# IMAT5232: Evolutionary Computing

## Assignment 2: Robots in Donutland

# Introduction

The purpose of the assignment was to design and develop a genetic algorithm for designing an optimised robot for Donutland. Donutland is so named as it is toroidal.

The robots to be designed had to comply with a specified genetic profile, as listed below:

* Energy depletion rate (max = 8, min = 1)
* Maximum energy (max = 8)
* Energy depletion reserve (max = 50)
* Protective casing repair rate (max = 8, min = 1)
* Maximum protective casing (max = 8)
* Protective casing repair reserve (max = 50)
* **[OPTIONAL]** Sensor strength (max = 2, min = 0)

Additionally, the robots had to be able to interact with their environment by responding to contact with both energy sources and corrosive patches.

In this report I will discuss the development of the simulation and the genetic algorithm and discuss the performance of the genetic algorithm in regards to the specified problem.

# Simulation system design

When developing the overall simulation system I decided to identify the key items and activities in the specification and break these down into bite sized functions. This meant that each function could be developed and tested independently; this was designed to make debugging as simple as possible.

## Modelling robots

The specified genetic profile of the robots has various maximum values some of which are not easy to represent as 4 bit groups of binary values. For this reason I decided not to model the robots as a binary string, but as a struct with a field for each genetic profile element plus additional fields where needed to aide management of the robots in the simulation. Where appropriate the field values were initialised with random values in the specified ranges.

Updates to the robots were split into four separate functions: updateRobotGeneric, updateRobotLocation, updateEnergyFound and updatePatchFound[[1]](#footnote-1). By separating the interactions with the environment it made them easier to develop and manage within the main simulation. Ongoing changes to energy levels and protective casing plus checking of status are handled by the updateRobotGeneric function; this is always called as is updateRobotLocation. The other two functions are only called if they can be found in the robot’s current location.

## Modelling Donutland

To model Donutland I created a multidimensional struct array. As each square in Donutland could contain 1+ energy patches or a corrosive patch I decided to model each square as a struct. The struct comprises of two fields: energysource and corrosivepatch. These were both initialised to zero on creation. To represent the structure of Donutland the structs were stored in a 40 x 40 array. An energy source(s) or a corrosive patch was randomly allocated to the squares. The toroidal nature of Donutland is modelled by the updateRobotLocation function, when a robot disappears off one edge it reappears at another.

As the number of robots in the population exceeded the possible directions of movement I decided to wait until all robots had moved a step before updating the squares visited. This prevented a robot being disadvantaged by another robot ‘visiting’ the same square on the same step depleting an energy source(s), enabling a fairer comparison of performance.

Upon initialisation a copy of Donutland was stored so that it could be used to reset the environment so that each generation of robots had the same experience and could be fairly compared.

## Implementation

I designed the main simulation to run until either all robots were dead or for 4000 iterations. I chose 4000 iterations so that potentially all energy sources would have been found or would start to run low. This is designed to enable a more realistic assessment of the sustainability of the robots in a low energy source environment.

Once the main simulation had executed the robots were assessed for fitness and evolved. The updated population was then used in the main simulation with the reset environment. This was designed to occur for the number of iterations defined by the variable evolutions.

The program was designed to output to the screen the most suitable robot from each generation, the calculated average fitness and the best fitness. This was to enable the user to assess the effectiveness of the evolution. When executing the program the average fitness value was not calculating correctly so I decided against recording these values to use in the comparison of the robots being evolved.

# Design of the genetic algorithm

The purpose of the genetic algorithm was to aid the development of a robot that could best survive the Donutland environment. This was my main consideration when deciding upon an appropriate fitness measure. As there is no energy cost in replenishing the protective casing of the robot I did not incorporate this into the fitness function. I therefore decided to base the fitness function solely on the longevity of the robots. I have used a simple bubble-sort algorithm to organise the robots from the shortest surviving to the longest.

For parent selection I experimented with both the roulette and elitist parent selection methods. The roulette method was chosen as it does not exclude potentially good genetic material from poorer performing robots unlike elitist selection methods and is not wholly random as it is weighted[[2]](#footnote-2) in favour of the better performing robots. As my population is small there is the risk of negative side effects such as the Founder Principle, Genetic Drift and the Bottleneck Effect.

