INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI®

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

OBJECT ORIENTED APPROACH TO GENETIC PROGRAMMING

Alexandre Oumanski

A thesis
in
The Department
of
Computer Science

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Computer Science
Concordia University
Montréal, Québec, Canada

March 1999

Abstract

Object Oriented Approach to Genetic Programming

Alexandre Oumanski

Genetic algorithms (GA), first invented by John Holland, have become increasingly popular during recent years. The popularity is mostly due to the ability of the algorithms to perform a robust and efficient search in a variety of complex and noisy search spaces, where the traditional search methods fail to work. Genetic algorithms were tried on a variety of complex problems ranging from machine learning of functions to artificial life. Genetic algorithms do not work directly with the problem at hand but rather with the string-encoded representation of the problem's parameters. Thus, the representation of the problem's parameters is the important issue in the research on genetic algorithms. The most popular representations used by GA researchers are parse-tree based and machine-code based. Though very flexible, those representations tend to produce a large number of meaningless organisms.

In this thesis, we developed our own representation scheme, called "Object-Oriented Trucking" approach. Under the "Object-Oriented Trucking" approach each organism is encoded by a fixed-length string. Each entry in the string represents a C++ base class responsible for a particular type of organism's behavior. The value in the string entry selects one of the several programmer-written subclasses or strategies derived from the base class. Each organism has the same set of base classes, which guarantees that all organisms have the same and meaningful structure. However, the strategies used change from organism to organism depending on the values of the string entries. Since the structure of any given organism is not random but rather meaningful, and the programmer-written subclasses used in the organism are also supposed to be meaningful, we can guarantee that any organism would behave meaningfully. Though our approach is proofed against producing nonsense organisms, in some degree it lacks flexible - it is hard to hand-write a sufficiently large number of distinct strategies, which are necessary, if we want the algorithm to be able to discover well-performing organisms. To solve this problem we propose a so called tree-level system which is able to automatically generate a large number of strategies from a set of primitive functions.

Acknowledgments

I would like to express my deepest gratitude to my supervisor Dr. P. Grogono. His guidance, academic wisdom and encouragement made my thesis work a pleasant and extremely educational experience.

I would like to thank the "Object-Oriented Trucking team", Jeff Edelstein, Debbie Papoulis, and Andrea Gantchev. Never before have I had the opportunity to work with a better group of people. The success of "Object-Oriented Trucking team" and this thesis would not have been possible without them.

More than anyone else, I would like to thank my parents Mousia and Fima, my grandmother Ira, and my brother Ilya. It was their continued support and encouragement that made this degree possible.

Contents

List of Figures vii

List of Tables ix

1 Motivation for GA 1

1.1 Traditional Search Methods . . . 1

1.1.1 Calculus-Based Search Methods . . . 1

1.1.2 Enumerative Search Methods . . . 6

1.1.3 Random Search Methods . . . 6

1.2 Genetic Search Methods . . . 6

1.2.1 Simple Genetic Algorithm . . . 7

1.2.2 Similarity Templates . . . 10

2 Existing GA approaches 15

2.1 Hierarchical Approach . . . 15

2.1.1 Automatic Discovery of Game Strategies . . . 18

2.1.2 Weaknesses of Hierarchical Approach . . . 22

2.2 Artificial Life Approach . . . 22

2.2.1 Tierra . . . 23

2.2.2 Weaknesses of Artificial Life Approach . . . 25

3 Our Approach 26

3.1 Contributors . . . 26

3.2 Our Methodology . . . 26

3.3 The Goal of Object Oriented Trucking . . . 31

3.4 The Rationale Behind Our Methodology . . . 32

3.5 High-Level Description of the Object-Oriented Trucking Environment . . . 32

3.5.1 Trucks . . . 32

3.5.2 Dealers . . . 33
3.5.3 Initial State and Principal Parameters . . . 34
3.5.4 Scheduling . . . 35
3.5.5 Arbitrators . . . 36
3.6 Implemented Gene-Strategies . . . 36
3.6.1 Init Gene . . . 36
3.6.2 Gas Gene . . . 37
3.6.3 Trade Gene . . . 37
3.6.4 Buy Gene . . . 38
3.6.5 Sell Gene . . . 39
3.6.6 Go Gene . . . 40
3.6.7 Move Gene . . . 40
3.6.8 Deal Gene . . . 41
3.6.9 Borrow Gene . . . 44
3.6.10 Lend Gene . . . 45
3.6.11 Payer-back Gene . . . 46
3.7 Strengths of Our Approach . . . 47
3.8 Weaknesses of Our Approach . . . 47
3.8.1 Inability to Automatically Generate New Strategies . . . 47
3.8.2 Small Population Size. . . . 59

4 Conducted Experiments **60**
4.1 Theoretical Maximum . . . 60
4.1.1 Earnings Calculations . . . 62
4.2 Non-Cooperating Trucks Experiment . . . 63
4.3 Cooperating Trucks Experiment . . . 69

5 Conclusions **72**
5.1 Summary . . . 72
5.2 The Effects of my Contributions . . . 73
5.3 Future Work . . . 74

Bibliography **74**

A Base Gene Classes **76**

List of Figures

1 Unimodal Search Space. 3
2 Multi-modal Search Space. 4
3 Noisy Search Space. 5
4 Cross-over of two S-expressions. 17
5 Game Tree with Payoffs [Koz92] 19
6 Class Truck with its data members representing genes. 29
7 Data type Representing Truck's State 48
8 Auxiliary Functions for State Manipulation 49
9 Sub-genes 50
10 Sub-genes (cont.) 51
11 Sub-genes (cont.) 52
12 Sub-genes (cont.) 53
13 Gene-Level Rules 55
14 Gene-Level Rules (cont.) 56
15 Organism-Level Rules 57
16 Chromosome and Cross-Over in a Three-Level System. 58
17 Population Performance (32 generations). 66
18 Best Trucks (32 generations). 67
19 Population Performance (20 generations). 70
20 Best Trucks (20 generations). 71
21 Base classes representing genes along with the derived from them subclasses representing strategies. 77
22 Base classes representing genes along with the derived from them subclasses representing strategies. (cont.) 78
23 Base classes representing genes along with the derived from them subclasses representing strategies. (cont.) 79

24 Base classes representing genes along with the derived from them subclasses representing strategies. (cont.) 80
25 Base classes representing genes along with the derived from them subclasses representing strategies. (cont.) 81
26 Base classes representing genes along with the derived from them subclasses representing strategies. (cont.) 82
27 Base classes representing genes along with the derived from them subclasses representing strategies. (cont.) 83
28 Base classes representing genes along with the derived from them subclasses representing strategies. (cont.) 84
29 Base classes representing genes along with the derived from them subclasses representing strategies. (cont.) 85
30 Base classes representing genes along with the derived from them subclasses representing strategies. (cont.) 86
31 Base classes representing genes along with the derived from them subclasses representing strategies. (cont.) 87
32 Base classes representing genes along with the derived from them subclasses representing strategies. (cont.) 88
33 Base classes representing genes along with the derived from them subclasses representing strategies. (cont.) 89
34 Base classes representing genes along with the derived from them subclasses representing strategies. (cont.) 90
35 Base classes representing genes along with the derived from them subclasses representing strategies. (cont.) 91
36 Base classes representing genes along with the derived from them subclasses representing strategies. (cont.) 92
37 Base classes representing genes along with the derived from them subclasses representing strategies. (cont.) 93

List of Tables

1 Initial Generation. 9
2 Mating Pool and the Next Generation. 10

Chapter 1

Motivation for GA

Artificial genetic algorithms, first developed by Holland and his colleagues at the University of Michigan [Hol75], are search procedures that employ certain concepts borrowed from natural genetics. Some of the concepts include survival of the fittest, reproduction via crossover, and gene-mutation. Similar to their biological counterparts, artificial genetic algorithms possess one very important property - robustness. Analogously with the ability of biological organisms to survive in various and sometimes harsh environments, robustness in genetic algorithms signifies the ability to perform a search efficiently in a wide range of search spaces that may range from very simple to very complex ones. Before explaining why genetic algorithms are robust, it is important to first take a look at more traditional search methods and the kind of search spaces that the methods best deal with.

1.1 Traditional Search Methods

All traditional search methods can loosely be divided into three main groups: calculus-based, enumerative, and random.

1.1.1 Calculus-Based Search Methods

Calculus based method can subsequently be subdivided into indirect and direct ones.

> Indirect methods seek local extrema by solving the usually nonlinear set of equations resulting from setting the gradient of the objective function equal to zero. This is a multidimensional generalization of the elementary calculus notion of extremal points. Given a smooth, unconstrained function, finding a possible peak starts by restricting search to those points with slopes of zero in all directions [Gol89].

> On the other hand, direct methods seek local optima by hopping on the function and moving in a direction related to the local gradient. This is simply the notion of bill-climbing: to find the local best, climb the function in the steepest permissible direction[Gol89].

The calculus-based methods lack robustness because of the two shortcomings that they possess: locality in scope, and the need for existence of derivatives. Locality in scope means that the search finds a function peak in the neighborhood of the start point. Thus, if we start to search in the locality of a smaller peak, we would miss the larger one. This implies that the calculus methods would work well in the unimodal search spaces, as in Figure 1, and would perform poorly in the multi-modal spaces as in Figure 2. The need for derivative existence implies that the calculus-based methods would work only in smooth and continuous search spaces. However, the majority of real-life spaces are discontinuous, not smooth, and multi-modal, as in Figure 3.

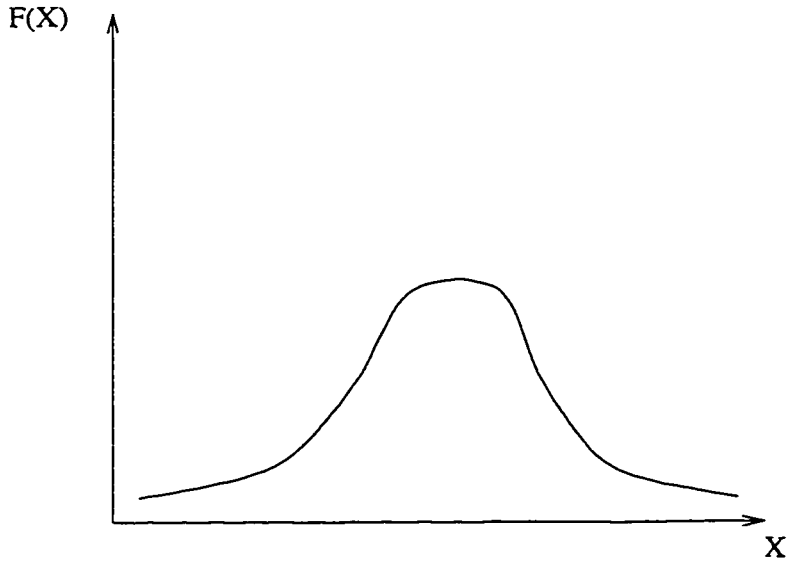


Figure 1: Unimodal Search Space.

