SOLVING A 3D MAZE WITH THROWING A BALL BY AN EVOLUTIONARY ALGORITHM

Tiit Vaino[1], **Sander Roosalu**[1], **Jakob Univer**[1]

[1]Institute of Computer Science, University of Tartu

UNIVERSITY OF TARTU

Introduction

The evolutionary algorithms (EAs) are designed with an idea to mimic how organisms evolve over time in nature. EA starts with the initial population. In every iteration, the new generation improves by using crossover and mutation to explore the results space and then choosing the fittest individuals to mimic natural selection.
EA applications are used for exploring large search spaces. This helps to reduce the computation time and helps to find good enough solutions.
We were fascinated that an algorithm, that mimics how things work in nature, can help solving many real-life problems. So we wanted to try out and get a better understanding of how an evolutionary algorithm works.

Problem

We tried to solve a problem where a ball is thrown inside a three-dimensional room, in conditions where there is neither gravity nor friction and the materials are perfectly elastic. The goal is to throw a ball in a direction so, that it reaches (as close as possible) to the endpoint somewhere in the room. The closer the ball reaches the destination the better the result. There are some obstacles on its way, which makes it harder to reach the endpoint. The ball can bounce from the room walls and obstacles.

Objective

Our main goal is to test out if we can use an evolutionary algorithm to solve this ball-throwing problem.

Methodology

Our plan was to implement an evolutionary algorithm. There are many different ways to do it. Our solution's high-level logic can be seen in Figure 1.

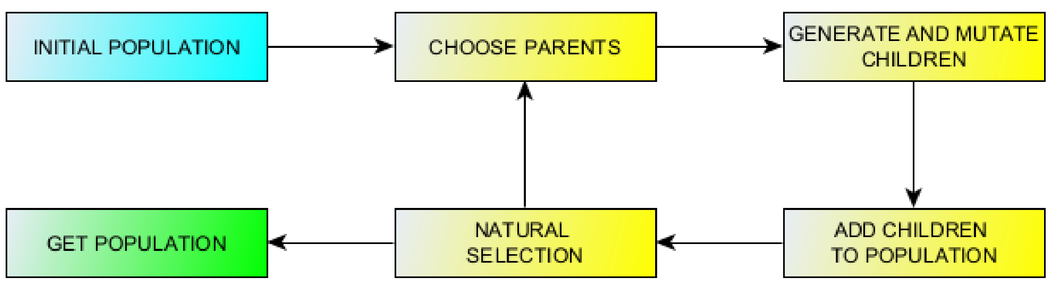

Figure 1. EA high-level logic

Initialising population

Getting the starting population right is quite important for the evolutionary algorithms. This helps to reduce the searching time and avoid local optima. Our algorithm's starting population is created with a lot of different regions under interest. This is achieved the following way: we randomly choose our angles for both dimensions in the range $[0°, 180°)$, then we randomly choose if the angle is negative or not. This way we get that angles are in the range $(-180°, 180°)$. That gives us an initial population, that is equally distributed in two-dimensional space.

Choosing parents

To get better offspring, good parents are needed. We chose a method where each individual inside the population is assigned a weight. The higher evaluation for the individual the higher weight. For our problem, the closer the ball gets to the endpoint in the room, the higher the parent's evaluation. The sum of all the probabilities is 1. Now we are randomly choosing some amount of pairs from the population with the help of weights. This way fitter (more highly evaluated) individuals are more likely to be chosen, but the less fit individuals still have a chance. This helps to keep diversity by not forcing some niche out of the competition.

Generating children

This step combines crossover and mutation. Both of them are core functions inside evolutionary algorithms. They help to explore the space for the best solution.

To perform a crossover on a pair, we take one angle from one parent and other angle from another parent. This way we get two new children as can be seen in Figure 2.

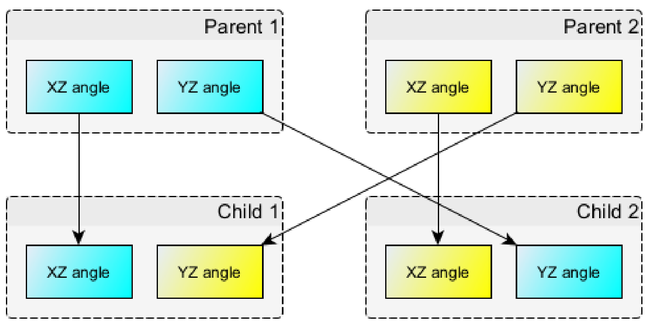

Figure 2. Crossover

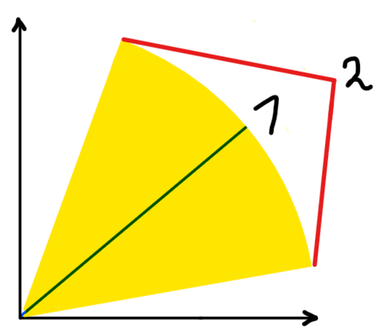

Figure 3. Angle mutation range

After we have done the crossover, we mutate both children's angles. To do this, we calculate for each angle mutation size by using normal distribution:

$$\text{new\_angle} = \text{old\_angle} + N(0, \sigma), \text{ where } \sigma = 360/6.$$

This can be seen in Figure 3, where 1 is the initial angle, and 2 shows a possible normal distribution cover. If the new_angle is out of the range $(-180°, 180°)$, then it will be just translated back into the initial range.