Additionally, I also experimented with the implementation of mutation. Initially the level of mutation and the new values were randomly generated. This was amended and compared with combinations of refined mutation for a random number of genes, refined mutations randomly affecting 50% of genes and random values affecting 50% randomly affecting 50% of genes. The experiments were conducted for both parent selection methods, the results of these experiments are discussed in the next section.

During development of the algorithm it ran for a fixed number of iterations, this was controlled by a while loop. As this was quite rigid this was amended so that the program ran until the algorithm failed to generate a specified number of better robots than the current ‘best robot’. Results were recorded for this value set to 10 and 20.

The simulation and algorithm predominantly perform as expected although the worseRobot count does not function quite correctly. As I had already recorded a series of results I decided to continue as is rather than amend the simulation and redo a series of experiments. My main concern is that the simulation and algorithm run relatively slowly. This has not caused any problems with regards to the simulation as a piece of coursework but would be too slow for a real world application.

# Implementation of the genetic algorithm

The genetic algorithm and simulation have been implemented using MATLAB. I chose MATLAB as it is the development tool with which I have most recent experience. Additionally, I will be assisting in delivering a MATLAB course shortly and felt that this was a good opportunity to further develop my MATLAB programming skills.

For my first degree I predominantly developed using C++ but also studied Java, but have not really used these since graduation. Although I have been learning a little Python and have previous implemented a web database utilising PHP, I did not feel confident enough to use these for the implementation.

As my previous programming experience was predominantly using much stricter languages such as C++ and Java I found developing in MATLAB frustrating at times, particularly with errors relating to variable or field names as instead of an error being identified it will create a new variable or field. In retrospect it may have been less time consuming in terms of development to have used C/C++.

# Discussion of results

For each parent selection method implemented the simulation and genetic algorithm was executed 8 times. The duration of each execution varied based on the worseCount and the performance of the algorithm.

To illustrate the performance of the genetic algorithm I have charted the Best Fitness value of each generation for the 8 executions. Figure 1 displays the results for the Roulette method, whilst Figure 2 displays the results for the Elitist method. For both Figure 1 and Figure 2 a log scale has been used for the Y axis, this was selected as it enabled clearer reading of the data.

[[3]](#footnote-3)

Figure

[[4]](#footnote-4)

Figure

For each of the 16 executions of the simulation and algorithm illustrated in Figures 1 and 2 DonutLand and the initial Population were re-initialised. This means that we are analysing the trends of the Best Fitness values across generations between the executions not the values themselves per se.

As illustrated in Figure 1 the genetic algorithm was not effective in generating robots utilising roulette parent selection. Looking at the data there is no clear correlation between the performance of the algorithm and the implementation of mutation.

The performance of the algorithm using elitist parent selection was marginally more successful, as illustrated in Figure 2. In particular, the use of Elitist parent selection and a mutation level of 50% with the random setting of values was the most successful configuration as it produced robots of a consistent quality.

It has not been possible to assess how well the generated robots would fare in a low energy environment as all robots had deceased before most energy sources had been found. The use of a larger population size may have made this possible. An additional benefit to a larger population is a greater breadth of genetic diversity potentially leading to better descendants from the algorithm.

Whilst conducting the executions I stored the most successful robot from each execution. Comparing the structure and profile of these robots there does not appear to be clear key characteristics for a ‘Best Robot’.[[5]](#footnote-5)

The most successful robot generated by the algorithm has the following structure and genetic profile:

|  |  |
| --- | --- |
| Energy depletion rate: | 5 |
| Maximum energy level: | 4 |
| Energy depletion reserve: | 4 |
| Protective casing repair rate: | 3 |
| Maximum protective casing: | 4 |
| Protective casing repair reserve: | 1 |
| Sensor strength: | 0 |
| Energy levels: | 4 |
| Protective casing: | -1 |
| Direction: | 5 (SW) |
| Location (at end of life): | [8 20] |
| Alive: | 0 |
| Energy reserve: | 19 |
| Count casing: | 0 |
| Steps (longevity): | 323 |

# Conclusion

If I was to re-development the simulation and algorithm I would introduce a parameter and suitable variables for population size instead of hard coding it as it would enable greater tuning of the algorithm. Additionally, Donutland would be static and not re-initialised each time the program is executed enabling a fairer comparison of the operation of the algorithm.

