Evaluating the Performance of *E-coli* with Genetic Learning From Simulated Testing

A. Meystel, J. Andrusenko
Drexel University, Philadelphia, PA 19104

Abstract

This paper addresses the problem of finding the techniques of performance evaluation for elementary agents. From an evolutionary standpoint, the robust navigational algorithms were used by even the simplest of biological systems because the systems were able to learn how to evaluate their performance. The objective of this paper is to study one of the simplest biological, yet intelligent systems, an *E. coli* cell, and see how this could be of benefit to the design of control strategies for the single-agent intelligent systems. The robot is equipped with sensors and actuators, has a rudimentary knowledge representation system and is capable of conducting search, i.e. is equipped by the means of decision making. The robot itself is looked upon from a two-dimensional perspective and is analyzed in a computer-simulated environment. We present a design of the **Variable Structure Controller (VSC)** that combines the properties of any two structures or strategies from the ten initially available to our robot. **VSC** equipped robot should be able to come up with its own strategies of motion, without human intervention. The system under consideration supports the rudimentary learning subsystems that could be envisioned. The idea of using **Genetic Programming (GP)** is not introduced here for the sake of finding the best controller but rather for the purpose of demonstrating that improved functionality can be achieved *via on-line or simulated learning*.

Keywords: Escherichia coli; evolutionary computation; genetic algorithms; genetic programming; intelligent agents, mobile robots; motion planning; navigation, natural search

1. Genetic Programming as a Combination Mechanism in VSC

We introduce a **VSC** that combines properties of any two strategies using the principles of **Genetic Programming (GP)** [1]. The idea of using **GP** is not introduced here for the sake of finding the best controller, but rather for demonstrating that the improvement of functioning can be achieved without making a thorough investigation, and, even ON-LINE, while moving towards the goal. By thorough investigation we mean the investigation of ALL possible meaningful combinations of strategies' properties, which could be a very time consuming task. Our robot has 10 different strategies to choose from (Appendix 1). It knows how well each strategy performs in the environment it is in right now. It also knows which of the five performance criteria it wants to either minimize or maximize (Appendix 2). Lets assume that we want to maximize the efficiency ($e = [\ D_{EUC} / D_{total}\ ] * 100\ \%$). It is our desire for the robot to reach the goal while traveling along the most prferable trajectory. Under the first scenario conditions, **Experiment 1**, simply choosing Strategy 5a as the most efficient one will not lead to the efficiency optimization. Hence, we must allow our robot to somehow let its controller to evolve in order to maximize (minimize) a desired criterion.

Genetic Programming (**GP**) originated from Genetic Algorithms (**GA**s). The main difference between **GP** and **GA**s is in the way the solution to the problem is represented. **GP** creates new computer programs as the solution whereas **GA**s generate a string of numbers or some quantity that represent the solution. **GP** is a lot more powerful than **GA**s. In essence, **GP** is the key in creation of intelligent systems that program themselves.

GP can be useful in the problems where there is no ideal solution, (for example, a program that drives a car or operates a tank) [2]. Moreover, **GP** is very useful in finding solutions where the variables are constantly changing (for instance, a robot's positioning). Generally, the program will find one solution for one type of environment, while it will find an entirely different solution for another one.

Step 1 - Initial (Virtual) Population

First, an initial population of random computer programs is generated. In our case we will assume that our 10 strategies comprise the initial population. All of the computations and changes take place within a single robot's "mind".

Step 2 - Reproduction Mechanism

Then, each program (strategy) in the population is executed and assigned a fitness value according to how well it solves the problem. Our *E. coli* robot already knows how well each strategy performs in the environment it is currently in. If a

strategy performs **above** or **below** the average **depending on the performance criterion** chosen, it is considered to be "fit", and, hence, will be allowed to participate in the reproduction process. For example, if our fitness function is based upon the **efficiency criterion**, a strategy with the **efficiency criterion** above the average is considered to be "fit". However, if the fitness function is based on the **energy criterion**, a strategy with the **energy criterion** above the average is considered to be "unfit". The pseudocode for the reproduction mechanism is shown in Table 1.

Table 1: Pseudocode for the Reproduction Mechanism

```
CHOSE the Performance Criterion for optimization, PCO
FIND AVE = average (PCO_Strategy 1 PCO_Strategy 1a … PCO_Strategy 5a)
for i=1:10
  if PCO of a particular strategy > (<) AVE, name this strategy FIT
  else, name it UNFIT
  end
end
```

Step 3 – Formation of a New Population

After that, a new population of computer programs (strategies) is created. "Parents" are chosen randomly, in pairs, based upon their fitness. Two parents produce two children. The population size usually remains fixed for the duration of the search [3, 4].