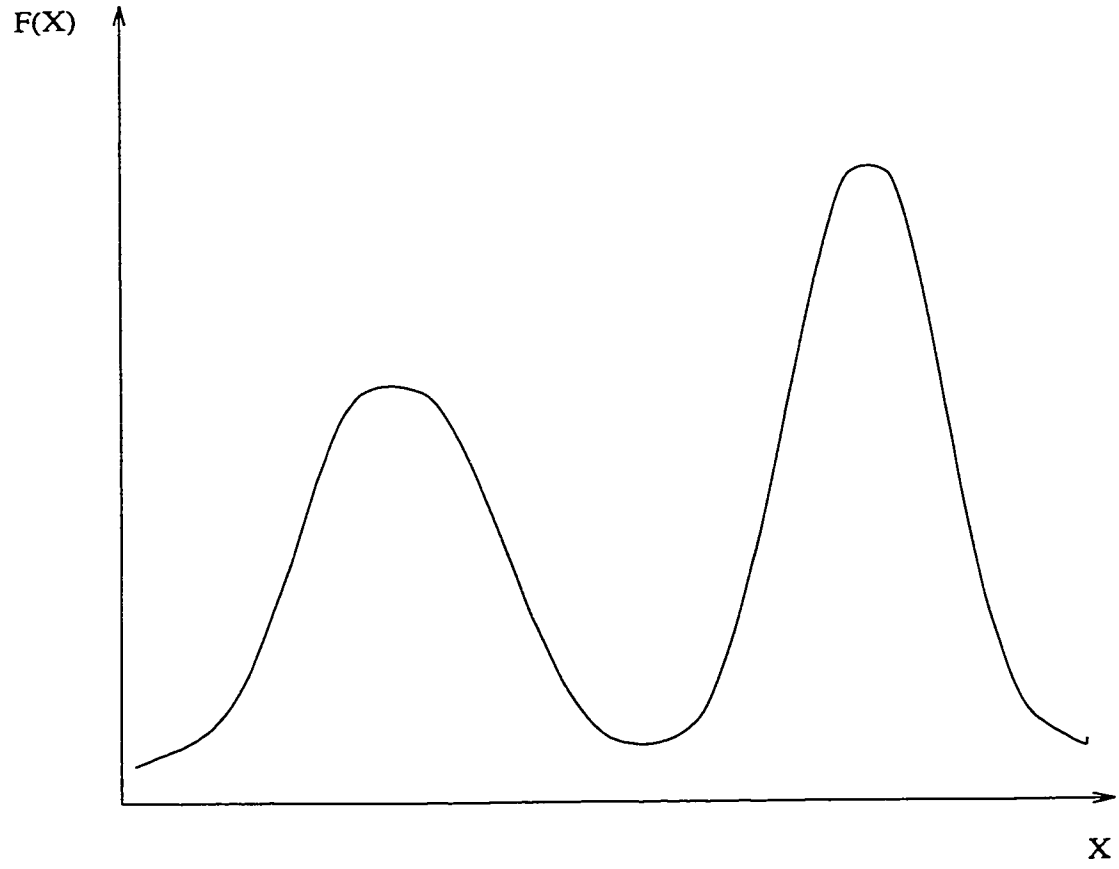
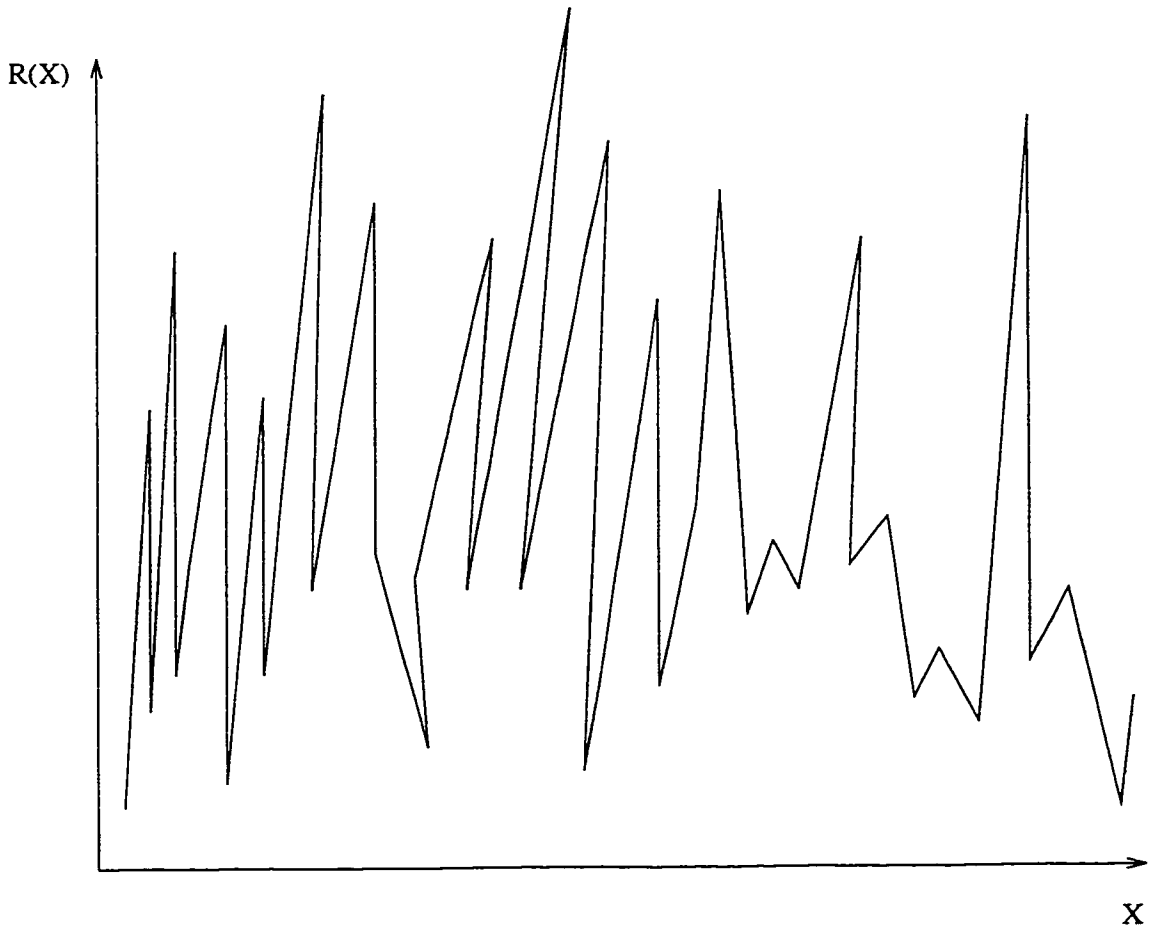


Figure 2: Multi-modal Search Space.



Note, that for the same value of X there could be more than one value of G(X). Thus, G(X) is not a function but rather a relation.

Figure 3: Noisy Search Space.

1.1.2 Enumerative Search Methods

Enumerative search methods work with the finite or discretized infinite search spaces by looking at the value of the objective function at each point in the space. Though attractive due to their simplicity, enumerative methods are not efficient when working with large spaces. A large search space is simply too big to enumerate.

1.1.3 Random Search Methods

Random search methods, similarly to enumerative schemes, lack efficiency. Though a random search, with some probability X can find a peak that is at least Y percent of the true maximum, the probability X becomes smaller as the size of the search space grows. To maintain X on the same level, as the size of the search space grows, we have to significantly increase the number of random samples. Thus, in a large space a random search is inappropriate. When talking about random methods, we must be careful in order to not confuse them with the randomized ones. Randomized methods, such as genetic algorithms, use random choice as a tool to guide a highly exploitative and directed search through a space of encoded parameters. On the other hand, random search does not have a direction.

1.2 Genetic Search Methods

Now, as we have identified the problems with robustness in the traditional search methods, it is time to show how the genetic methods cope with those problems.

The first problem is the inability to operate in multi-modal search spaces. Genetic algorithms solve this problem by having a population of start points as opposed to a single point, as this is the case with the calculus-based methods. A single start point in the neighborhood of a smaller peak would hit that peak missing the bigger one. Multiple start points have much lesser chance of missing big peaks.

The second problem is an inability to operate in non-smooth and discontinuous spaces, which do not have derivatives. Unlike the calculus-based methods, genetic algorithms do not have this problem because they do not rely on the existence of derivatives. Genetic algorithms work directly with the search parameters encoded in the gene string and do not require any auxiliary information, such as derivatives.

The third problem is an inefficiency in large search spaces. Unlike enumerative and random searches, which are undirected, a genetic search exploits the similarity information among the highly fit gene strings to guide itself towards the optimum. This guidance allows a genetic algorithm to converge without having to check each and every single point of the

search space.

In the next two sections we will show how a simple genetic algorithm works and why it can search efficiently exploiting the similarity information among the gene strings.

1.2.1 Simple Genetic Algorithm

A simple genetic algorithm consists of an encoding scheme that maps the parameter set of the problem of interest into a finite length string over a finite alphabet, and probabilistic transition rules, which do mapping from one population of strings into another. The encoding scheme may vary from problem to problem, and moreover, it is always possible to find several alternative encodings for the same problem. Usually each entry in the string or gene corresponds to one of the problem's parameters and can take on a set of values, where each value determines the characteristics of the parameter. The probabilistic transition rules consist of three operations - reproduction, crossover, and mutation.

Reproduction is a process in which individual strings are copied into the mating pool according to their objective function value. An objective function evaluates the performance of an individual encoded by the string and assigns a fitness value to that individual. The higher the fitness value the higher the probability that the individual would be selected for reproduction and would contribute genes to the next generation. A simple algorithm for reproduction would be to find the ratio of a particular string fitness value to the sum of all strings fitness values and copy that particular string into the mating pool the number of times proportional to that ratio. That is,

$$n_i = (\frac{\mathit{fitness}_i}{\sum_{j=1}^{N} \mathit{fitness}_j})N$$

where n_i is the number of copies that string i contributes to the mating pool, N is the total number of strings that we wish to maintain in a single population, and *fitness* is the fitness value associated with a string.