In conclusion I have produced a functioning simulation and genetic algorithm but have failed to evolve robots that were suitable for their environment. Key factors in this were the size of the population and with how the robots were re-initialised during each execution.

# Appendix

function donutLandSimulation()

%this is the main function

%initialise donutLand and robots

population=initPopulation();

donutLand=initDoLand();

resetDoLand=donutLand;

%create evolution loop

generations=1;

worseCount=0;

while worseCount<20

%set counters for sim loop

deadRobots=0;

loopCount=0;

%set main loop for run of simulation

while or(deadRobots<10,loopCount<4000)

%int robotLoopCount

robotLoopCount=1;%enables loop through of population

locationCount=0;

while robotLoopCount<11

if population(robotLoopCount).alive==0

deadRobots=deadRobots+1;

robotLoopCount=robotLoopCount+1;

else

locations=initLocations(); % create array to store robot locations

%disp(locations())

%locationCount=0

population(robotLoopCount)=updateRobotLocation(population(robotLoopCount));

if locationCount==0

locations(1).loc=population(robotLoopCount).location; %store 1st location automatically

%disp ('location 1:');

%disp(locations(1));

locationCount=locationCount+1;

else

count=1;

present=0;

while count<9

if locations(count).loc==population(robotLoopCount).location %check if location already present in array

present=present+1;

end

count=count+1;

end

if present==0 %if location not present, add

locations(locationCount).loc=population(robotLoopCount).location;

%disp('location 2');

%disp(locations(locationCount));

end

locationCount=locationCount+1;

end

%update to match genetic profile

%disp('robotLoopCount');

%disp(robotLoopCount);

population(robotLoopCount)=updateRobotGeneric(population(robotLoopCount));

%check contents of current location square & act

a=population(robotLoopCount).location(1,1);

b=population(robotLoopCount).location(1,2);

energy=chkSqEnergy(donutLand(a,b));

if energy==1

population(robotLoopCount)=updateEnergyFound(population(robotLoopCount));

end

corrosive=chkSqCorrosion(donutLand(a,b));

if corrosive==1

population(robotLoopCount)=updatePatchFound(population(robotLoopCount));

end

robotLoopCount=robotLoopCount+1;

end

end % end of loop for each move

%loop needed to move through locations and update square

sqCount=1;

while sqCount<9 %this updates the squares visited

%disp(locations(sqCount).loc);

% disp(locations(sqCount).loc(1,2));

if and(locations(sqCount).loc(1,1)>0,locations(sqCount).loc(1,2)>0)

sq=(locations(sqCount).loc);

c=sq(1,1);

d=sq(1,2);

donutLand(c,d)=updateSquare(donutLand(c,d));

end

sqCount = sqCount+1;

end

loopCount=loopCount+1;

end

%disp(deadRobots);

%disp(donutLand);

%disp(population(1));

%disp(population(5));

%disp(population(9));

sortedPop=arrangeByFitness(population);

disp('generations');

disp(generations);

disp(sortedPop(10));

bestFitness=sortedPop(10).steps;

tfLoop=1;

tFitness=0;

while tfLoop<11

tFitness=tFitness+sortedPop(tfLoop).steps;

tfLoop=tfLoop+1;

end

aFitness=tFitness/10;

disp(bestFitness);

disp(aFitness);

if generations==1

bestRobot=sortedPop(10);

end

if sortedPop(10).steps>bestRobot.steps

bestRobot=sortedPop(10);

else

worseCount=worseCount+1;

end

%disp(sortedPop);

child1=createChild(sortedPop);

child2=createChild(sortedPop);

disp('child1')

disp(child1)

disp('child2')

disp(child2)

sortedPop(1)=child1;

sortedPop(2)=child2;

population=sortedPop;

population=resetPopulation(population);

donutLand=resetDoLand;

generations=generations+1;

end

disp('bestRobot');

disp(bestRobot);