Natural selection

After all this, we merge children with the population. The new generation will favour fitter individuals, similar to choosing parents. This way we are mostly choosing good individuals, but we still leave room for some exploration of other niches. It helps to avoid getting stuck into local optima.

Finding solution

Now we have everything to generate a new generation. But as in nature to evolve, we need to iterate through many generations. The only problem is that in nature it happens infinitely, but we want our solution sooner. So we have to know when to stop our evolution.
Our algorithm allows the user to control the EA with four restrictions. First is how big the population can be. This helps to control the exploration rate in the search space, but also control the computation rate. Secondly can be controlled how many iterations can be done, therefore controlling the computation time. Thirdly can be controlled how long the algorithm can run, achieving a similar effect as in the second restriction. And lastly, a user can restrict how many iterations without progress can be done. This helps to stop earlier if the solution is found.
After the algorithm has stopped, it returns the population with evaluations, out of which a user can choose the desired solution.

Results

For relatively open "mazes" the program took under a minute to compute the possible paths for the ball. For more closed mazes, where a human would find the solution rather easily, the program took hours to find the solution. We suspect that this is because we did not limit the length of the throw enough and the iterations of the program ran for too long even if there was no way for the ball to get to the finish. An example of this is when the ball is only given an up-down movement in a case where the endpoint is to the side of the ball.

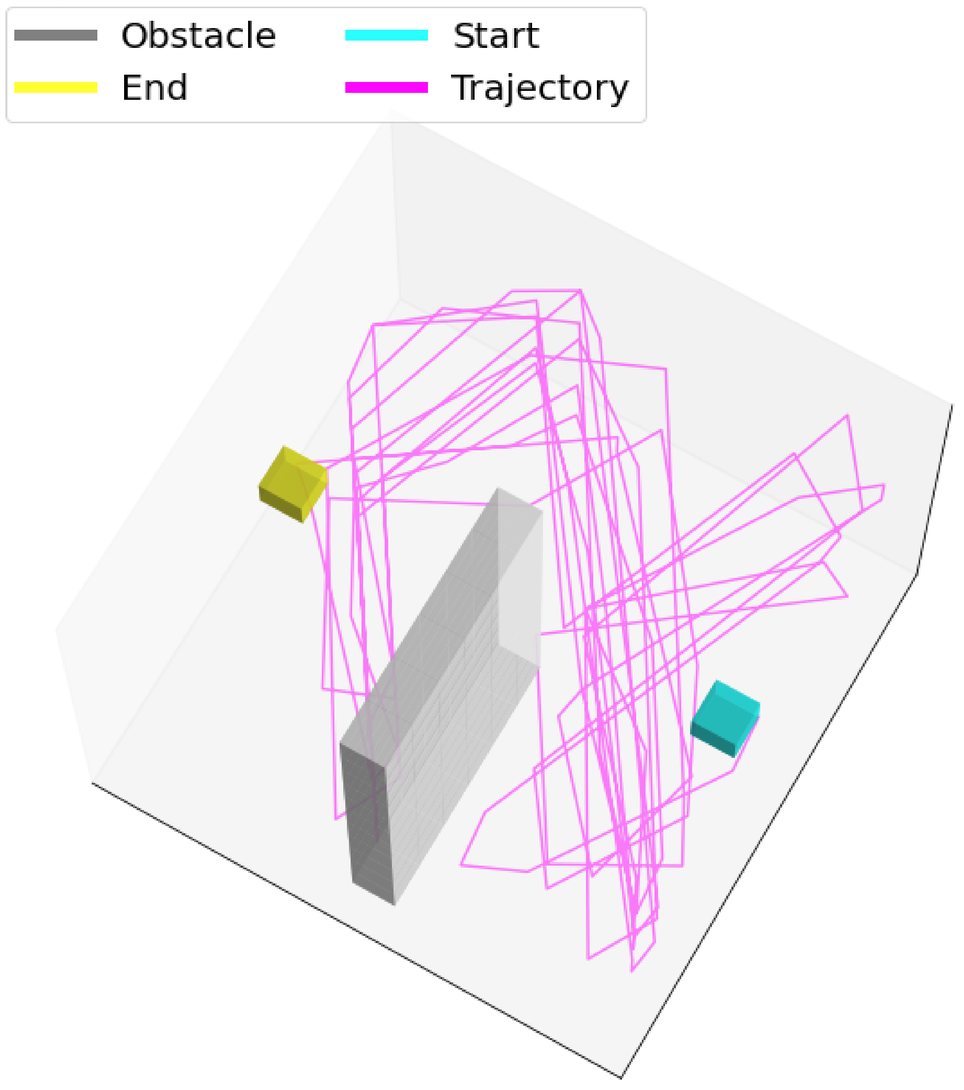

Figure 4. Example of one possible solution in a relatively open maze

Most of the runs were rather messy with a lot of bouncing against the walls. As it can be seen on the Figure 4, some bounces do not connect to a wall. For this we suspect that something went wrong with either calculating the bounce coordinates on obstacles or visualising the trajectory of the throw.

Conclusions

The EA worked with simple mazes with a small amount of obstacles, finding solutions there within a minute, which is rather good. If the maze had a little bit of complexity however, the EA was not really feasible for finding the optimal path to the endpoint. At that point, the solution could be found faster and more reliably with an another algorithm or with just human vision.
If we would have optimized the solution, it would have been easier to test. Also if we would have limited the number of bounces or even further limited the throw length, the EA would not have wasted so many resources on running the trajectories and would have been faster in finding the solutions.

References

Our GitHub code repository. https://github.com/JakobUniver/algorithmics_3D_maze