Crossover is a process of producing offspring strings from the pairs of parent strings. Our simple crossover consists of two steps. During the first step we randomly break the population of the mating pool into the pairs. During the second step, for each pair we randomly select a cross-over point. The cross-over point is a number in the range $1 \ldots l$, where l is the length of a string. Then, we cut both parent strings along the cross-over point, obtaining four substrings - left and right substrings of the first parent, and left and right substrings of the second parent. By concatenating the left substring of the first parent with the right substring of the second parent we obtain our first offspring. To obtain the second offspring, we concatenate the left substring of the second parent with the right substring

of the first parent. Thus, each pair of parents contributes exactly two offsprings. The fact that the fit individuals have a large number of copies in the mating pool ensures that during the crossover the fit strings would become a parent in the large number of string pairs, and thus contribute genes to the large number of offspring.

Mutation is a process of value alteration in a string position with some small probability. Genetic algorithms use mutation for two reasons. Firstly, mutation prevents a string value at a certain string position from being totally eliminated from the population. Imagine the situation when all the strings in some generation that have a value of seven at position three produced very poor fitness values. The operation of reproduction would not copy those strings into the mating pool. Thus, the value seven at position three would be lost for all subsequent generation, even though there is a chance that there exists a string that would perform well with this value. Mutation, however, can introduce back this value by mutating the value of entry three of some string into seven. Secondly, mutation prevents the algorithm from exploring one particular region of the search space and ignoring all the other regions. Random changes of gene values is equivalent to throwing the search into a different region of the search space. Occasionally changing of the search region is good because it prevents the algorithm from finding peaks in one particular region and missing possibly bigger peaks in the other regions. However, the mutation rate should be set low. Otherwise, the search would be jumping from region to region too fast without ever finding big peaks in any of the regions. Effectively, a high mutation rate would transform a genetic algorithm into a random search.

Now, as we described the three genetic operations - reproduction, crossover, and mutation, the operations can be put together into an algorithm:

```
step 1: Randomly generate a population of N strings.

step 2: Apply fitness function to the population of strings assigning
        a  fitness value to each of them.

step 3: Using the fitness values obtained in step two, create
        a mating pool via the operation of reproduction.

step 4: Using the mating pool obtained in step three, create a new
        population of N strings via the operation of crossover.
        Replace the original population by the new one.
```

step 5: Change the values of some genes in the new population via the operation of mutation. Go to step two.

The following example, adapted from [Gol89] shows the simple genetic algorithm at work (note, that we omit mutation to keep the example simple).

Suppose we want to maximize the function $f(x) = x^2$ on the interval [0..31]. That is, we want to find such an individual x from the interval [0..31] that the value of $f(x) = x^2$ is maximized. We encode each individual as a string of length five, where each entry can take on values from the set $\{0, 1\}$. The string $s = s_1s_2s_3s_4s_5$ represents a number $x = s_12^4 + s_22^3 + s_32^2 + s_42^1 + s_52^0$. That is, for the encoding we simply choose a binary representation of an unsigned integer number. To compute the fitness value of an individual string, the objective function simply has to convert the binary code of a number into the decimal and then square it. That is, the objective function has the following form:

$$o(s_1s_2s_3s_4s_5) = (s_12^4 + s_22^3 + s_32^2 + s_42^1 + s_52^0)(s_12^4 + s_22^3 + s_32^2 + s_42^1 + s_52^0).$$

In this example the size of our population is four. We start off by randomly generating four binary strings of length five. Table 1 shows these four strings along with their fitness values, a fraction of the sting's fitness over the cumulative population's fitness $f_i = \frac{f(x_i)}{\sum_{j=1}^{N} f(x_j)}$, the expected number of string copies in the mating pool f_iN, and the actual number of string copies in the mating pool. We obtain the actual number of string copies by rounding the expected number of string copies.

i	Initial Population	x	$f(x) = x^2$	$f_i = \frac{f(x_i)}{\sum_{j=1}^{N} f(x_j)}$	Expected Count f_iN	Actual Count
1	0 1 1 0 1	13	169	0.14	0.58	1
2	1 1 0 0 0	24	576	0.49	1.97	2
3	0 1 0 0 0	8	64	0.06	0.22	0
4	1 0 0 1 1	19	361	0.31	1.23	1
Sum			1170	1.00	4.00	4.00
Average			293	0.25	1.00	1.00
Max			576	0.49	1.97	2.00

Table 1: Initial Generation.

Table 2 shows the mating pool obtained by copying initial population strings according to their fitness values, mate selection, randomly generated cross-over points, new generation obtained by the cross-over, and the fitness of the new generation.

Mating Pool	Mate	Crossover point	New Population	x	$f(x)$
0 1 1 0 | 1	2	4	0 1 1 0 0	12	144
1 1 0 0 | 0	1	4	1 1 0 0 1	25	625
1 1 | 0 0 0	4	2	1 1 0 1 1	27	729
1 0 | 0 1 1	3	2	1 0 0 0 0	16	256
sum					1754
average					439
max					729

Table 2: Mating Pool and the Next Generation.

As can be seen from table 2, only after two generation the total fitness improved from 1170 to 1754, the average fitness from 293 to 439, and the best fitness from 576 to 729.

1.2.2 Similarity Templates

Now, after we have explained how the simple genetic algorithm works, we will give some mathematical foundation adapted from [Gol89] explaining why the algorithm works.

As we mentioned before, genetic algorithms use similarities in certain string positions among the highly fit strings to guide the search. For instance, we can notice that in the example above (table 1) the two most fit strings 1 1 0 0 0 and 1 0 0 1 1 are similar in the way that they both have 1's at their first positions. To understand how those similarities help the search, we first have to introduce a notion of *Similarity Template* or *Schema*. Holland in [Hol68] and [Hol75] defines schema as a similarity template describing a subset of strings with similarities at certain string positions. In our analysis, without loss of generality, we would consider only schemata for the strings defined over the binary alphabet $\{0, 1\}$. However, the same line of reasoning would hold for the strings defined over an arbitrary alphabet. Schema is a string defined over a ternary alphabet

$$\{0, 1, *\},$$

which serves as a matching device. Symbol 0 at a particular position in the schema matches all the strings that have 0 at that position, 1 matches all the strings that have 1 at that position, and * all the strings that have 0 or 1 at that position. That is, * serves as a "don't care" symbol. For example, the schema 1 * * 0 1 matches all binary strings of length five that start with 1 and end with 0 1. Thus, the strings

$$\{10001, 10101, 11001, 11101\}$$

are the four strings that the schema 1 * * 0 1 can generate. We call those four strings the examples of the schema 1 * * 0 1.

By looking at different schemata we can notice that some schema are more specific about certain similarities than the others. For example, schema 0 * 1 * * 1 matches more specific strings than schema 0 * * * * *. Furthermore, certain schemata span more of the total string length than the other. For example, schema * * 1 * * * 0 * spans over 5 positions and schema * * 1 * 0 * * * spans only over three positions. To quantify these ideas we introduce two schema properties: *schema order* and *defining length.*

> The order of a schema H, denoted by $o(H)$, is the number of fixed positions (in a binary alphabet the number of 1's and 0's) present in the template [Gol89].

For example, the order of * * 0 * * 1 * is two, and the order of * 0 1 * 0 * * is three.

> The defining length of a schema H, denoted by $\delta(H)$, is the distance between the first and the last specific string position [Gol89].

For example

$$\delta(*1***0**) = 6 - 2 = 4,$$

$$\delta(1*1*) = 3 - 1 = 2,$$

and

$$\delta(*1****) = 2 - 2 = 0.$$

With the notion of *similarity template* or *schema* in mind, we can explain formally the work of a simple genetic algorithm, or more precisely the effect of the three genetic operations - reproduction, crossover, and mutation.

First, we would determine the effect of reproduction. Suppose at the time t, or generation t, we have m examples of a particular schema H contained within the population $A(t)$ where we write

$$m = m(H, t).$$

During the reproduction a string A_i gets selected with probability

$$p_i = \frac{f_i}{\sum_j f_j},$$

contributing

$$(\frac{f_i}{\sum_j f_j})n$$

copies to the mating pool, where f_j is a fitness value of the string A_j and n is the number of strings in population $A(t)$. Since each string contributes to the mating pool

$$(\frac{f_i}{\sum_j f_j})n$$

copies, $m(H,t)$ instances of the schema H would contribute

$$m(H,t)(\frac{f(H)}{\sum_j f_j})n = m(H,t)(\frac{f(H)}{\frac{\sum_j f_j}{n}})$$

copies, where $f(H)$ is the average fitness of the strings that are instances of the schema H. Under the operation of reproduction alone, with no crossover and mutation, the population of the mating pool is equivalent to the population of the next generation $A(t+1)$. Thus, the number of instances of template H in generation $t+1$ under the operation of reproduction alone would be $m(H,t+1) = m(H,t)(\frac{f(H)}{\sum_j f_j})n = m(H,t)(\frac{f(H)}{\frac{\sum_j f_j}{n}})$. Since $\frac{\sum_j f_j}{n}$ is the average population fitness $\overline{f}$, we can rewrite the formula above as

$$m(H,t+1) = m(H,t)(\frac{f(H)}{\overline{f}})$$

The formula above shows that the number of instances of a particular schema grows or decays as the ratio of the average schema fitness to the average population fitness. In other words, schema with fitness above the population average would receive an increasing number of samples in the next generation, while schema with fitness below population average would receive a decreasing number of samples. Since the formula above holds for any schema present in the population $A(t)$, all the schemata of $A(t)$ would receive a decreasing or increasing number of samples in parallel. This parallelism is the key to the efficiency of genetic algorithms.

Suppose that the fitness of a particular schema H remains above or below the average fitness by a constant amount for all generations. Then, the fitness $f(H)$ could be rewritten as

$$f(H) = \overline{f} + c\overline{f},$$

where $c \in [-1, \infty]$ and c is a constant. Then, we can write

$$m(H,t+1) = m(H,t)\frac{\overline{f} + c\overline{f}}{\overline{f}} = m(H,t)(1+c)$$

Starting at time 0 and assuming stationary value of c we obtain the equation

$$m(H,t) = m(H,0)(1+c)^t$$

As can be seen from the formula above, with time, the above (below) average schemata receive an exponential increase (decrease) in the number of strings that represent them.