%disp(child1);

end

function pop=initPopulation ()

n = 1;

while (n<11) %initialises population, fields 1-10 form chromosome

field1 = 'edra'; value1 = randi([1,8],1,1); %energy depletion rate

field2 = 'maxe'; value2 = randi([0,8],1,1); %max energy

field3 = 'edre'; value3 = randi([0,50],1,1); %energy depletion reserve

field4 = 'paccra'; value4 = randi([1,8],1,1); %protective casing repair rate

field5 = 'mpc'; value5 = randi([0,8],1,1); %max protective casing

field6 = 'pcrre'; value6 = randi([0,50],1,1); %protective casing repair reserve

field7 = 'ss'; value7 = randi([0,2],1,1); %sensor strength

field8 = 'el'; value8 = {0}; %energy level

field9 = 'pc'; value9 = value5; %protective casing level

field10 = 'direction'; value10 = randi([1,8],1,1);

field11 = 'location'; value11 = [20 20];

field12 = 'alive'; value12 = {1};

field13 = 'energyRes'; value13 = value3;%enables reserve to be re-set

field14 = 'countCasing'; value14={0};

field15 = 'steps'; value15={0};

pop(n) = struct(field1,value1,field2,value2,field3,value3,field4,value4,field5,value5,field6,value6,field7,value7,field8,value8,field9,value9,field10,value10,field11,value11,field12,value12,field13,value13,field14,value14,field15,value15);

%pop(n)

n=n+1;

end

end

function dland=initDoLand()

a=1;

while (a<41)

b=1;

while (b<41)

field1 = 'energysource'; value1 = {0};

field2= 'corossivepatch'; value2={0};

dland(a,b) = struct (field1,value1,field2,value2);

dland(a,b)

b=b+1;

end

a=a+1;

end

%dland();

c=1;

while(c<350)

d = randi([1,40],1,1);

e = randi([1,40],1,1);

if(dland(d,e).corossivepatch == 0)

dland(d,e).corossivepatch = dland(d,e).corossivepatch + 1;

end

c=c+1;

end

f=1;

while(f<3501)

g = randi([1,40],1,1);

h = randi([1,40],1,1);

if(dland(g,h).corossivepatch==0)

dland(g,h).energysource = dland(g,h).energysource + 1;

end

f=f+1;

end

end

function location=initLocations()

n=1;

field1 ='loc'; value1 = [0 0];

while n<9

location(n) = struct(field1, value1);

n=n+1;

end

end

function robot=updateRobotLocation(robot)

%disp(robot.direction);

%if robot.direction(1,1)==[2]

%disp('this is a test');

%end

%decisions based on direction

if robot.direction(1,1)==[1] %N

%disp('north');

if robot.location(1)==1

robot.location(1)=40;

else

robot.location(1)=robot.location(1)-1;

end

end

if robot.direction(1,1)==[2] %NE

%disp('north\_e');

if robot.location(1)==1

robot.location(1)=40;

end

if and(robot.location(1)>1,robot.location(1)<=40)

robot.location(1)=robot.location(1)-1;

end

if robot.location(2)==40

robot.location(2)=1;

else

robot.location(2)=robot.location(2)+1;

end

end

if robot.direction(1,1)==[3] %E

%disp('east');

if robot.location(2)==40

robot.location(2)=1;

else

robot.location(2)=robot.location(2)+1;

end

end

if robot.direction(1,1)==[4] %SE

%disp('southEast');

if robot.location(1)==40

robot.location(1)=1;

else

robot.location(1)=robot.location(1)-1;

end

if robot.location(2)==40

robot.location(2)=1;

else

robot.location(2)=robot.location(2)+1;

end

end

if robot.direction(1,1)==[5] %S

%disp('south');

if robot.location(1)==40

robot.location(1)=1;

else

robot.location(1)=robot.location(1)+1;

end

end

if robot.direction(1,1)==[6] %SW

%disp('south\_west');

if robot.location(1)==40

robot.location(1)=1;

else

robot.location(1)=robot.location(1)+1;

end

if robot.location(2)==1

robot.location(2)=40;

else

robot.location(2)=robot.location(2)-1;

end

end

if robot.direction(1,1)==[7] %W

%disp('west');

if robot.location(2)==1

robot.location(2)=40;

else

robot.location(2)=robot.location(2)-1;

end

end

if robot.direction(1,1)==[8] %NW

%disp('northWest');

if robot.location(1)==1

robot.location(1)=40;