The following sub-steps take place:

a) The best existing strategies are copied into a new population.

b) *Crossover*

New computer programs (strategies) are formed as a result of a ***crossover*** (sexual reproduction). In our case, during ***crossover***, the chosen "parenting" strategies swap the bottom halves of their programs (second parts) to produce two children. This process is represented below graphically:

The probability of *crossover* was chosen to be 0.6. If a randomly generated number in [0,1] interval is less than a *crossover* probability, the chosen pairs of strategies will go for crossover [5]. If *crossover* doesn't occur, the exact copies of parents are placed into the new population. The pseudocode for the process of crossover is represented in Table 2.

c) *Mutation*

New strategies are formed as a result of *mutation*. Here, we will somewhat deviate from the traditional definition of the mutation

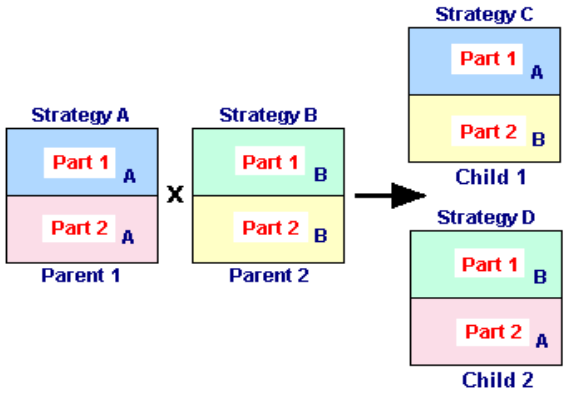

Figure 1: The Process of Crossover

mechanism to suit the design purposes of our robot's controller. First of all, in our design, it was a desire to have a *mutation* probability of 1 (usually it is preferred to have a very low mutation rate[5]). Then, we define the *mutation* operator as a change of some Control Variable Parameter's value to a randomly generated number. For instance, the average length of the robot's jump μ_J can undergo mutation when specified, i.e. μ_J will be changed to some random value. The pseudocode of the mutation process is shown below:

Table 2: Pseudocode for the Process of Crossover

```
while the formation of a new population is NOT completed
{
  Randomly choose two parents out of FIT strategies
  Generate a random number P in the [0, 1] interval
  if P < 0.6, CROSSOVER and place two children into a new population
  else, place the exact copies of parents into a new population
  end
}
CALCULATE Performance Criteria of a new population
```

Table 3: Pseudocode for the Mutation Process

while *the formation of a new population is NOT completed*

{

MUTATE a particular strategy by randomly changing a specified Control Variable Parameter

}

CALCULATE Performance Criteria of a new population

Step 4 - The Best-So-Far Solution

The best strategy that appeared in any generation, "the best-so-far solution", is designated as the result of **GP** [6].

Benefits of GP Implementation in VSC

In previous chapter, we roughly estimated the most plausible ranges of operation for the **Control Parameters**. However, for the particular scenario, we never found a specific value of each **Control Parameter** under which a specific strategy would perform the best. We have 6 **Control** Parameters, 6 sets of values per **Control Parameter**, and 10 control strategies. Under assumption that there are at least 20 values per set, we would have to perform 1200 computations! Instead of performing all 1200 computations we could simply allow our strategies to ***mutate***, let's say for 5 generations. In other words, now, we would do the same type of calculations but with 5 randomly chosen values from each set of 20. The number of computations reduces to 300. However, we are not guaranteed that these 300 computations would contain the *best solutions* (but we are hoping). Most likely, we are able to determine just improved solutions.

In a summary, what are the possible benefits of **GP** implementation into our controller? First of all, as it was mentioned earlier, we believe that it is possible to find the ***improved*** (not necessarily the best of all) solution without making a thorough investigation of all meaningful combinations of control rules. Second, with this type of controller, our robot could improve its operability while still moving towards the goal, i.e. being ON-LINE!

VSC should be able to:

- Reduce the computational complexity via **GP**, by finding better solution (best-so-far and not necessarily the best of all) faster
- Create new strategies otherwise unimaginable to humans
- Improve robot's behavior while it is still in motion towards the goal, i.e. stay ON-LINE
- Reduce the cost factor
    - All of the calculations and iterations happen inside a **single robot's "mind"** (as opposed to multiple intercommunicating agents)

When we refer to our robot being ON-LINE, we envision the following scenario: While being in ON-LINE mode, i.e. while being on its way to the goal, our robot could *locally* evaluate the **Performance Criteria** of the strategy it's currently using, every ***n*** units of time. Then, it would decide on whether to change its strategy of motion or not in accordance with the results.

Experimentations with Genetic Operators: Mutation and Crossover

1st Set of Tests: Using the Reproduction and Mutation Mechanisms only (Scenario 1)

Below is the general schema used for this particular set of tests:

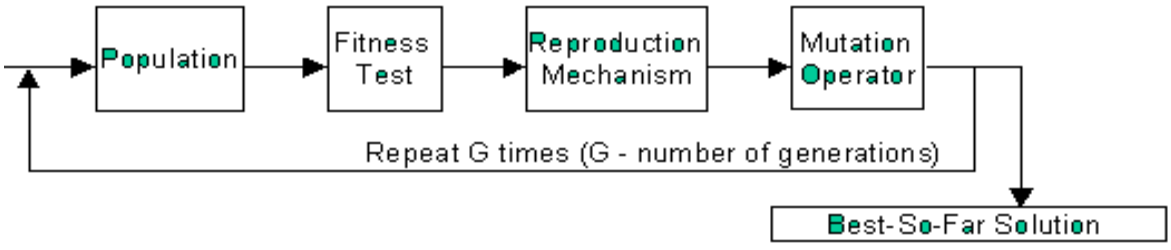

Figure 2: General Schema for the 1st Set of Tests

Table 4 describes the algorithm used for this set of tests. We chose the *efficiency* criterion to be our *Performance Criterion for optimization* ***(PCO)***, i.e. the Fitness function in the ***Reproduction*** mechanism is based upon *efficiency*. ***Mutation*** was done by the change of the average value of the random jump m_J to some random value. The reason why we chose to optimize (maximize) the efficiency via m_J is because there is a dependency of the efficiency criterion on m_J. For example, if we wanted to optimize (minimize) the *energy* criterion we would have to ***mutate*** either m_J, r or **R**. The main idea is, to make sure that there is correlation between the chosen *performance criterion* and the *control parameter* to be mutated.

Table 4: Algorithm of Actions for the 1st Set of Tests

Given: *Initial (virtual) population – 10 control strategies*

Known: *Their five Performance Criteria*

for *j=1:G (number of generations)*

CHOSE the Performance Criterion for optimization, PCO

Reproduction Mechanism (for all strategies):

FIND AVE = average ($PCO_{Strategy\ 1}$ $PCO_{Strategy\ 1a}$ … $PCO_{Strategy\ 5a}$)

for *i=1:10*

if *PCO of a particular strategy > (<) AVE, name this strategy FIT*

else*, name it UNFIT*

end

end

COPY the best existing strategy into a new population

Mutation:

while *the formation of a new population is NOT completed*

{

MUTATE a particular strategy by randomly changing a specified Control Variable Parameter

}

CALCULATE Performance Criteria of a new population

end

CHOSE the best-performed strategy from the current generation

Results:

For Scenario 1, from the initial population we can see that Strategy 5a is the most efficient one. The number of generations **G** was set to 5. Eventually, original 10 strategies were all replaced by Strategy 5a. In the 5th generation, the algorithm found the value of m_J with which the efficiency of Strategy 5a increased. In the initial population, the *efficiency* criterion (mean value of 10 runs) of Strategy 5a was found to be **63.84 %** (see Table 3.13) with m_J = **50**. However, in the 5th generation, with the mutated m_J = **40.76**, the *efficiency* of Strategy 5a increased to almost **65 %** .

2nd Set of Tests: Using the Reproduction and Mutation Mechanisms only (Scenario 2)

The only difference between this set of tests and the 1st set of tests is in the initial setup (Scenario 2). The general schema and the algorithm of actions are identical to those of the 1st set.

Results:

For Scenario 2, from the initial population we can see that Strategy 4a is the most efficient one. The number of generations **G** was again set to 5. Eventually, original 10 strategies were all replaced by Strategy 4a. In the 5th generation, the algorithm found the value of m_J with which the efficiency of Strategy 4a increased. In the initial population, the *efficiency* criterion (mean value of 10 runs) of Strategy 4a was found to be **57.94 %** with m_J = **100**. However, in the 5th generation, with the mutated m_J = **50.89** the *efficiency* of Strategy 4a increased to **62.09 %** .

Conclusion for the 1st and 2nd Sets of Tests:

From the results of 1st and 2nd sets of tests we conclude that through the sole use of the reproduction and mutation mechanisms we may find the value of the chosen control parameter under which the best-so-far strategy may perform even better.

3rd Set of Tests: Using the Reproduction and Crossover Mechanisms only (Efficiency Fitness Function, Scenario 1)

Below is the general schema for the 3rd set of tests:

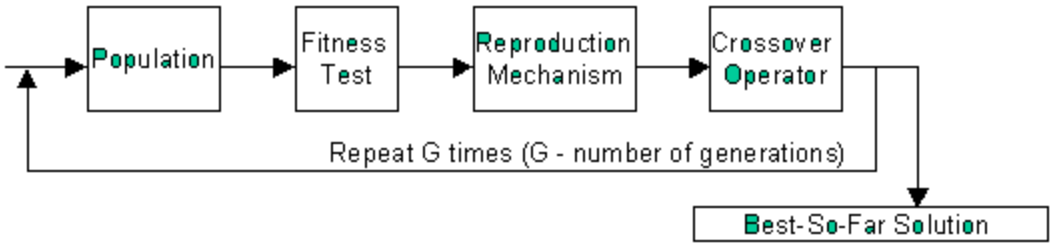

Figure 3: General Schema for the 3rd Set of Tests

This schema is described algorithmically in Table 5. Once again, we chose the *efficiency* criterion to be our ***PCO***, i.e. the Fitness function in the ***Reproduction*** mechanism is based upon *efficiency*. Since we are not changing (*mutating*) any of the control variable parameters, there should be nothing that would affect the **PCO**. The point of performing a crossover is in the fact that when we are pairing FIT parents (e.g. with efficiency above the average), we'll have a higher probability of getting an offspring with better **PCO**. However, by attempting to improve one performance criterion we might inadvertently improve others as well.