Though reproduction promotes the strings with above the average fitness, it alone does not create any new strings, and thus, does not explore any new regions of the search space. Crossover creates new string creatures with a minimum disruption to the schemata selected by reproduction. To see the effect of crossover on schemata, consider two schemata H_1 and H_2, a string A, which is an instance of H_1 and H_2, and a crossover point that is equal to three:

$$A = 011|1000$$

$$H_1 = *1*|***0$$

$$H_2 = ***|10**$$

As can be seen, the schema H_1, which has a greater *defining length*, is more likely to be disrupted in A by crossover, than the schema H_2 that has a shorter *defining length*. This happens because a cross-over point is more likely to fall between the larger span of defining positions. To quantify this observation, note that the probability of a cross-over point falling between the two extreme define positions is the ratio of the number of cross-over points between the extreme define positions and the total number of cross-over points in the string. This probability p_d is the probability of schema being destroyed. The number of cross-over points between the extreme define positions is the same as *defining length* $\delta(H)$. The number of possible cross-over points in the string is equal to the length of the string minus one: $l - 1$. Thus, the probability of schema H being destroyed during the cross-over is:

$$p_d = \frac{\delta(H)}{l-1}.$$

The probability of schema H surviving the cross-over is

$$p_s = 1 - p_d = 1 - \frac{\delta(H)}{l-1}.$$

If we assume that a pair of parents performs the cross-over occasionally with some probability p_c, the probability of being destroyed would be

$$p_d = p_c\frac{\delta(H)}{l-1}$$

and the probability of survival

$$p_s \geq 1 - p_c\frac{\delta(H)}{l-1}.$$

A schema H receives

$$m(H, t+1) = m(H, t)(\frac{f(H)}{\bar{f}})$$

copies during reproduction. Each of these copies survives the cross-over with the probability

$$p_s \geq 1 - p_c \frac{\delta(H)}{l-1}.$$

Hence, the number of copies that a schema H receives after the reproduction and cross-over is

$$m(H, t+1) \geq m(H, t)(\frac{f(H)}{\overline{f}})(1 - p_c \frac{\delta(H)}{l-1})$$

As can be seen from the expression above, schema H grows or decays depending upon a multiplication factor. With both reproduction and cross-over combined, this factor depends on two things: whether the fitness of the schema is above or below the average, and whether the schema has a relatively short or long defining length. As the defining length goes to zero, the effect of crossover on the number of schema's copies diminishes. Thus, the schemata with the fitness above average and with the relatively short *defining length* $\delta(H)$ still receive an exponential number of copies.

The last thing to consider is the effect of mutation. An individual string position mutates with a probability p_m. In order for schema H to survive, all of its define positions (positions that are 0 or 1 but not *) must survive. The probability of a position survival is $1 - p_m$. Schema H has $o(H)$ define positions. With the probability of survival at each of the $o(H)$ positions being statistically independent on each other, the probability of H surviving mutation is $(1 - p_m)^{o(H)}$. For $p_m << 1$ we can approximate $(1 - p_m)^{o(H)}$ by $1 - o(H)p_m$. Combining the results for the number of copies for schema H after the reproduction and crossover with the mutation survival probability $1 - o(H)p_m$ we would obtain $m(H, t+1) \geq m(H, t)(\frac{f(H)}{\overline{f}})(1 - p_c \frac{\delta(H)}{l-1})(1 - o(H)p_m) = m(H, t)(\frac{f(H)}{\overline{f}})(1 - p_c \frac{\delta(H)}{l-1} - o(H)p_m + \frac{\delta(H)}{l-1} o(H)p_m$. Since the term $\frac{\delta(H)}{l-1} o(H)p_m$ is very small, we can drop it obtaining:

$$m(H, t+1) \geq m(H, t)(\frac{f(H)}{\overline{f}})(1 - p_c \frac{\delta(H)}{l-1})(1 - o(H)p_m) = m(H, t)(\frac{f(H)}{\overline{f}})(1 - p_c \frac{\delta(H)}{l-1} - o(H)p_m.$$

As can be seen from the expression above, short, low order, above average schemata receive an exponential number of trials in the subsequent generations. This conclusion is called the *Schemata Theorem* or *Fundamental Theorem of Genetic Algorithms*. The *Schemata Theorem* states that the above average schemata receive an exponential number of trials in the subsequent generations. Why is this strategy good ? Holland gave an answer to this question, showing in [Hol75] that this strategy is near optimal in that it maximizes expected overall average payoff when the adaptive process (search) is viewed as a classical multi-armed slot machine problem requiring an optimal allocation of future trials given the currently available information.

Chapter 2

Existing GA approaches

In the previous chapter we gave some background on genetic algorithms. In this chapter we will examine some of the existing GA approaches and their applications.

2.1 Hierarchical Approach

One of the important themes in the GA research is the representation or encoding of the problem's parameters. Genetic algorithms directly manipulate encoded representation of a problem, therefore, inappropriate encoding schemes can severely limit the window by which the system observes its world. It has been realized by researchers that the structure of many mathematical and physical objects is too complex to be encoded in a fixed length string. This is especially true when the solution to a problem has a hierarchical structure whose shape and size are not known in advance. Steven Smith in his PHD dissertation [Smi80] departed from fixed length strings and introduced variable length strings whose entries were if-then rules, rather than single values. Other researchers, such as John Koza, continued to work in this direction giving a birth to the hierarchical approach to encoding, also known as a Genetic Programming Paradigm.

Under the Genetic Programming Paradigm, an organism is a hierarchical composition of functions and terminals appropriate to the particular problem. The hierarchy could vary in size and shape. The set of functions used by the hierarchy usually consists of arithmetical and logical operations, mathematical functions, and domain-specific functions. The domain of every function should be defined for any value in the range of any other function in the set. The set of terminals usually includes sensor inputs appropriate to the problem domain and various constants. The search space is thus comprised of all possible combinations of terminals and functions that could be recursively built into a parse tree from the available function and terminal sets. In turn, the goal of the search is to discover a computer program,

which is a parse tree consisting of functions and terminals, capable of solving the problem of interest.

The evolution of programs under the Genetic Programming Paradigm starts by a random generation of a population of parse tree like organisms. Usually, to represent a parse tree, LISP S-expressions that can very naturally represent hierarchical structures, are used. Almost always, one S-expression corresponds to one organism. After the initial population is generated, the values get assigned to the input sensor terminals in each S-expression. Then, each S-expression gets evaluated, producing the output used to compute the fitness value for the correspondent organism. Based on the computed fitness values, the operation of reproduction selects the most fit parse trees for the mating pool. Then, the mating pool is broken into pairs of parents and the operation of cross-over is applied to each pair. Cross-over proceeds by randomly selecting a node in the parse tree of the first parent and in the parse tree of the second parent. The sub-tree rooted at the selected node of the first parent is removed and replaced by the parse tree rooted at the selected node of the second parent, producing the first offspring. Similarly, the selected sub-tree of the second parent is removed and replaced by the selected sub-tree of the first parent, producing the second offspring. Such kind of sub-tree exchange is always possible due to the fact that each function in the function set is defined for any value in the range of any other function in the function set. Figure 4, adapted from [Koz92] gives an example of two S-expressions, two parse trees that the S-expressions represent, and two offspring obtained by crossing-over the S-expressions. Note, that the left offspring in Figure 4 is a perfect solution for the exclusive-or function, namely $(OR(AND(NOTD_0)(NOTD_1))(ANDD_0D_1))$. The offspring obtained during cross-over replace the original population and the process of the input terminal's initialization, fitness evaluation, reproduction, and cross-over repeats until an individual of the desired fitness level is found.

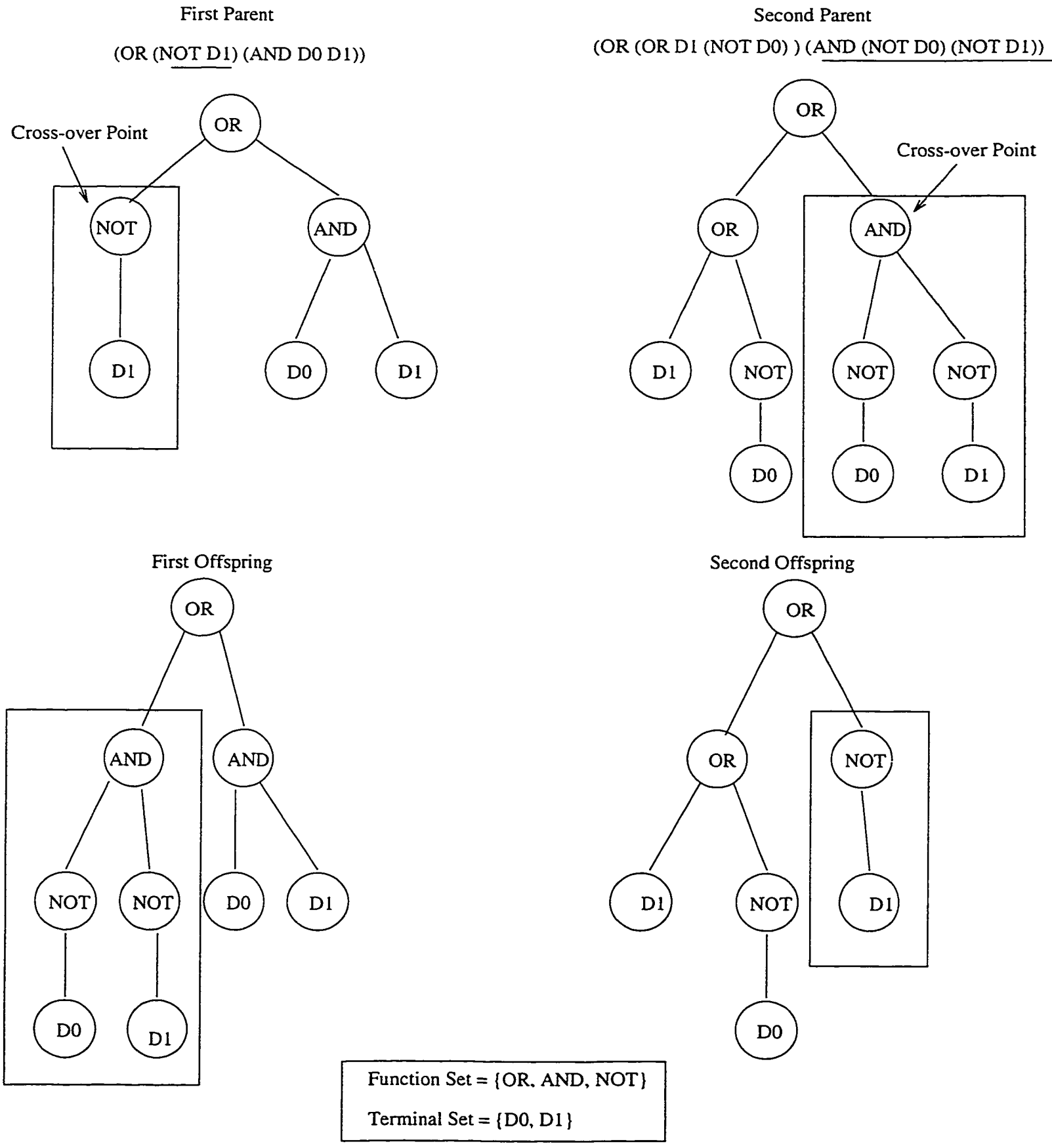


Figure 4: Cross-over of two S-expressions.