else

robot.location(1)=robot.location(1)-1;

end

if robot.location(2)==1

robot.location(2)=40;

else

robot.location(2)=robot.location(2)-1;

end

end

%error catching

if robot.location(1)<1

robot.location(1)=1;

end

if robot.location(2)>40

robot.location(1)=40;

end

if robot.location(2)<1

robot.location(2)=1;

end

if robot.location(2)>40

robot.location(2)=40;

end

end

function robot=updateRobotGeneric(robot)

if or(robot.el<0, robot.pc<0)%check life critical stats

robot.alive=0; % set robot to dead

else %robot still alive so

%update counters

robot.steps=robot.steps+1;

%check if casing needs updating

robot.pcrre=robot.pcrre-robot.paccra;

if robot.pcrre<0

if robot.pc < robot.mpc

robot.pc=robot.pc+1;

robot.el=robot.el-1;

end

robot.pcrre=robot.energyRes;

end

%check if energy levels need updating

robot.edre=robot.edre-robot.edra; %subtract depletion rate from reserve

if robot.edre<=0

robot.el=robot.el-1; %reduce energy level

robot.edre=robot.energyRes; %reset energy reserve

end

end

end

function energy=chkSqEnergy(square)

if square.energysource>0

energy=1;

else

energy=0;

end

end

function robot=updateEnergyFound(robot)

%function called if energy source found

if robot.el ~= robot.maxe

robot.el=robot.el+1;

end

end

function corrosion=chkSqCorrosion(square)

if square.corossivepatch==1

corrosion=1;

else

corrosion=0;

end

end

function robot=updatePatchFound(robot)

%updates robot if corrosive patch found

robot.pc = robot.pc-1;

end

function square=updateSquare(square)

if square.energysource >=0

square.energysource=square.energysource-1;

end

end

function robots=arrangeByFitness(robots)

i=1;

while i<8

j=1;

while j<9

if robots(j).steps > robots(j+1).steps

temp = robots(j);

robots(j)=robots(j+1);

robots(j+1)=temp;

end

j=j+1;

end

i=i+1;

end

end

function child=createChild(sortedPop)

%create child struct

field1 = 'edra'; value1 = {0}; %energy depletion rate

field2 = 'maxe'; value2 = {0}; %max energy

field3 = 'edre'; value3 = {0}; %energy depletion reserve

field4 = 'paccra'; value4 = {0}; %protective casing repair rate

field5 = 'mpc'; value5 = {0}; %max protective casing

field6 = 'pcrre'; value6 = {0}; %protective casing repair reserve

field7 = 'ss'; value7 = {0}; %sensor strength

field8 = 'el'; value8 = {0}; %energy level

field9 = 'pc'; value9 = value5; %protective casing level

field10 = 'direction'; value10 = {0};

field11 = 'location'; value11 = [20 20];

field12 = 'alive'; value12 = {1};

field13 = 'energyRes'; value13={0};%enables reserve to be re-set, set after crossover

field14 = 'countCasing'; value14={0};

field15 = 'steps'; value15={0};

child = struct(field1,value1,field2,value2,field3,value3,field4,value4,field5,value5,field6,value6,field7,value7,field8,value8,field9,value9,field10,value10,field11,value11,field12,value12,field13,value13,field14,value14,field15,value15);

%find parents for crossover

%parent1=findRouletteParent(sortedPop);

parent1=sortedPop(10);

%disp(parent1);

%parent2=findRouletteParent(sortedPop);

parent2=sortedPop(9);

%disp(parent2);

%find crossover point

crossPoint=randi([1,10],1,1);

loop=1;

%wholly inelegant but cannot think of better

while loop<crossPoint

if loop==1

child.edra=parent1.edra;

end

if loop==2

child.maxe=parent1.maxe;

end

if loop==3

child.edre=parent1.edre;

end

if loop==4

child.paccra=parent1.paccra;

end

if loop==5

child.mpc=parent1.mpc;

end

if loop==6

child.pcrre=parent1.pcrre;

end

if loop==7

child.ss=parent1.ss;

end

if loop==10

child.direction=parent1.direction;

end

loop=loop+1;

end

while loop<11

if loop==1

child.edra=parent2.edra;

end

if loop==2

child.maxe=parent2.maxe;

end

if loop==3

child.edre=parent2.edre;

end

if loop==4

child.paccra=parent2.paccra;

end

if loop==5

child.mpc=parent2.mpc;

end

if loop==6

child.pcrre=parent2.pcrre;

end

if loop==7

child.ss=parent2.ss;

end

if loop==10

child.direction=parent2.direction;

end

loop=loop+1;

end

child.energyRes=child.edre;