Table 5: Algorithm of Actions for the 3rd Set of Tests

Given: *Initial (virtual) population – 10 control strategies*

Known: *Their five Performance Criteria*

for *j=1:G (number of generations)*

CHOSE the Performance Criterion for optimization, PCO

Reproduction Mechanism (for all strategies):

FIND AVE = average ($PCO_{Strategy\ 1}$ $PCO_{Strategy\ 1a}$ … $PCO_{Strategy\ 5a}$)

for *i=1:10*

if *PCO of a particular strategy > (<) AVE, name this strategy FIT*

else*, name it UNFIT*

end

end

COPY the best existing strategy into a new population

Crossover:

while *the formation of a new population is NOT completed*

{

Randomly choose two parents out of FIT strategies

Generate a random number P in the [0, 1] interval

if *P < 0.6, CROSSOVER and place two children into a new population*

else*, place the exact copies of parents into a new population*

end

}

CALCULATE Performance Criteria of a new population

end

CHOSE the best-performed strategy from the current generation

Results:

In Scenario 1, from the initial population we know that Strategy 5a is the most efficient one. The number of generations **G** was set to 2. During the process of crossover Strategies 3a and 4a were chosen for mating. One of their children turned out be highly **efficient**, since it was the **efficiency** that we tried to maximize. The results of this crossover are tabulated below. Table 6 also demonstrates from which parent the child inherited this or that property. Table 7 compares the performance criteria of parents, Strategies 3a and 4a, to those of their offspring, Children 1 and 2.

From these tables one can see that, **efficiency** wise, Child 1 performed extremely well. None of the original 10 strategies, in the same scenario, could ever achieve the efficiency of **73 %** ! However, Child 2 performed quite poor in terms of **efficiency**. Nevertheless, in all of the other aspects, it performed slightly better than one of its parents, Strategy 3a. Thus, we conclude that when optimizing one performance criterion we may also inadvertently improve other criteria as well.

Table 6: 3rd Set of Tests - Results of the Crossover

Strategy	Part 1	Part 2	Control Rules Used	Supplemental Rules Used	Utilized Sensors
3a (1st move is always a jump)	If ΔC_s & ΔC_t < 0, **rotate**	If ΔC_s & ΔC_t > 0, **jump_decrease**, else, **rotate**	1, 2, 3, 4	1, 2	head, tail, belly
4a	**rotate** ***n*** times and measure all ***n*** C's; find max C out of ***n*** C's; **rotate**; find Cnew; while Cnew < max C, **rotate**	**jump_decrease**	3, 4	1, 2	belly

Child 1 of 3a & 4a	**rotate** ***n*** times and measure all ***n*** C's; find max C out of ***n*** C's; **rotate**; find Cnew; while Cnew < max C, **rotate**	If ΔC_s & $\Delta C_t > 0$, **jump_decrease**, else, **rotate**	2, 3, 4	1, 2	head, tail, belly
Child 2 of 3a & 4a (1st move is always a jump – inherited from 3a)	If ΔC_s & $\Delta C_t < 0$, **rotate**	**jump_decrease**	1, 3	1, 2	head, tail, belly

Table 7: 3rd Set of Tests - Parents' Performance vs. Children's With Efficiency Fitness Function

Strategy	Ave Time of 10 runs / Std Dev		Ave Velocity of 10 runs / Std Dev		Ave Efficiency of 10 runs / Std Dev		Ave Energy of 10 runs / Std Dev		Ave Error of 10 runs / Std Dev	
Parent 1 (3a)	46.05	21.45	13.63	4.52	45.31	14.30	33	13.67	10.32	0.74
Parent 2 (4a)	151.63	40.62	2.97	0.65	55.69	10.08	148.10	39.73	11.20	0.41
Child 1	177.91	67.192	2.02	0.7314	73.35	7.435	165.1	64.578	10.65	0.26
Child 2	41.27	12.49	14.28	2.88	42.91	11.33	26.10	9.87	9.71	0.87

In Figure 4 we compare Strategies 3a and 4a trajectories of motion to those of their "children". It is apparent that Child 1 has the highest efficiency (the thickness of the "tube" is smaller than that of others).

4th Set of Tests: Using the Reproduction and Crossover Mechanisms only (Energy Fitness Function, Scenario 1)

The general schema and the algorithm of actions are the same as in 3rd Set of Tests. For this particular set of tests we chose the *energy* criterion to be our ***PCO***, i.e. the Fitness function in the ***Reproduction*** mechanism is based upon *energy*.

Results:

For Scenario 1, from the initial population (Table 13) we know that Strategy 5a is the most efficient one. The number of generations **G** was set to 2. During the process of crossover Strategies 1a and 2a were chosen for mating. Their children turned out to be more *energy* efficient than one of their parents (remember it was the *energy* performance criterion that we tried to minimize).
The results of this crossover are tabulated below:

The comparison of performance criteria of parents, Strategies 1a and 2a, to those of their offspring, Children 1 and 2 are collected in the table:
From another table one can see that, *energy* wise, both children performed better than Parent 2 (Strategy 2a). Also, the *efficiency* criterion for both children is a lot better than that of Strategy 2a. Thus, we come to the same conclusion (see results for the 3rd Set of tests) again that when optimizing one performance criterion we can also unconsciously improve other criteria as well.