The ability of the genetic algorithms under the Genetic Programming Paradigm to automatically discover computer programs capable of solving problems in a variety of different areas has been successfully tried on a number of problems. The list below, adapted from [Koz92] and [Koz90] lists some of those problems:

- automatic programming (e.g. discovering a computational procedure for solving pairs of linear equations, solving quadratic equations for complex roots, and discovering trigonometric identities).
- empirical discovery (e.g. rediscovering Kepler's Third Law, rediscovering the well-known econometric "exchange equation" $MV = PQ$ from actual noisy time series data for the money supply, the velocity of money, the price level, and the gross national product of an economy).
- planning (e.g. navigating an artificial ant along an irregular trail, developing robotic action sequence that can stack an arbitrary initial configuration of blocks into a specified order).
- machine learning of functions (e.g. learning the Boolean 11-multiplexer function).
- sequence induction (e.g. inducing a recursive computational procedure for generating sequences such as the Fibonacci and the Hofstadter sequence).
- symbolic "data to function regression", symbolic "data to function" integration, and symbolic "data to function" differentiation.
- symbolic solution to functional equations (including differential equations with initial conditions, integral equations, and general functional equations).
- Automatic strategy discovery in simple minimax games.

2.1.1 Automatic Discovery of Game Strategies

To illustrate the Genetic Programming Paradigm, consider how in [Koz92] Koza, a prominent researcher in the area of AI at Stanford University, applies the paradigm to a problem of finding the optimal strategy for a 32-outcome two player discrete game where each player has access to a complete history of his opponent's and his own moves. In this game there are two players - X and O. The players take their turns to move. Player X makes the first move. A move could be either "go left" (L) or "go right" (R). If the move is (L), the game's current position in the game tree advances to the left child of the current node. If the move is (R), the game's current position advances to the right child. The game tree is five

branches deep, therefore, game's current position reaches a leaf node after three moves by player X and two moves by player O. The last move is always made by player X. Each of the 32 leaf nodes has a pay-off value associated with it. The goal of player X is to find a strategy that produces a sequence of three moves that maximizes the pay-off value. The goal of player O is to produce a sequence of two moves that minimizes the pay-off value. Figure 5 shows the complete game tree with the pay-off values at each leaf-node, along with the values assigned to each tree node by the optimal for this kind of problem mini-max algorithm. According to the mini-max algorithm, the most optimal strategy for player X would be to move to the left, the most optimal answer to this move by player O would be to move to the right, X should answer with the move to the right, O should answer with the move to the right, and X, at last, should answer with move to the right, attaining the pay-off value of 12. Thus, the most optimal sequence of moves for X would be $\{L, R, R\}$, and for O the most optimal sequence would be $\{R, R\}$. If player O chooses any sequence of moves other than its optimal, it is possible for player X to attain a pay-off value better than the mini-max optima of 12.

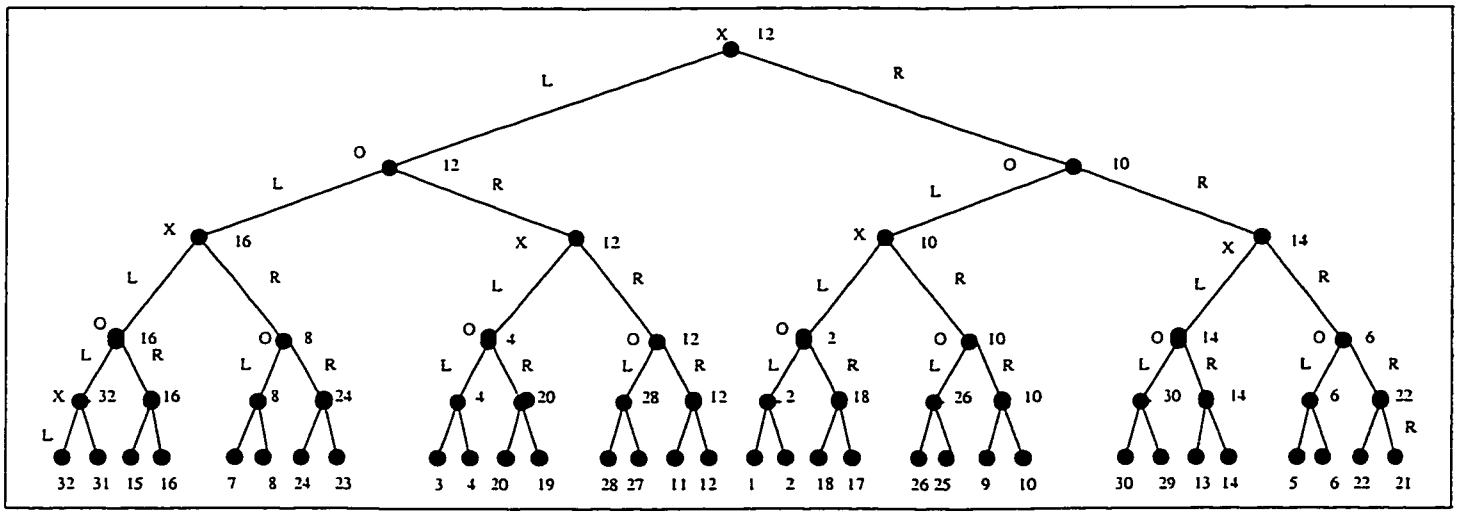


Figure 5: Game Tree with Payoffs [Koz92]

Now, as the rules of the game are stated, the next step is to genetically breed two programs, player X and player O, that take as an input a history of moves and produce the next move as an output. The history of moves is stored in four variables: XM1 - the first move of X, OM1 - the first move of O, XM2 - the second move of X, and OM2 - the second move of O. Note, that there is no need to store the history on the third move of X, because there will be no more moves after the third move, and thus, nobody would ever need this history. The values of variables XM1, OM1, XM2, and OM2 could be one the following: L, R, and U. L means that the move was made to the left, R means that the move was made to the right, U (undefined) means that the move has not yet been made. To run the game, we first set the four history variables to U (undefined). Then we run player X, which reads XM1, OM1, XM2, and OM2 and produces the first move (L or R). We store the first move in XM1 and run player O. Player O reads XM1, OM1, XM2, and OM2 and produces its first move. We store O's move in OM2 and run player X. X produces its second move. We store the move in XM2 and run player O. O in turn produces its second move, which we store in OM2. At last, we run X reaching a leaf node and reading the pay-off value. To compute a fitness value for a particular X player we run the game four times, using all four possible sequences of moves ($\{L, L\}$, $\{L, R\}$, $\{R, L\}$, and $\{R, R\}$) that an opponent O can produce. The sum of the four obtained pay-off values is the fitness value for player X. The higher the sum, the fitter player X is. Similarly, to compute a fitness value for a particular O player we run the game eight times, using all eight possible sequences of moves ($\{L, L, L\}$, $\{L, L, R\}$, $\{L, R, L\}$, $\{L, R, R\}$, $\{R, L, L\}$, $\{R, L, R\}$, $\{R, R, L\}$, and $\{R, R, R\}$) that an opponent X can produce. The sum of the eight obtained pay-off values is the fitness value for player O. The lower the sum, the fitter player O is.

The X and O players in this problem are recursively composed of the following sets of terminals and functions:

$T = \{L, R\}$ - the set of terminals

$F = \{CXM1, COM1, CXM2, COM2\}$ -the set of functions.

Function CXM1 has three arguments. Each of the three arguments evaluates to either L or R. The value returned by CXM1 is also either L or R. If the variable XM1 is undefined (U), CXM1 evaluates and returns its first argument. If the value of XM1 is L, CXM1 evaluates and returns its second argument. If the value of XM1 is R, CXM1 evaluates and returns its third argument. Similarly, function COM1 looks at the value of OM1 variable and returns its first, second, or third argument depending on that value. Functions CXM2 and COM2 work in the same way, looking at the values of XM2 and OM2 variables respectively.

First, Koza proceeds by generating X player. After six generation the genetic algorithm generates the following organism:

(COM2 (COM1 (COM1 L (CXM2 R (COM2 L L L) (CXM1 L R L))(CXM1 L L R))L R)
L (COM1 L R R))

We can simplify this computer generated expression by eliminating functions that are unreachable due to unsatisfiable conditions, and by replacing functions that always return the same terminal by the terminal that they return. The expression above simplifies into:
(COM2 (COM1 L L R) L R).
When it's the first move of player X, OM2 variable is undefined, therefore COM2 function evaluates and returns its first argument. OM1 variable is also undefined by the time of X's first move, therefore COM1 returns its first argument L, which in turn is returned by COM2. Thus, the first move of player X is always L. When it's the second move of X, variable XM1 is always L, variable OM1 is either L or R, variable OM2 is undefined, and XM2 is undefined as well. COM2 evaluates and returns its first argument. If OM1 is L, then COM1 returns its second argument L. If OM1 is R, then COM1 returns its third argument R. Thus, the second move of player X is L if the first move of player O was L, and the second move of player X is R if the first move of player O was R. When it's the third move of player X, XM1 is L, OM1 is either L or R, XM1 is either L or R, and OM2 is either L or R. If OM2 is L, COM2 evaluates its second argument and returns L. If OM2 is R, COM2 evaluates its third argument and returns R. Thus, the third move of player X is L if the second move of player O was L, and the third move of player X is R if the second move of player O was R. If we play this strategy against the opponent O's sequence $\{L, L\}$, we would get $\{L, L, L\}$ sequence with a pay-off value 32 (see Figure 5). If O's sequence is $\{L, R\}$, we would get $\{L, L, R\}$ sequence with a pay-off value 16. If O's sequence is $\{R, L\}$, we would get $\{L, R, L\}$ sequence with a pay-off value 28. If O's sequence is $\{R, R\}$, we would get $\{L, R, R\}$ sequence with a pay-off value 12. Summing up the pay-off values would give us the fitness value of X: $\mathit{fitness} = 32 + 16 + 28 + 12 = 88$. As can be seen, the organism X produced the pay-off values at least as good as the mini-max value of 12.