%cross over complete, child needs to be mutated

%disp(child)

%mutationCount=randi([1,10],1,1);

mutationCount=5;

mutLoop=1;

while mutLoop<mutationCount+1

mutateField=randi([1,10],1,1);

if mutateField==1

%child.edra=randi([1,8],1,1);

if child.edra<8

child.edra=child.edra+1;

else

child.edra=1;

end

end

if mutateField==2

%child.maxe=randi([0,8],1,1);

if child.maxe<8

child.maxe=child.maxe+1;

else

child.maxe=0;

end

end

if mutateField==3

%child.edre=randi([0,50],1,1);

if child.edre<50

child.edre=child.edre+1;

else

child.edre=0;

end

end

if mutateField==4

%child.paccra=randi([1,8],1,1);

if child.paccra<8

child.paccra=child.paccra+1;

else

child.paccra=1;

end

end

if mutateField==5

%child.mpc=randi([0,8],1,1);

if child.mpc<8

child.mpc=child.mpc+1;

else

child.mpc=1;

end

end

if mutateField==6

%child.pcrre=randi([0,50],1,1);

if child.pcrre<50

child.pcrre=child.pcrre+1;

else

child.pcrre=1;

end

end

if mutateField==7

child.ss=randi([0,2],1,1);

end

if mutateField==10

%child.direction=randi([1,8],1,1);

if child.direction<8

child.direction=child.direction+1;

else

child.direction=1;

end

end

mutLoop=mutLoop+1;

end

if child.energyRes~=child.edre

child.energyRes=child.edre;

end

child.pc=child.mpc;

% disp(child)

end

function resetPop=resetPopulation(resetPop)

n=3;

while n<11

resetPop(n).alive=1;

resetPop(n).steps=0;

resetPop(n).countCasing=0;

resetPop(n).el=0;

resetPop(n).energyRes=resetPop(n).edre;

resetPop(n).pc=resetPop(n).mpc;

n=n+1;

end

end

# Appendix 2

Roulette Parent Selection

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| BestRobot1 | BestRobot2 | BestRobot3 | BestRobot4 | BestRobot5 | BestRobot6 | BestRobot7 | BestRobot8 |
| edra: 7 | edra: 4 | edra: 2 | edra: 7 | edra: 7 | edra: 5 | edra: 7 | edra: 6 |
| maxe: 2 | maxe: 1 | maxe: 3 | maxe: 3 | maxe: 2 | maxe: 6 | maxe: 2 | maxe: 6 |
| edre: 5 | edre: 8 | edre: 1 | edre: 1 | edre: 5 | edre: 19 | edre: 5 | edre: 5 |
| paccra: 3 | paccra: 1 | paccra: 7 | paccra: 2 | paccra: 3 | paccra: 4 | paccra: 3 | paccra: 7 |
| mpc: 1 | mpc: 0 | mpc: 2 | mpc: 5 | mpc: 1 | mpc: 4 | mpc: 1 | mpc: 2 |
| pcrre: 4 | pcrre: 0 | pcrre: 1 | pcrre: 17 | pcrre: 4 | pcrre: 7 | pcrre: 4 | pcrre: 10 |
| ss: 1 | ss: 0 | ss: 0 | ss: 2 | ss: 1 | ss: 1 | ss: 1 | ss: 1 |
| el: 2 | el: 1 | el: 3 | el: 3 | el: 2 | el: -1 | el: 2 | el: 6 |
| pc: -1 | pc: -1 | pc: -1 | pc: -2 | pc: -1 | pc: 2 | pc: -1 | pc: -1 |
| direction: 4 | direction: 6 | direction: 3 | direction: 6 | direction: 4 | direction: 6 | direction: 4 | direction: 2 |
| location: [1 6] | location: [25 15] | location: [20 36] | location: [20 20] | location: [1 6] | location: [1 39] | location: [1 6] | location: [8 24] |
| alive: 0 | alive: 0 | alive: 0 | alive: 0 | alive: 0 | alive: 0 | alive: 0 | alive: 0 |
| energyRes: 19 | energyRes: 8 | energyRes: 15 | energyRes: 29 | energyRes: 19 | energyRes: 19 | energyRes: 19 | energyRes: 17 |
| countCasing: 0 | countCasing: 0 | countCasing: 0 | countCasing: 0 | countCasing: 0 | countCasing: 0 | countCasing: 0 | countCasing: 0 |
| steps: 20 | steps: 10 | steps: 15 | steps: 39 | steps: 20 | steps: 140 | steps: 20 | steps: 125 |