Figure 5 compares the parents' trajectories of motion to those of their offspring. Visually, it is difficult to make any sort of conclusion about strategies' performances. Even though the "tube" of trajectories for Child 2 seems to be narrower, numerically,

Strategy 1a has the highest efficiency.

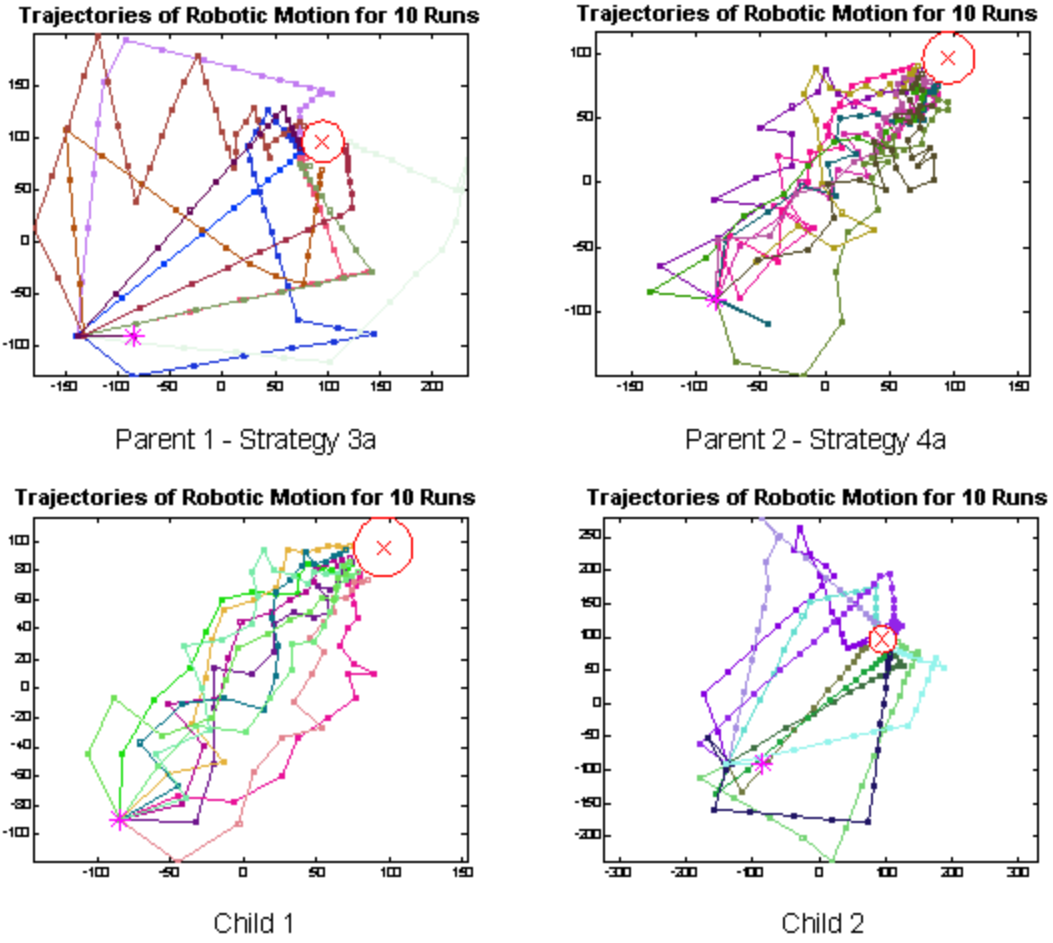

Figure 4: 3rd Set of Tests - Trajectories of Robotic Motion for Strategies 3a, 4a, and their Children

Table 8: 4th Set of Tests - Results of the Crossover

Strategy	Part 1	Part 2	Control Rules Used	Supplemental Rules Used	Utilized Sensors
1a	If $\Delta C_s < 0$, **rotate**	If $\Delta C_s > 0$, **jump_decrease**	1, 2	1, 2	head, tail
2a (1st move is always a jump)	If $\Delta C_t < 0$, **rotate**	If $\Delta C_t > 0$, **jump_decrease**	3, 4	1, 2	belly

Child 1 of 1a & 2a (1st move is always a jump)	If $\Delta C_s < 0$, **rotate**	If $\Delta C_t > 0$, **jump_decrease** (else, **rotate –** if neither of conditions is met—an additional rule we had to introduce)	1, 4	1, 2	head, tail, belly
Child 2 of 1a & 2a (1st move is always a jump)	If $\Delta C_t < 0$, **rotate**	If $\Delta C_s > 0$, **jump_decrease** (else, **rotate –** if neither of conditions is met—an additional rule we had to introduce)	2, 3	1, 2	head, tail, belly

Table 9: 4th Set of Tests - Parents' Performance vs. Children's

With Energy Fitness Function

Strategy	Ave Time of 10 runs / Std Dev		Ave Velocity of 10 runs / Std Dev		Ave Efficiency of 10 runs / Std Dev		Ave Energy of 10 runs / Std Dev		Ave Error of 10 runs / Std Dev	
Parent 1 (1a)	32.73	9.30	12.64	1.36	60.8	17.74	31.9	9.10	10.58	0.74
Parent 2 (2a)	36.1	15.18	21.88	4.29	33.96	10.64	35.2	14.78	10.09	0.52
Child 1	35.97	16.67	18.58	4.68	42.93	16.58	34.60	16.19	10.54	0.45
Child 2	34.73	9.92	14.33	3.40	50.95	11.69	33.40	9.56	10.50	1.05

Conclusion for the 3rd and 4th Sets of Tests:

From the results of 3rd and 4th sets of tests we conclude that through the sole use of the reproduction and crossover mechanisms we may find new strategies that perform better than their parents or at least one of the parents.