After obtaining a well-performing organism X, Koza switches to the process of breeding organism O. After the nine generations he obtains the following organism:
(CXM2 (CXM1 L (COM1 R L L) L)(COM1 R L (CXM2 L L R))
(COM1 L R (CXM2 R (COM1 L L R)(COM1 R L R))))
This computer generated strategy for player O simplifies to:
(CXM2 (CXM1 $ R L) L R),
Where the $ denotes a portion of an S-expression that is inaccessible by virtue of unsatisfiable conditions. This S-expression produces the following strategy: If this is the first move of player O, then move R if the first move of X was L, and move L if the first move of X

was R.If it's the second move of O, then move L if the second move of X was L, and move R if the second move of X was R. This strategy produced a fitness value of 52, and was in fact the mini-max strategy for player O.

2.1.2 Weaknesses of Hierarchical Approach

As we've seen, the Hierarchical Approach is very flexible in that it can be applied to a broad range of problems and automatically generate computer programs capable of solving those problems. However, the approach has its weaknesses. To see the weaknesses, consider the organism X generated in the previous example. The computer generated S-expression
(COM2 (COM1 (COM1 L (CXM2 R (COM2 L L L) (CXM1 L R L))(CXM1 L L R))L R)
L (COM1 L R R))
has 25 functions and terminals. However, this S-expression can be simplified to
(COM2 (COM1 L L R) L R).
The simplified S-expression has only 7 functions and terminals. From this, it can be seen that only $\frac{7}{25}100 = 28\%$ of the S-expression does the useful work, the rest is the "dead code" unreachable due to unsatisfiable conditions. In general, this is the problem with any parse tree generated by a genetic algorithm - only a small fraction of usually huge tree does the useful work, the rest is the "dead code" resulted from the unconstrained exchange of subtrees between the mating partners.

Another problem with the hierarchical approach is that the large fraction of the parse trees in any given generation does not do any useful work at all. This fraction consists of the nonsense organisms having a nonsense structure, unable to perform the task at hand. This happens because the hierarchical approach does not put strong constrains on the structure of an organism - any sub-tree can be attached to any node representing a function, as long as the type of the function's argument where we attach the sub-tree is compatible with the type of the sub-tree. However, the type compatibility constraint is not strong enough because under the hierarchical approach each function is defined for all values in the range of all the other functions, and thus, any function argument is type compatible with any sub-three.

2.2 Artificial Life Approach

Thomas Ray in [Ray95] defines the discipline of Artificial Life (AL) as follows:

> Artificial Life (AL) extends the field of biology by allowing us to study living

> forms other than those occurring naturally on Earth. Some of the most significant advances in AL have been in the area of synthetic evolutions within computers. One of the major currents in this work has been to move towards systems which evolve freely within the digital medium, like the evolution by natural selection in the carbon medium that generated life on Earth. The primary objective is to provoke digital evolution to generate complexity within the digital medium, comparable in magnitude to the complexity of organic life.

Artificial Life systems usually consist of a population of self-replicating computer programs. The programs are usually written in a machine code of a real or a software-emulated CPU. The goal of AL systems is to create a diversity of adaptable organisms and observe their evolution, rather than finding some top-performing individual. The survival of organisms is determined by the Natural Selection, which means that we don't have a hard coded objective function, but rather the fitness of an organism is determined by the organism's ability to adapt to the environment and to interact with the other organisms in the population.

Genetic variations are introduced into the population by a mutation - a random bit flip in the machine code. Since the machine code is undergone to random bit flips, it should be robust - a bit flip in a machine instruction should, with a high probability, produce another valid machine instruction.

Ray introduced Tierra, one of the most famous AL systems [Ray95], [Ray90].

2.2.1 Tierra

Tierra is a simulation environment populated by a self-replicated programs written in a special machine code, which do nothing more than copying themselves in the memory of the computer. The special machine code, invented by Ray, insures that random bit flips in a machine instruction would produce with a high probability another meaningful machine instruction. To replicate, a program needs to get a memory block from Tierra operating system, and then, to get CPU cycles to copy itself. Each program owns the block of memory that it occupies, and has an exclusive privilege of writing on its own memory block. However, any process may read, or execute the machine instructions in any part of memory. This makes the information-sharing among the organisms possible. The memory in Tierra is analogous to the physical space in biological systems, machine instructions to a genetic code, and CPU cycles to energy. During execution of the part of organism's code that does copying, the Tierra CPU can introduce, with small probability, such errors as random bit flipping, and imprecision in arithmetic and addressing operations. These errors that create

a diversity of organisms are analogous to mutations in biological systems.

At the beginning of simulation, the environment is populated with a single "ancestor" organism. It is an eighty byte machine code program that first examines itself to determine its size and location in the memory. It then, allocates a memory block and copies itself byte by byte. Then, it spawns the replica as an independent process. Each new organism tries to replicate itself. When the memory of the Tierra environment is full, the environment kills all the processes that have lived long enough, and releases their memory. Those organisms that can copy itself in a fastest possible way, capturing as many CPU cycles as possible, thrive, starving all the other organisms.

After a number of simulation runs Ray noticed that the evolving organisms use to tactics to adapt - "ecological solutions" and "optimizations".

> Ecological solutions involve interactions between the programs sharing the memory, whereas optimizations involve innovations within the individual algorithm that result in faster replication [Ray95].

Ray reports the emergence of five types of ecological solutions: parasites, immunity, hyper-parasites, cooperation, and cheaters.

- Parasites, or informational parasites, don't have their own code to replicate, but rather they execute the replicating code of other organisms populating the environment. Since the parasites don't have a replicating code, they are shorter in length than the other organisms, and thus, parasites can copy themselves faster than the other organisms.

- Immunity is the ability of hosts to evolve strategies that free them from parasites.

- Hyper-parasites exploit the parasites' tactics of using replication code of other organisms. When a parasite executes a replication code of a host, it passes the control over the CPU to the host. If the host is a hyper-parasite, it uses the CPU to copy its own code, rather than parasite's one, effectively stealing the CPU cycles from the parasite.

- Cooperation emerges when the environment is dominated by a single type of an organism. An individual of the dominant type is unable to copy itself on its own, but when the code of two or three of them is combined, the reproduction is possible.

- A cheater inserts itself among the cooperating individuals, disguising itself as one of their kind. When the control over the CPU is passed to the cheater, it uses the CPU cycles to copy itself rather then to cooperate.

During the evolution, along with the ecological solutions, several optimization techniques have emerged. First technic involves reduction of the organism's code. The best individual succeeded in reducing its size to 22 bytes, comparing to 80 bytes of the original ancestor organism. This allowed the optimized organism to copy itself six time as fast as its ancestor. Another optimization techniques involves loop unrolling. The ancestor organism has a loop that copies one byte on each iteration. The most optimized individuals managed to evolve strategies that can copy two or three bytes on each iteration.

2.2.2 Weaknesses of Artificial Life Approach

Artificial Life Approach advertises evolution of organisms within electronic medium with as little of human interference as possible. The survivability of an organism is determined only by the ability to reproduce fast, and not by a hard-coded fitness function. This is an ideal set-up for studying natural evolution, however, to solve any practical engineering problem we need some amount of control to direct the evolution towards the desired solution. A mere ability to reproduce and survive does not yet mean that an organism is a solution to the problem of interest.

Chapter 3

Our Approach

In the previous chapter we identified the existing methodologies that employ genetic algorithms, the kind of problems they were used to solve, and the shortcomings of those methodologies. In this chapter we will describe our approach to genetic programming, the kind of problems the approach was tried on, and its strengths and weaknesses.

3.1 Contributors

The original architecture of our system, which we call "Object-Oriented Trucking", was proposed by Dr. Grogono. The first version of the system, implemented by Edelstein, Papoulis, and Dr. Grogono, had 24 strategies and nine genes. I have augmented the system with the three new genes, borrow, lend, and payer-back, and implemented additional 18 strategies. In addition, I have proposed a three-level system (see pages 47 - 59), implemented adaptive search parameters (see page 64), and done a number of bug-fixes.

3.2 Our Methodology

Our approach, "Object-Oriented Trucking", which is a simulation environment consisting of a network of roads populated by competing and evolving trucks and dealers, has three major properties that distinguish it from any other approach.

The first property is that all organisms (trucks) have the same overall structure, which guarantees that any individual organism would behave meaningfully. The structure of an organism determined by its set of genes along with genes' interaction patterns does not evolve and does not vary from individual to individual. In contrast, other genetic algorithms, especially those where an organism is encoded by a parse-tree representing a fragment of code, allow the overall structure of an organism to evolve, sometimes producing organisms

with totally meaningless structures. Those genetic algorithms spend a significant amount of their time generating non-working specimens, which slows down the convergence rate of their search towards the optimal individuals. On the other hand, with our approach, all the specimens are working, though with different degree of success, which makes their evolution more directed and speeds up the convergence rate. To illustrate how we manage to have organism that both evolve and have their overall structure unchanged, consider as an example our trucks.

A truck is an organism consisting of 11 genes. Those genes are "initialize", "gas", "trade", "buy", "sell", "go", "move", "deal", "borrow", "lend", and "payer_back". When a truck gets created it calls its "initialize" gene to initialize its internal variables. At the beginning of each simulation cycle a truck first checks if it has enough gas by calling its "gas" gene, then, if necessary, it tries to borrow some money from other trucks. For this purpose it makes a call to a "borrow" gene. If a truck receives a request for money, it consults with its "lend" gene to determine the amount it should lend. After a call to a "borrow" gene, a truck initiates its trading activities by calling a "trade" gene. If a truck does not have a plan indicating where to buy and where to sell, the "trade" gene makes a call to the "deal" gene to find a profitable seller-buyer pair. If a seller-buyer pair is already found, a truck makes a call to its "buy" gene to buy from the seller and then a call to the "sell" gene to sell to the buyer. After the trading is done a truck tries to move around the country by calling its "move" gene. The "move" gene determines the direction of movement and calls the "go" gene to go in that particular direction. At the end of each simulation cycle a truck calls its "payer_back" gene to determine if it's a good time to pay any borrowed money back.