Elitist Parent Selection

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| BestRobot1 | BestRobot2 | BestRobot3 | BestRobot4 | BestRobot5 | BestRobot6 | BestRobot7 | BestRobot8 |
| edra: 5 | edra: 7 | edra: 1 | edra: 5 | edra: 7 | edra: 7 | edra: 7 | edra: 6 |
| maxe: 7 | maxe: 2 | maxe: 8 | maxe: 4 | maxe: 2 | maxe: 7 | maxe: 2 | maxe: 1 |
| edre: 2 | edre: 5 | edre: 10 | edre: 4 | edre: 5 | edre: 20 | edre: 5 | edre: 2 |
| paccra: 8 | paccra: 3 | paccra: 8 | paccra: 3 | paccra: 3 | paccra: 3 | paccra: 3 | paccra: 1 |
| mpc: 6 | mpc: 1 | mpc: 3 | mpc: 4 | mpc: 1 | mpc: 7 | mpc: 1 | mpc: 4 |
| pcrre: 2 | pcrre: 4 | pcrre: 2 | pcrre: 1 | pcrre: 4 | pcrre: 17 | pcrre: 4 | pcrre: 4 |
| ss: 2 | ss: 1 | ss: 1 | ss: 0 | ss: 1 | ss: 0 | ss: 1 | ss: 0 |
| el: -1 | el: 2 | el: -1 | el: 4 | el: 2 | el: 7 | el: 2 | el: 1 |
| pc: 5 | pc: -1 | pc: 3 | pc: -1 | pc: -1 | pc: -1 | pc: -1 | pc: -1 |
| direction: 3 | direction: 4 | direction: 1 | direction: 5 | direction: 4 | direction: 4 | direction: 4 | direction: 8 |
| location: [20 22] | location: [1 6] | location: [9 20] | location: [8 20] | location: [1 6] | location: [1 14] | location: [1 6] | location: [40 40] |
| alive: 0 | alive: 0 | alive: 0 | alive: 0 | alive: 0 | alive: 0 | alive: 0 | alive: 0 |
| energyRes: 2 | energyRes: 19 | energyRes: 10 | energyRes: 19 | energyRes: 19 | energyRes: 41 | energyRes: 19 | energyRes: 8 |
| countCasing: 0 | countCasing: 0 | countCasing: 0 | countCasing: 0 | countCasing: 0 | countCasing: 0 | countCasing: 0 | countCasing: 0 |
| steps: 10 | steps: 20 | steps: 310 | steps: 323 | steps: 20 | steps: 33 | steps: 20 | steps: 45 |

1. See Appendix for details of implementation. [↑](#footnote-ref-1)
2. The weighting for each member of the population was calculated by firstly summing the denary numbers 1 to 10. I then divided 360 by the sum total. This calculated value was then multiplied by the population member’s position in the array. See function rouletteParentSelection for details of implementation. [↑](#footnote-ref-2)
3. MutR = Random level of mutation, ValR = Mutated to value set randomly, Mut.5 = 50% of genes mutated, ValRef=increment of 1 on gene value, if at max it is reset to min, WC= worseCount value. [↑](#footnote-ref-3)
4. MutR = Random level of mutation, ValR = Mutated to value set randomly, Mut.5 = 50% of genes mutated, ValRef=increment of 1 on gene value, if at max it is reset to min, WC= worseCount value. [↑](#footnote-ref-4)
5. Please see Appendix 2 for details of these robots. [↑](#footnote-ref-5)