Operation of the Genetically Programmed VSC

Combining results from the four sets of tests analyzed above, we came up with the following design of our **Variable Structure Controller**:

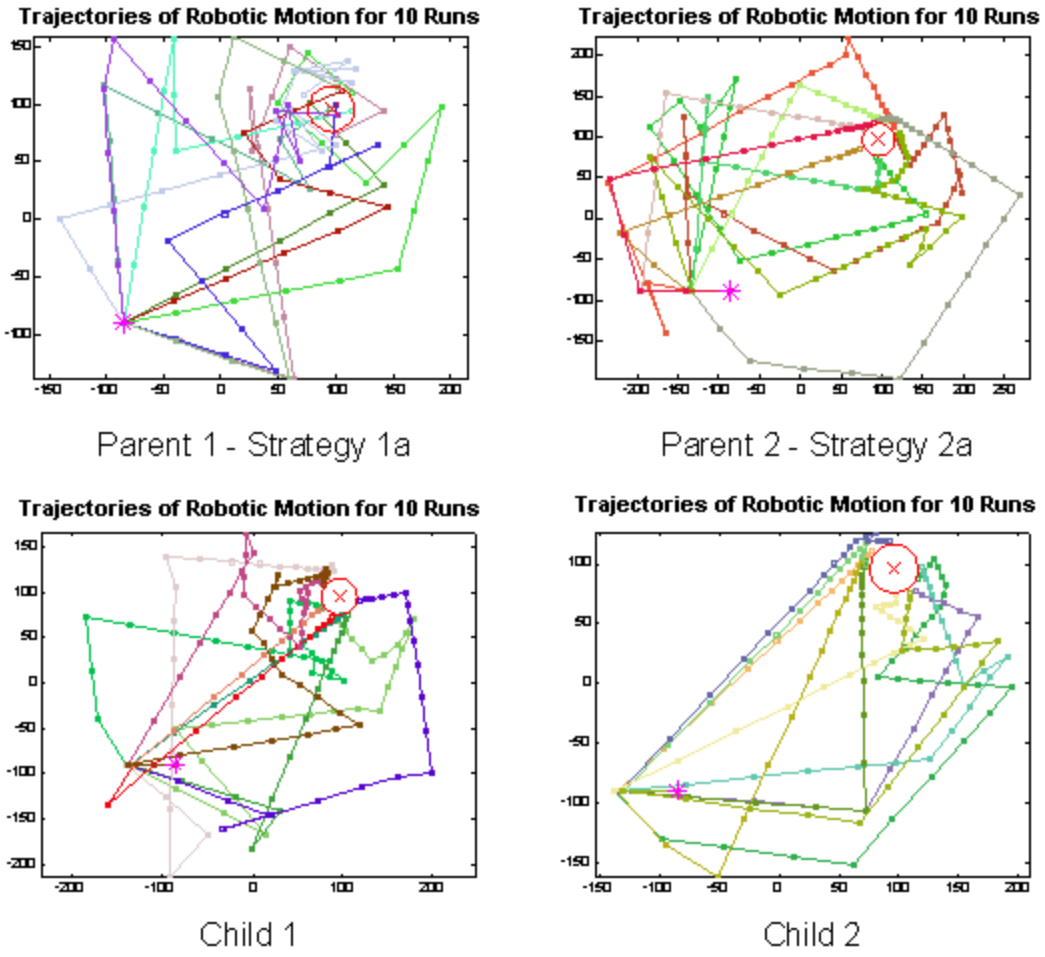

Figure 5: 4th Set of Tests - Trajectories of Robotic Motion for Strategies 1a, 2a, and their Children

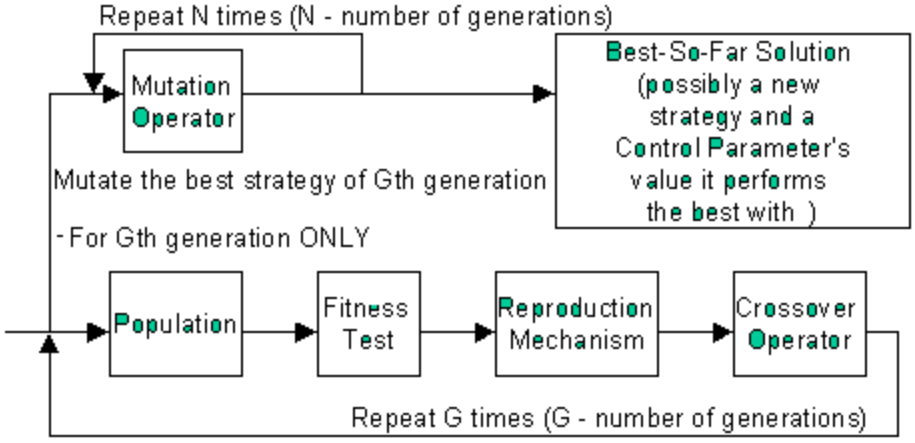

Figure 6: Variable Structure Controller's General Schema

Generally, **VSC** does the following:

