Ahmad Humayun – 21100053 – AI (Spring 2019) – Assignment 1

Encoding the solution:

The most important step in solving a problem using genetic algorithms is to find a way to represent the solution in the form of a manipulatable string. There were multiple ways to do this. The one used here is a mapping between each of the 32 possible combinations of the robot's sensors and a corresponding action the robot will take for that sensor reading. The robot has 5 sensors (to detect walls to its West, North-West, North, North-East and East) that are either HIGH or LOW depending on whether the sensor detects a wall in that direction or not. Given this configuration the readings from the 5 sensors can be encoded in 5 bits, each bit representing a sensor:

<West><North-West><North><North-East><East> = 00000

From this we can immediately see that the robot has a maximum of 32 unique 5-bit sensor readings each representing a different wall pattern. We can store a response to each of these unique 5 sensor combinations in the form of a 2-bit string to represent the 4 possible moves. Our final string representation will not explicitly contain the 5-sensor readings as there is no need to do so. Instead the final solution string will be a 2-bit response for each of the 32 possibilities resulting in a 64-bit solution. The *position* of the 2-bit code inside the 64-bit solution string will indicate which 5-bit sensor reading it corresponds to. The format of the string is as follows:

2^{63}	2^{62}	2^{61}	2^{60}	….	2^{1}	2^{0}
Response to 11111		Response to 11110		….	Response to 00000	

This way, the encoded solution string essentially becomes a look-up table, indexed by the sensor reading.

Algorithm:

- Randomly generate 20 64-bit integers as a starting population
- Calculate fitness for all individuals
- Select the top 50% fittest solutions and put them in a separate list
- For each consecutive pair in the list apply crossover at any random position between 1 and 63
- The last individual performs crossover with the first individual in the list. This restores the population to its original size. (for a population of size **n**, n/2 individuals will be selected and n/2 crossovers will occur resulting in (n/2)*2 children)
- Mutate the children with a probability of 1/64 <= p <= 1/20
- Replace the current generation with all the children
- Repeat from step 2 until goal fitness is reached

Calculating fitness:

The fitness was calculated by setting up a basic simulator that manages and tracks the progress of the robot for a fixed number of sense/act cycles. Out of the 28 moves, we count how many moves it stayed with the wall. There are a total of 20 positions that are against the wall on the test map. We don't score the starting position so that leaves a maximum score of 19 that the robot can obtain in 28 cycles. We also don't want to encourage the robot to back-track so a list of already visited positions is maintained and the robot is not scored multiple times for visiting the same box.

Crossover and Mutation

Crossover can be achieved by simple bit manipulation and masking. Separate the head bits from the tail at the crossover point **p**. Use bitwise or to recombine the head of a gene to the tail of the other.

```
def crossover_mutate(gene1, gene2, p):
        MASK =  0xffffffffffffff
        head_mask = (MASK << p) & MASK
        tail_mask = MASK - head_mask
        g1_head = gene1 & head_mask
        g2_head = gene2 & head_mask
        g1_tail = gene1 & tail_mask
        g2_tail = gene2 & tail_mask

        return [mutate(g1_head | g2_tail), mutate(g2_head | g1_tail)]
```

Mutation can be done by taking a bitwise xor with a 1 on the position we want to apply the mutation. Formally, the probability of a bit being mutated can range from 1/string_length to 1/population_size. Which is 1/64 to 1/20 in this case. Running the code multiple times showed that a mutation rate nearer to the upper-bound (1/20) works really well compared to a lower rate of mutation such as 1/64.

Advantages and Limitations

The obvious benefit of this approach is that even though it was tested on a given map, it can solve most other maps that have a similar structure. However, by encoding the solution like this the number of bits required to store responses to all the sensor combinations goes up exponentially with the number of sensors. For the N-sensor robot the bits required would be $2^N * 2$.