate a number between one and hundred, if the number is below 5 then the mutation function is called else the function is not called. Inside the function, a randomly generated number determines how many mutations will occur on the string and we run a loop that many times. In the loop, a number between 1 and the length of the string (chromosome) which is 56 in our case will be generated and we flip the bit at that index.

**Crossover**: Crossover refers to the exchange of a sequence of bits between two parent bit strings starting from a certain index value till the end of the string. From our roulette selected solution space, we first shuffle the mating pool and then divide them into pairs (10 pairs). For each pair, we then have to decide whether crossover shall take place or not. The crossover rate is between 0.6-0.9. We randomly generate a number between 60 and 90 as our deciding rate and then for each pair we generate a number from 1 to 100. If the number is less than our decided rate then we call the Crossover function. If the number is greater than our decided rate, we do not apply the crossover functions and pass the parent pairs as they are to the next generation. Inside the crossover function we randomly generate a number between 0 and our string length (56) and choose that number as the starting index. The sequence of bits after that starting index are switched for the parents and the two recombinant strings that are formed are passed onto the next generation.

**Analysis:** Now that we have generated our new generation, we will run this whole process again to get more and more generations. We see that as the generations go on, the quality of our fitness space starts to increase. While we initially got fitness ranging from 2 till 6, a few generations later the lowest fitness individuals are eliminated while the ones with higher fitness go on to produce even better solutions.   
An example run of my code shows:

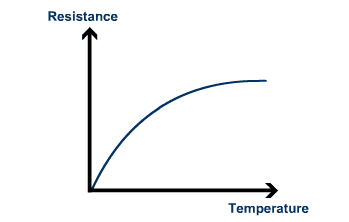
Generation 1: [2, 3, 4, 2, 2, 4, 2, 3, 5, 5, 2, 3, 9, 3, 5, 2, 2, 2, 1, 3]

Generation 30: [7, 7, 6, 7, 7, 7, 7, 7, 6, 3, 7, 7, 8, 7, 9, 7, 9, 7, 7, 8]

Generation 150: [12, 12, 12, 10, 12, 12, 11, 12, 12, 12, 12, 7, 11, 7, 12, 12, 12, 12, 12, 11]

Generation 200: [12, 15, 12, 12, 12, 15, 12, 15, 12, 12, 15, 15, 12, 12, 12, 15, 15, 12, 12, 12]

So thus, we can see there is in-fact an increase in the quality of solutions as the generations go on and this occurs at a decreasing rate. Understanding this phenomenon, we should assume that after a finite amount of cycles we should reach the optimal solution.



--20

--16

Quality of fitness 🡪

Generations 🡪

However, there are certain problems that I have encountered in trying to achieve the optimal fitness. Running the code multiple times and for over even 2 lac cycles does not generate the optimal solution. In fact, the highest fitness that I am able to achieve is 16 and when run for an almost indefinite amount (2 lac cycles) we reach a point where almost all the solutions give the fitness 16, which is an increase in the quality of the population, however a single solution does not rise above the fitness of 16.

Generation 100392 = [15, 16, 16, 16, 15, 16, 15, 16, 15, 14, 14, 16, 16, 16, 16, 15, 15, 16, 16, 16]

This is because by looking at the solution of 16, we can see that the robot tries to follow the path which awards it the most point. Now in this, we can see the point where it goes wrong (marked as P) is because of the two blocks that are protruding from the rightmost wall. If it goes on to the point P it usually encounters the wall and is not able to successfully remove itself from the pocket that it is stuck in leading to it getting a lower fitness score (around 6 or 7). If it instead turns at one box left of P it is able to get more score because of two protruding boxes and then is able to move onto following the rest of the wall (getting it fitness of 16).

The optimal solutions consist of very specific sequence of moves that allow it to traverse this point which are turn-right / move forward / turn right / move forward / turn left / move forward. Now finding this very specific sequence at the specific point that we require is very difficult and has very low probability. Ideally, what we would is that a mutation or crossover allow for some of this region to be generated and that would be passed on to the next generation and to the next and after some generations, more of this sequence would be made until we reached the complete optimal (and the robot is able to maneuver its way out of the pocket). However, in this problem even if such a mutation occurs and we get part of this specific sequence, its fitness value drops (because it encounters the wall) and its chances of getting to the next generation become lower and very soon it is eliminated from the solution space. Which means that it needs to stay in the solution space until enough generations have passed for it to be able to evolve into the proper maneuver to get out of the pocket, but since its fitness falls to 7 at this time, it is eliminated before it can achieve this.

So basically, to reach the optimal we would need very specific mutation/crossovers to all occur at the same time (or in close generations) which allow for this complete sequence to be made before it is eliminated for having lower fitness. The probability of that occurring is extremely low. Even when by slim probability this sequence is developed, the fitness goes from 16 to 17 or 18 and then we would need for it to get the sequence for the next pocket (shown in the grid) for it to reach fitness 20. Thus, it can be concluded that the probability of reaching the optimal solution using my current algorithm is extremely low. The Near optimal Solution that my robots can achieve every-time is 16 and In rare conditions, it has reached fitness 17 and even 18 once. Thus, the termination condition I have set is to check for 17 or greater and if not found within 1lac generations, then return the fitness of 16. The number of generations it takes to achieve the goal can vary depending on the initial fitness of the strings, the mutations and crossovers that occurr.

Near-optimal traversal achieved by Algorithm. Fitness 16 (can be either clockwise or anticlockwise)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **W** | **W** | **W** | **W** | **W** | **W** | **W** | **W** |
| **W** | 1  **>** | 1 **>** | 1 **>** | 1  **>** | 1  **>** | 1  **P** | **W** |
| **W** | **1**  **^** |  |  |  | **v** | 1 | **W** |
| **W** | **1**  **^** |  |  |  | **1**  **v** | **W** | **W** |
| **W** | **1**  **^** |  |  |  | **1**  **v** | **W** | **W** |
| **W** | **1**  **^** |  |  |  | **v** | 1 | **W** |
| **W** | **1**  **^** | **1**  **<** | **1**  **<** | **1**  **<** | **1**  **<** | 1 | **W** |
| **W** | **W** | **W** | **W** | **W** | **W** | **W** | **W** |

Optimal Solution. Fitness level 20

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **W** | **W** | **W** | **W** | **W** | **W** | **W** | **W** |
| **W** | 1 **>** | 1 **>** | 1 **>** | 1  **>** | 1  **>** | 1  **v** | **W** |
| **W** | 1  **^** |  |  |  | **v** | 1  **<** | **W** |
| **W** | 1  **^** |  |  |  | 1  **v** | **W** | **W** |
| **W** | 1  **^** |  |  |  | 1  **v** | **W** | **W** |
| **W** | 1  **^** |  |  |  | **>** | 1  **v** | **W** |
| **W** | 1  **^** | 1  **<** | 1  **<** | 1  **<** | 1  **<** | 1  **<** | **W** |
| **W** | **W** | **W** | **W** | **W** | **W** | **W** | **W** |