- Uses the Reproduction and Crossover mechanisms for a **G** number of generations.
- It may create a new strategy that performs better than its parents or at least one of its parents. If a new strategy is created, it's placed into a new population.
- In $\mathbf{G}^{th}$ generation it chooses the best performed strategy and mutates it **N** number of times by changing some specified Control Variable Parameter to a random value.
- Outputs an IMPROVED solution in terms of the best-performed strategy and the Control Variable Parameter's value it performs the best with.

Also, we believe that if we let our controller vary the fitness function from generation to generation, it might be able to come up with a strategy that will have an improvement along more than one *performance criterion*. Below, we will describe the operation of our **VSC** algorithmically:

Table 10: Pseudocode of the VSC's Operation

Given: *Initial (virtual) population – 10 control strategies*

Known: *Their five Performance Criteria*

for *j=1:G (number of generations)*

CHOSE the Performance Criterion for optimization, PCO

Reproduction Mechanism (for all strategies):

FIND AVE = average ($PCO_{Strategy\ 1}$ $PCO_{Strategy\ 1a}$ … $PCO_{Strategy\ 5a}$)

for *i=1:10*

if *PCO of a particular strategy > (<) AVE, name this strategy FIT*

else*, name it UNFIT*

end

end

COPY the best existing strategy into a new population

Crossover:

while *the formation of a new population is NOT completed*

{

Randomly choose two parents out of FIT strategies

Generate a random number P in the [0, 1] interval

if *P < 0.6, CROSSOVER and place two children into a new population*

else*, place the exact copies of parents into a new population*

end

}

CALCULATE Performance Criteria of a new population

end

CHOSE the best-performed strategy from the current generation

Mutation:

for *k=1:N* (number of generations)

MUTATE the best-performed strategy by randomly changing a specified Control Variable Parameter

CALCULATE Performance Criteria of a mutated strategy

end

RETAIN the value of a mutated Control Variable Parameter under which the best-performed strategy performs even better

In essence, our **VSC** not only can create new strategies, it can also determine under which value of the specified **Control Variable Parameter** they perform the best.

Conclusions and Recommendations

In this paper, the following three major goals were pursued:

- To study a behavior of a real *E. coli* bacterium
- To synthesize robotic control strategies that are both efficient and robust based on the observations of *E. coli*'s behavior
- To design a robotic controller that would presume a creation of a very broad scope of logically compatible combinations of control rules comprising the earlier developed control strategies

It is worth mentioning that out of our 10 designed control strategies Strategy 2 emulates the behavior of a real *E. coli* bacterium the best, even though it is not the most robust strategy. In the figure below we compare the behavior of our robot implementing Strategy 2 to that of a real *E. coli* bacterium in a nearly isotropic homogenous medium:

The decision-making mechanism of an *E. coli* cell helped us design 10 robust control strategies. This led to the creation of a variable structure controller (**VSC**) that not only can create new strategies all on its own, but can also determine under which value of the specified **Control Variable Parameter** they perform the best.

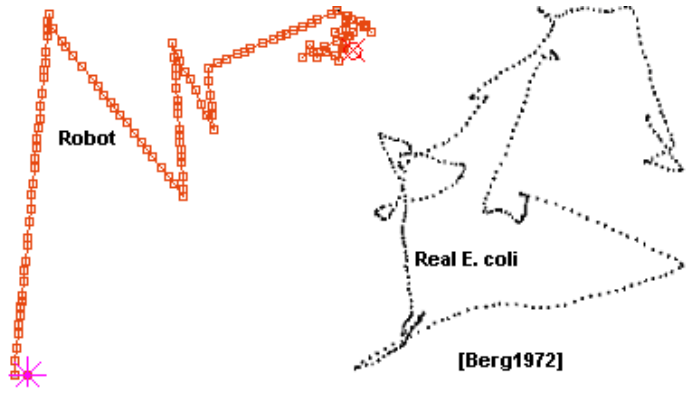

Figure 7: Robotic Trajectory of Motion (Strategy 2) vs. Real *E. coli* Bacterium's Trajectory of Motion

References

1. Andrusenko, Julia, *Biologically Inspired Variable Structure Controller (VSC) for an Autonomous Robot*, MS Thesis, Drexel University, 2001
2. Fernandez, Jaime, The GP Tutorial, last updated: June 03, 2000 http://www.geneticprogramming.com/Tutorial/index.html
3. Grefenstette, John J., *Learning Decision Strategies with Genetic Algorithms*, Naval Research Laboratory, Washington, DC, 2000
4. Aha, David W., *Tutorial on Machine Learning, 1995 AI & Statistics Workshop*, Ft. Lauderdale, FL, Jan. 1995
5. Pal, Sankar K., Wang, Paul P., *Genetic Algorithms for Pattern Recognition*, Boca Raton, FL: CRC Press, Inc., 1996
6. Koza, John R., *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, Cambridge, MA: The MIT Press, 1992