As can be seen, all trucks have the same 11 gene structure and the order of calls to these genes is the same for any truck, however, what differentiates one truck from another is the implementation, or as we call it the "strategy", that the truck uses for a particular gene. Each gene has a pool of alternative implementations, or "strategies", available for a truck to choose from. Each "strategy" is a C++ member function hand-written by a programmer. The information determining which "strategy" is used by a truck for each of its 11 genes is encoded in the truck's "chromosome". The truck's "chromosome" is a string 11 entries long. Each entry of the string corresponds to 1 of the 11 genes. The value stored in an entry selects one particular "strategy" from the pool of "strategies" of the gene that the entry represents. A "chromosome" of the form "0 2 3 4 1 0 1 8 1 0 3" , for example, would represent a truck that has "init" strategy number 0, "gas" strategy number 2, "trade" strategy number 3, "buy" strategy number 4, "sell" strategy number 1, "go" strategy number 0, "move" strategy number 1, "deal" strategy number 8, "borrow"

strategy number 1, "lend" strategy number 0, and "payer_back" strategy number 3. At the beginning of a simulation we create a number of trucks with their "chromosomes" randomly initialized. Then, we run the simulation for a time interval equal to one generation. At the end of that time interval the performance of each truck gets evaluated. To evaluate truck's performance, the objective function simply looks at the total profit made by a truck. After the performance evaluation, top scoring 45% of the trucks are selected for the reproduction and for the life in next generation. The selected 45% are broken into mate-couples by "marrying" trucks whose performance is ranked next to each other. Each mate-couple is mated two times, producing two offspring. To mate a couple, we randomly select 6 genes from the "chromosome" of the first parent, remaining 5 genes for cross-over are taken from the second parent. If, for example, the first parent has a "chromosome" "11 12 13 14 15 16 17 18 19 20 21" , the six randomly selected genes are {gene1, gene3, gene5, gene7, gene9, gene11}, and the second parent has a "chromosome" "31 32 33 34 35 36 37 38 39 40 41", then the child's "chromosome" would be "11 32 13 34 15 36 17 38 19 40 21". After the reproduction, we obtain the next generation of trucks where 45% are the top scoring trucks from the previous generation, other 45% are their offsprings, and the last 10% are randomly generated trucks. The simulation runs for a specified number of generations, trying to find well performing individuals along the way. Thus, to summarize, the overall truck's structure does not evolve, the individual "strategies" also do not evolve, what evolves is the chromosome, which represents a particular combination of "strategies".

The second property that makes our approach different from most of the others is that the smallest building block of our organisms is a "strategy" - a complete and self-contained high-level language routine capable of solving the problem that it was designed to deal with. In contrast, with the other approaches, the smallest building block is either a low-level assembly instruction (such simulation environments as "Tierra" [Ray95], [Ray90]) or a high-level language expression, which is a part of some, sometimes totally random, parse tree. Having smallest building blocks capable of solving particular subproblems increases the chance that the organism built of such blocks would perform the overall task well. Also, meaningful minimal building blocks prevent the genetic algorithm from generating a large number of non-working organisms, speeding up the evolution and making it more directed.

As was mentioned above, our smallest building blocks, "strategies", are programmer-written C++ methods. Each method is a member of some "strategy" subclass derived from one of the 11 base classes representing 11 genes. Each of the 11 base classes defines an interface by means of which the interactions with the gene represented by the base class are performed. To define the interface, a base class uses C++ virtual functions. The actual implementation of those virtual functions is done in the derived strategy subclasses.

A derived strategy subclass must implement the interface functions defined in the base class. In addition, the derived class is allowed to have any number of auxiliary functions. Figures 21 22 23 24 25 - 37 in the Appendix give the definitions of the base gene classes used in our simulation along with the derived strategy subclasses.

Class "Truck", which defines a truck specimen, has 11 data members representing 11 truck's genes (Figure 6). The type of each "gene" is a pointer to one of the 11 base gene classes. When a truck gets created, its constructor reads the strategy ids encoded in the "chromosome" passed as an argument and initializes the genes to point to the appropriate instances of strategy subclasses. Though the genes' types are pointers to the base classes, C++ late binding and virtual function mechanisms dispatch messages correctly to the appropriate strategy subclasses.

```
class Truck
{
public:
  ...
  Init *init;              // Pointer to init gene
  Gas *gas;                // Pointer to  gas gene
  Trade *trade;            // Pointer to trade gene
  Buy *buy;                // Pointer to buy gene
  Sell *sell;              // Pointer to sell gene
  Go *go;                  // Pointer to go gene
  Move *move;              // Pointer to move gene
  Deal *deal;              // Pointer to deal gene
  Borrow* borrow;          // Pointer to borrow gene
  Lend* lend;              // Pointer to lend gene
  Payer_back* payer_back;  // Pointer to payer_back gene
  ...
};
```

Figure 6: Class Truck with its data members representing genes.

The third property that puts our approach aside from most of the others has to do more with the simulation environment itself and the behavior of our organisms as a group rather than with an individual organism. The complexity of the search space presented by our simulation environment is high due to the extensive amount of interaction and competition among the organisms populating the environment, due to dependency on time of our search, and due to sensitivity of our algorithm to the initial state of our organisms and the environment. The large amount of interactions among the organisms comes from the

fact that the trucks are allowed to lend and borrow money from each other, making some earnings on the interest. The competition comes from the fact that the trucks compete with each other for getting better buy and sell prices and for being able to buy and sell desired amount of commodities. The time dependency comes form the fact that our objective function is time dependent, that is the profit made by a truck may change with each simulation cycle. The sensitivity to the initial state of the simulation is due to the fact that variations in such factors as start locations of the trucks, initial amounts of gas and money that trucks possess, initial prices, and initial rates at which prices change and commodities get produced and consumed could drastically change the results of the simulation.

To put formally the ideas above,

let $o_1, o_2, \ldots, o_N$ be N organisms that populate the environment.

Let $s_1, s_2, \ldots, s_N$ be the initial states of the organisms, where

$$s_k = \{initial_{gas,k}, initial_{money,k}, initial_{location,k}\}$$

Let s_{env} be the initial state of the environment.

Let $chr_1, chr_2, \ldots, chr_N$ be N chromosomes of our N organisms, where

$$chr_k = \{strategyForGene1_k, strategyForGene2_k, \ldots, strategyForGeneM_k\}$$

Let $t = 0 \ldots maxTimeForOneGeneration$ be the simulation time.

Let $f(chr_i, s_i, s_{env}, t, chr_1, \ldots, chr_N, s_1, \ldots, s_N)$ be the objective function that gives the profit made by an organism o_i with chromosome chr_i and initial state s_i at moment in time t, given that the initial state of the environment is s_{env}, the N organisms that participate in the simulation have chromosomes $chr_1, \ldots, chr_N$, and organisms' initial states are $s_1, \ldots, s_N$. Then, the ultimate goal of the simulation is to find an organism "o" with a chromosome "chr", such that the value of the objective function

$$f(chr, s, s_{env}, t, chr_1, \ldots, chr_N, s_1, \ldots, s_N)$$

is maximized at time $t = maxTimeForOneGeneration$, for any initial state "s" of the organism, for any initial state s_{env} of the environment, and for any genetic make-up $chr_1, \ldots, chr_N$ and the initial states $s_1, \ldots, s_N$ of other organisms that participate in the simulation. It might be difficult, if possible at all, to find an organism "o" with chromosome "chr" that maximizes value of $f()$ for any value of

$$(s, s_{env}, t, chr_1, \ldots, chr_N, s_1, \ldots, s_N)$$

tuple, so instead, it makes sense to search for an organism "o" that on average maximizes the value of $f()$ for all possible values of

$$(s, s_{env}, t, chr_1, \ldots, chr_N, s_1, \ldots, s_N),$$

that is, to search for an organism that works on average well with all possible surrounding conditions. Alternatively, we can search for a generation of organisms $\{o_1, \ldots, o_N\}$ that maximizes the value of

$$Average(f(chr_1, \ldots), f(chr_2, \ldots), \ldots, f(chr_N, \ldots))$$

The last alternative would be simply to search for a global maximum (if possible), or for a good local maximum of function $f()$, that is to search for an organism "o" that performs very well only in combination with one particular population of organisms.

As can be seen from above, the dimensions of our search space are

$$(chr_i, s_i, s_{env}, t, chr_1, \ldots, chr_N, s_1, \ldots, s_N)$$

The dimension s_{env} is presented due to the fact that the performance of an organism changes when the initial state of the environment is changed.
The dimension "t" is presented due to the fact that organism's performance changes with time.
The dimensions $(chr_1, \ldots, chr_N, s_1, \ldots, s_N)$ are presented because an organism "o" interacts and competes with other organisms, and if those other organisms are replaced by some organisms with different genetic make-up and different start conditions, the performance of the organism "o" would change.

Whereas the dimensions of our search space are

$$(chr_i, s_i, s_{env}, t, chr_1, \ldots, chr_N, s_1, \ldots, s_N),$$

most of the other genetic algorithms do not allow their specimens to interact with each other, and their simulations run for one simulation cycle, eliminating time dependency. So, their search space boils down to two (chr_i, s_i) or three (chr_i, s_i, s_{env}) dimensions, which is much simpler than our search space that grows with the number of participating organisms.

3.3 The Goal of Object Oriented Trucking

The goal of our simulation is to find a population of trucks that maximizes the system throughput. In other words, we are interested in finding a collection of trucks that can deliver goods from the producers to the consumers at as high rate as possible.

The performance of an individual truck is dependent on the behavior of the other trucks participating in the simulation. Thus, it does not make sense to search for an well-performing individual truck, for a truck performing well in a collaboration with one particular population of trucks may not perform well in a collaboration with some other

population of trucks. Instead, we are looking for a population of trucks that maximizes the throughput, using the total amount of money earned by a generation of trucks as an objective function. Since the final capital possessed by a generation of trucks is a function of the system throughput, in our implementation we use the final capital as the objective function instead of using the system throughput.

3.4 The Rationale Behind Our Methodology

There are many real-life problems where the overall problem is complex but breakable into a number of simpler subproblems, and it's easy to find several alternative solutions (strategies) for each subproblem, however it's not trivial to determine which particular combination of strategies would work together the best. For these kind of real life problems our approach, "Object-Oriented Trucking", seems to suit well, with its genes representing subproblems and its strategies representing the alternative solutions to those subproblems. Since it's easy to find several competing solutions for each subproblem, a software engineer that uses our methodology can design and implement the strategies for the subproblems by hand, and then, run the simulation trying to find a solution to the overall problem.

3.5 High-Level Description of the Object-Oriented Trucking Environment

"Object-Oriented Trucking" environment is a grid of intersecting streets and avenues, 10 by 10 in the current implementation. The grid is populated by trucks and dealers. Dealers are placed at the intersections. Each intersection can have from zero to one dealer and from zero to some maximum number of trucks. If that maximum is less than the total number of trucks, congestion is possible.

Trucks and dealers populating the country are all involved in a trading. Trading activities are done at the intersections where there are dealers.

