Robot Navigation in AI

Asmaa Sehnouni
Computer Science and Technology
Temple University
Philadelphia, USA
tug25597@temple.edu

Jasmine Dsouza
Computer Science and Technology
Temple University
Philadelphia, USA
tue97411@temple.edu

***Abstract* — A solution is designed that will find the path of a robot from random points A and B, in the best possible track/path while also avoiding obstacles. This can work for a random selection of start and end points and this, can also work for a user selected value of start and end points. This solution is optimized using Breadth first search and also a comparable solution is designed using Genetic Programming. We have done the coding in Python but this can be replicated in any other comparable programming language.**

I. PROBLEM STATEMENT

We consider a problem where a robot has to traverse a path from A to B. We also consider that this path must be the shortest path from A to B. To further complicate the scenario, we consider obstacles. The robot should find the shortest path from A to B while also avoiding obstacles. We consider that the robot can travel in any direction as long as the robot is facing the desired direction. The robot can travel in 1, 2 or 3 steps at a time. So for example, the solution can contain "a3, R" which means the robot has travelled 3 steps and turned right. Or the solution can contains "L, L and a2" which mean the robot took 2 left turns and then travelled 2 steps.

We also consider an iteration of the problem where the user can provide the variable values – which are the start and end points, and obstacle locations. These values can be part of the file the user provides. The same solution can be used to solve both the random creation of the problem and user defined values for the problem. In the user defined file, the first line in the file contains the number of rows and columns. From the second line, the user fills out the array as an array of 0's for the given number of rows and columns. A 1 value denotes an obstacle. At the end of the desired number of rows and columns, the user can denote the location of the start and end points and direction the robot is facing in the start. A "0 0" at the end of file denotes the last line and the processor knows to stop reading the file at this line.

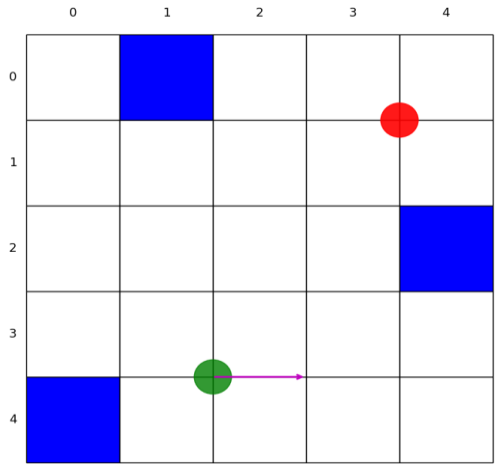

Figure 1

In Figure 1, we can see an example of the Problem. Start point is denoted by a green dot and the direction the robot is initially facing is denoted by an arrow from the green dot. End point is denoted by a red dot. Any obstacle is denoted in blue. Robot cannot travel to any path or node containing the obstacle. To face the desired direction, the robot can perform movement steps like Left and Right. For example, if the robot if facing up and wants to move left, it can perform one Left Movement.

Once we completed the design of the solution using Breadth First Search, we focused on an alternative solution using Genetic Programming.

II. SOLUTION

We designed the solution in Python due to the flexibility of the language and the ease of use of the Graphs and various libraries.

In the solution, we first request the user to enter the number of rows and columns using Tkinter which is a widget creator tool. We also allow the user to enter the number of obstacles. If the number of obstacles is larger than the grid (matrix nodes), this solution will not work. The first step of the solution algorithm is to generate the obstacles. We create a matrix using the number of rows and columns entered by the user. We fill this matrix with all zero values. We then generate the obstacles in random locations in the matrix using numpy commands.
Once the obstacles are generated, we generate the graph with all the possible solutions using networkx library for DiGraph. If the node is an obstacle node, we do not add the 4 directions and the path traversals. If the node is not an obstacle node, we add the 4 directions as 4 possible nodes the robot can be facing in. We also add all the actions the robot can perform to go from one possible direction to the other. We also check if the robot can travel 1 or 2 or 3 steps from the considered node and we add edges accordingly. In this way, every path mapped out is an achievable path.

The next step is to find the optimum solution from all these solutions. The main crux of this solution is we used Breadth first optimal path solution. For this solution, we give 2 attributes to each node – we give each node a distance value and a parent value. We initialize all the distances to infinity. We revise these distances as we find a path to the final node. We start out from the start node. Initialize distance of start node to 0. Then we place start node in the queue. We look for all its neighbors and place them in the queue. All the neighbors get a distance of 1 since they are one edge away from the start node. Parent of the neighbor node is marked as the start node. Once all the neighbors of a node are considered, the node is taken off from the queue and the next node from the queue is considered. 1 is added to the distance from the parent node for the neighbor/child node. We use the start node and loop through the start and all its neighboring nodes.
We then take the end node and try to trace a path back to the start node.

Figure 2

In figure 2, we can see an example where the user selected 5 rows and 5 columns and 3 obstacles. The start and end points and obstacle locations are random. We can see the final path in yellow.

PSEUDO CODE TO SOLVE PROBLEM USING BREADTH FIRST SEARCH:

- Read File to create a matrix OR Generate random start point, end point, and obstacles.
- Generate a graph
- Choose all the solutions in the graph from start to end point
- Pick the short solution using bread first search
- Build the grid
- Display the grid with path
- Write the path in a text file

The result diagram and graph is displayed using matplotlib library. In the resulting diagram, the start point is indicated by a green circle and end point by a red circle. The direction of the robot when he starts is indicated by an arrow from the start point. Any obstacle is indicated by blue. A node in white basically has no obstacles.