Appendix 1

Control Strategies

Strategy	Part 1	Part 2	Control Rules Used	Supplemental Rules Used	Utilized Sensors
1	If $\Delta C_s < 0$, **rotate**	If $\Delta C_s > 0$, **jump**	1, 2	1	head, tail
1a	If $\Delta C_s < 0$, **rotate**	If $\Delta C_s > 0$, **jump_decrease**	1, 2	1, 2	head, tail
2 (1st move is always a jump)	If $\Delta C_t < 0$, **rotate**	If $\Delta C_t > 0$, **jump**	3, 4	1	belly
2a (1st move is always a jump)	If $\Delta C_t < 0$, **rotate**	If $\Delta C_t > 0$, **jump_decrease**	3, 4	1, 2	belly
3 (1st move is always a jump)	If ΔC_s & $\Delta C_t < 0$, **rotate**	If ΔC_s & $\Delta C_t > 0$, **jump**, else, **rotate**	1, 2, 3, 4	1	head, tail, belly
3a (1st move is always a jump)	If ΔC_s & $\Delta C_t < 0$, **rotate**	If ΔC_s & $\Delta C_t > 0$, **jump_decrease**, else, **rotate**	1, 2, 3, 4	1, 2	head, tail, belly
4	**rotate** ***n*** times and measure all ***n*** C's; find max C out of ***n*** C's; **rotate**; find Cnew; while Cnew < max C, **rotate**	**jump**	3, 4	1	belly
4a	**rotate** ***n*** times and measure all ***n*** C's; find max C out of ***n*** C's; **rotate**; find Cnew; while Cnew < max C, **rotate**	**jump_decrease**	3, 4	1, 2	belly
5	If $\Delta C_s < 0$, **rotate**	If $\Delta C_s > 0$, **jump**, **rotate**	1, 2	1	head, tail
5a	If $\Delta C_s < 0$, **rotate**	If $\Delta C_s > 0$, **jump_decrease**, **rotate**	1, 2	1, 2	head, tail

Appendix 2

Performance criteria

Introduction of Performance Criteria

The performance criteria (for a single run) of our 10 strategies are defined as follows:

Time, **t** (sec) – total time it takes to complete a single run

Velocity, **V** (units/sec) – overall velocity, defined as a total distance traveled, $\mathbf{D_{total}}$, over total time: $\mathbf{V = D_{total} / t}$

Efficiency, e (%) – Euclidean (shortest) distance, $\mathbf{D_{EUC}}$, over total distance traveled: $\mathbf{e = [D_{EUC} / D_{total}] * 100\ \%}$. $\mathbf{D_{EUC}}$ is the distance between initial position of our robot's tail and the sugar point. For instance, for the scenario that we chose (Table 3.2), $\mathbf{D_{EUC}}$ = 232.03 units of length. The reason why we are finding distance between the robot's tail and the sugar point instead of the one between the robot's belly and the sugar point is because of the fact that our

Energy, **E** (elementary moves) – energy in this thesis is defined as a total number of elementary moves (jumps and rotations). It is assumed that both **JUMP** and **ROTATION** have a unit of energy.

Error, **Err** (% from $\mathbf{D_{EUC}}$) – error of arrival to the goal. When $\mathbf{D_{EUC}}$ is calculated there is a need to

compensate for the error of arrival to the goal. Due to the fact that it would be quite difficult for the *E. coli* robot to find a single (sugar) point, we introduced a **Stopping Rule** with its circle of radius **R** around the sugar point. Introduction of this so-called circular "sugar vicinity" also introduces an error of arrival to the goal. To compensate for that we do the following:

$$\mathbf{D_{EUC} - (h / 2 + R)},$$

where **h** is the height or length of our robot and **h /2 + R** quantity represents the maximum **Err** possible in units of length. To elaborate on what we mean by the maximum error possible we present the picture below:

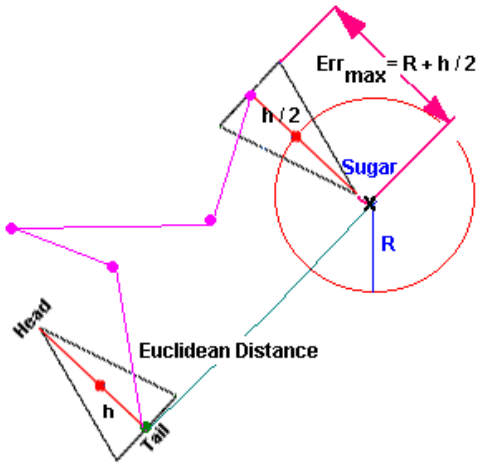

Figure A : Depiction of the Robot's Stop in the Sugar Vicinity when the Error (in units of length) of Arrival to the Goal is Maximum

Remember that the robot stops if the distance between its belly and sugar point is less or equal to **R**. Thus, the $\mathbf{Err_{max} = R + h / 2}$ since we are calculating distances from the robot's tail and not its belly.