III. OTHER SOLUTION USING GENETIC PROGRAMMING

Genetic Programming is a very unique and powerful solution. Genetic programming might not always be able to solve new questions but they can find better answers to existing questions. Hence, we decided to experiment one solution with genetic programming to test its implementation and application. The central theme of genetic programming is in how robust the fitness function has been designed. We noticed this when we tested various iterations and worked through the solution. Fitness score is a measure of how good the current individual is at solving the problem on hand. Genetic Programming tries to emulate natural selection

and evolution. The individuals in Genetic Programming will not change. But as in evolution, the individuals with a higher fitness value, will survive and create more children or offsprings which have a higher chance of survival.

Before we can use genetic programming to solve the problem on hand, we have to design a way of encoding the potential solution to the problem. The solution is designed with a given start and end point and a certain number of obstacles. We already know the ideal path using the previous solution.

We designed the individual as a possible solution to the problem. We choose 3 in our example as we are aware that the solution can be found in 3 robot movements. The population is designed as a number of individuals. We choose 100 in our example. We start off with an initial population of 100 individuals, then we give each individual a fitness number. We calculate, for each individual, the final X and Y coordinates, considering we have the start coordinates and direction. If the final coordinates are out of the bounds of the graph, we add a 100 to the fitness function. We calculate the difference between the final coordinates and the start coordinates and add this to the fitness number. The less the number, the closer the individual is to the actual solution or final location. The fitness function takes into account if the individual only has direction change steps like Left, Right, Up and Down and not any movement steps like take 1, 2 or 3 step forward. We add 50 to the fitness if there are no movement steps because obviously the robot is not any closer to the end point without any movements.

We then calculate the average fitness for the entire population. After the average fitness, we then evolve each population into the next generation. The evolution process starts off with calculating the fitness of each individual then grading the population by best fitness first. We can select the retain function to be 20%. Hence, 20% of the best population is selected as the parents and are retained in the next generation of individuals. Besides the parents, random individuals are selected into the list of parents, from the rest of the current generation in order to add genetic diversity to the next generation. Based on a 1% mutation factor, a random action is added to a random location in an individual in the selected parents. The fitness is then reevaluated for the modified parent.

We then crossover the parents to create the children so that the total number of individuals in a population is the same (100 in our case). Till we can create the entire group of children, we select 2 different random individuals from the parent population as male and female, then we select the first 2 from the male and the last 2 from the female parent into the crossover child individual.

In this way, we evolve populations for 100 generations of individuals. We can notice that the fitness value improves considerably. One same result set is attached.

PSEUDO CODE TO SOLVE PROBLEM USING GENETIC PROGRAMMING:

- Set the size of population(100) and length of each individual(3)
- Pick a random set of individuals for the first population
- Calculate the fitness function of each individual in the population. Fitness function has been described above.
- LOOP: for a new generation, pick the top 20 individuals with best fitness value.
- LOOP: Randomly add other individuals to promote diversity
- LOOP: Mutate some individuals.
- Cross over the parents to create children who are part of the new generation.
- Repeat the steps for creating the new generation 100 times.

IV. TIME TAKEN AND CHALLENGES

In Breadth First Search, since the robot cannot travel any side or edge of the node, we had to decide how to design the Graph and that was challenging. Selecting the shortest path while also avoiding the obstacles was a challenge. There are various algorithms available which could solve this problem like A* or Depth First Search. But we selected Breadth First due to its ease of use and performance.

In Genetic Programming, selecting good parameters for the fitness function was challenging. For the fitness function to work – we made it depend on how far the individual was from the final location. Also, if the individual has no movement steps, then we gave it a higher value (further away from solution) since it is not moving to the final step. If we assume we already know one ideal response, we can compare each individual to the ideal response for the fitness function too.

V. CONCLUSION

Genetic Programming can do well in any environment but they work especially well when the set of solutions

can be very large and has many peaks and ebbs. Genetic Programming can work well where there is no ideal solution. In Genetic Programming, we do not need to do any preparation or cleanup for the input and output data. However, we might need a large population size and more generations to give good or ideal results. There is no guarantee of finding the ideal solution. Most of the applications in Genetic Programming depend on how well the fitness function is defined. In Genetic Programming, a key parameter is selecting the children and parents for next generation. It has to take the best of the current generation and also have enough mutants to make the generation diverse. It might take some time to code as there are not as many example using Genetic Programming and there is no standard for use.

Time: Time complexity of breadth first search is at the least O(V+E) where V is the number of vertices and E is the number of edges. There is a guarantee that breadth first search will find a solution if exists. Same cannot be said for Genetic Programming. Also, if there is more than one solution, breadth first search can be used to find the minimal solution. The main issue with breadth first search is that it requires higher space to save all the node values. Also, if the final node is far away from the start node, solution will take more time.

For the scenario we have selected, we prefer using the traditional approach of Bread First Search. Breadth First Search will definitely find a solution if one exists. Genetic Programming may or may not find a solution. The more the number of generations or evolutions, the better the chances of a solution. However, after a certain number of generations, the population almost stabilizes and we do not see a large variation.

VI. REFERENCES

1) Genetic Programming and AI Planning Systems1, http://faculty.hampshire.edu/lspector/pubs/gp-aaai.pdf
2) https://en.wikipedia.org/wiki/Breadth-first_search
3) https://en.wikipedia.org/wiki/Genetic_programming
4) http://www.genetic-programming.org/
5) http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html