3.5.1 Trucks

Trucks move from a dealer to a dealer, buying and selling commodities, trying to make profit on the difference between the buy and sell prices. Moving from intersection to intersection consumes truck's time and gas. When a truck is low on gas, it moves to a gas dealer to buy some gas. While traveling, a truck records an information about the intersections that it has visited. The recorded information is used by a truck to make a decision on where to

buy and where to sell, and includes such data as the kind of commodities that the visited dealers buy and sell, along with the prices. If a truck wishes to obtain an information from a remote intersection, it can make a phone call to that intersection. However, phone calls consume time and cost money. If a truck wishes to increase profit from a buy-sell transaction, it can borrow some money from another truck, paying an interest, and use the money to buy extra commodities. However, a truck has to make sure that it has enough capacity to accommodate the extra commodities, and that the profit made by selling those commodities exceeds the interest paid. If a truck receives a request for money from another truck, the former truck grants the request by lending the amount that may range from zero and up to the requested amount. If the granted amount is greater than zero, the interest rate is set to the latest rate advertised by the truck-lender prior to the request. While lending money, a truck has to make sure that the profit from using the money for a buy-sell transaction does not exceed the gain from the interest.

3.5.2 Dealers

Dealers, unlike trucks, can't move, but are rather permanently attached to their intersections. The goal of the majority of the dealers, the same as with the trucks, is to make money on the difference between the buy and sell prices. In the current implementation we have four types of dealers - producers, consumers, retailers, and gas stations.

Producers produce commodities with a constant rate and sell them to trucks in large bulks called crates.

Consumers consume commodities with a constant rate and buy them in small quantities called items (a crate consists of 20 "items").

Retailers buy from trucks in crates and sell in items, while making profit on the difference between the buy and the sell prices. If trading goes well, retailers lower their buy prices and raise their sell prices. Thus, the sooner a truck reaches a popular retailer, the better price it gets. Conversely, if the trading does not go well, retailers raise their buy prices and lower their sell prices. Thus, a truck can get a better price if it manages to find an unpopular retailer. The amount of commodities that a retailer can buy or sell is constrained by retailer's capacity and by the number of items that are currently in retailer's possession. The number of retailers in the system can change, since some of them can go bankrupt.

Gas stations sell gas to the trucks. A gas station has an unlimited supply of gas and can satisfy a request for any quantity. Gas stations also raise or lower their prices as they become more or less popular.

Producers and consumers provide the system with a flow of commodities, which enter

and leave the system with a constant rate. The purpose of retailers is to unpack crates into items, enabling the trucks to transfer the commodities between producers and consumers.

3.5.3 Initial State and Principal Parameters

Rate r_c, at which commodities pass through the system is determined by the consumption rate of consumers. This rate, which is set at the beginning of simulation and does not change, could be considered as a part of the environment's initial state. Other important parameters that do not change during the simulation and are part of the environment's initial state are items per crate k, number of consumers n_c, number of producers n_p, weight of one item w, producer's sell price a, and consumer's buy price b. Thus, the environment initial state s_{env} is a tuple

$$(k, n_c, n_p, r_c, w, a, b)$$

Important (principal) parameters that change during the simulation are the number of trucks specializing in buying from producers and selling to retailers n_{pr}, the number of trucks specializing in buying from retailers and selling to consumers n_{rc}, the number of retailers n_r, average time to deliver a commodity from producer to consumer t_{av}, a profit rate of producer-to-retailer trucks p_{pr}, a profit rate of retailer-to-consumer trucks p_{rc}, and retailer's profit rate p_r. These seven parameters control the performance of the simulation and are determined by the behavior of the organisms populating the environment. Hence, there is a mapping "m" from organisms' initial states and genetic make-ups, which determine organisms' behavior, to these parameters. Here is the mapping "m"

$$m : (chr_1, \ldots, chr_N, s_1, \ldots, s_N) \longrightarrow (n_{pr}, n_{rc}, n_r, t_{av}, p_{pr}, p_{rc}, p_r)$$

As can be seen, mapping "m" does not depend on simulation time. We can drop the time dependency if we assume that after some amount of initialization time the system enters a stable state and the principal parameters

$$(n_{pr}, n_{rc}, n_r, t_{av}, p_{pr}, p_{rc}, p_r)$$

stop changing. Since the seven parameters above determine the performance of the system and there is a mapping from the organisms' behavior to the parameters, we can approximate our objective function

$$f(chr, s, s_{env}, t, chr_1, \ldots, chr_N, s_1, \ldots, s_N)$$

by an equivalent objective function

$$g(chr, s, s_{env}, t, n_{pr}, n_{rc}, n_r, t_{av}, p_{pr}, p_{rc}, p_r)$$

Function $g()$ gives a performance of an organism "o" with chromosome "chr" and initial state "s" at moment in time t, given that initial state of the environment is

$$s_{env} = (k, n_c, n_p, r_c, w, a, b),$$

and the principal parameters of the environment in a stable state are

$$(n_{pr}, n_{rc}, n_r, t_{av}, p_{pr}, p_{rc}, p_r)$$

The benefit of having objective function $g()$ instead of $f()$ is that the search space of $g()$,

$$(chr, s, s_{env}, t, n_{pr}, n_{rc}, n_r, t_{av}, p_{pr}, p_{rc}, p_r),$$

does not grow as the number of organisms grows. On the other hand, the search space of $f()$,

$$(chr, s, s_{env}, t, chr_1, \ldots, chr_N, s_1, \ldots, s_N)$$

grows with the number of organisms. It should be noticed, that the actual simulation run attempts to find an optima of function $f()$. However, when we try to analyze the system, it is more convenient to determine the principal parameters by sampling the system, and then work with function $g()$, which is an approximation.

When applying the function $g()$ we have to make sure that the system under consideration enters a stable state with respect to the principal parameters. If this is not the case, the function $g()$ would not be a good approximation. To make sure that a system has a stable state we have to measure the principal parameters several times during the simulation. If starting with the measurement n and for all the consecutive measurements the principal parameters do not change, the system does have a stable state. Even if some principal parameter does not stabilize over the simulation, we still can use $g()$ as a good approximation. To do this, we have to use the average over the simulation time value of that unstable parameter : $p_{av} = \frac{p_1 + p_2 + \ldots + p_T}{T}$.

3.5.4 Scheduling

To divide the simulation time among the organisms fairly, the environment uses the following scheduling system: The simulation time is divided into simulation cycles. A cycle consists of ten time slots. An organism get scheduled to run for one cycle, then it's preempted, the next organism gets to run, and so on. When all organisms have run for one cycle, they get scheduled for another one, and so on, until the simulation time expires. The order of picking the next organism for a run is random, to prevent any unfairness imposed by a deterministic order. If an organism does not perform any time consuming transaction, its

simulation cycle would never expire, preventing other organisms from ever been scheduled. To avoid this kind of deadlocks, any organism that does not consume time gets preempted. In addition, if an organism runs out of money, gas, or the total allocated time, it also gets preempted.

3.5.5 Arbitrators

To ensure a fair play for all organisms, we have introduced two types of objects that serve as arbitrators. These objects, controllers and managers, enforce the rules of the simulation by adjusting according to the rules the amount of resources, such as money, time, and gas possessed by an organism, whenever the organism spends or acquires a resource. Any transaction, such as buy, sell, move, borrow, or lend involving resources, is performed through the interface provided by an arbitrator attached to an organism. In the current implementation each dealer has an attached manager, and each truck has an attached controller.

3.6 Implemented Gene-Strategies

The current version of the "Object-Oriented-Trucking" environment has 42 different strategies for its 11 genes - two strategies for "init" gene, four for "gas", four for "trade", six for "buy", five for "sell", one for "go", two for "move", nine for "deal", three for "borrow", three for "lend", and three for "payer_back". With those strategies, it's possible to build $2 * 4 * 4 * 6 * 5 * 1 * 2 * 9 * 3 * 3 * 3 = 466560$ different trucks. Below is given the description of genes and their strategies.

3.6.1 Init Gene

The purpose of the "init" gene is to initialize truck's plan and set the initial direction of movement.

Peteinit:

Initialize truck's plan as "NO PLAN", and the direction of movement as north-east.

Debinit:

Initialize truck's plan as "go to avenue 9 and street 9", and the direction of movement as north-east.

3.6.2 Gas Gene

The purpose of the "gas" gene is to find a gas station and refill the tank.

Petegas:

If we have gas enough for 20 steps, do not refill. Check the info on all visited dealers and find the closest gas station. Go to the found gas station and fill the tank to capacity.

Jeffgas:

Check the info on all visited dealers and find the closest gas station. If no gas station found, keep moving. If there is a gas station at truck's current location, fill the tank to capacity, even if there is enough gas. If the gas station is not at the current location, go there and refill, but only if we have gas for less than 20 steps.

Debgas:

Check the info on all visited dealers and find the closest gas station. If we have enough gas to reach the found gas station and make another 10 steps, do not refill. Else, go to the found gas station and refill. If the found gas station gives the best price from the seen so far, refill, even if there is enough gas. Otherwise, refill only if low on gas.

Jeff2gas:

Check the info on all visited dealers and find the closest gas station.

3.6.3 Trade Gene

The purpose of the trade gene is to dispatch trading activities.

Petetrade:

If no plan, find a deal. If plan is buying, go and buy. If plan is selling, go and sell.

Jefftrade:

If have never passed by a gas station, do nothing.

Else, if no plan, find a deal, if plan is buying, go and buy, if plan is selling, go and sell.

Debtrade:

If no plan, find a deal.

If plan is buying and there is enough simulation time left, go and buy.

If plan is selling, go and sell.

Deb2trade:

If no plan and there is enough simulation time left, find a deal.

If plan is buying, go and buy.

If plan is selling, go and sell.

3.6.4 Buy Gene

The responsibility of the "buy" gene is to go to the seller prescribed by the plan and buy there some commodities. Currently there are six "buy" strategies.

Petebuy:

Go to the seller prescribed by the plan. Buy as much as possible, leaving money to buy gas enough for one round the country journey.Set the plan's price_bought field to be equal to the value of plan's seller price field.

Jeffbuy:

Go to the seller prescribed by the plan. Buy as much as possible, leaving enough money for one full thank of gas. Set the plan's price_bought field to be equal to the value of plan's seller price field.

Debbuy:

Go to the seller prescribed by the plan. Buy as much as possible, leaving enough money for one full thank of gas. Record in the plan's price_bought field the actual price paid to the seller. This actual price could be different from the one indicated in the plan as seller's price.

Alex1Buy:

Alex1buy checks on each call to it if the buy-sell transaction from the plan is still profitable. The check is done by persistent querying of the seller and the buyer via Phone-Update call. If the transaction becomes non-profitable at some point, while the transaction is in progress, that is seller's price becomes larger than the buyer's price, the strategy aborts buying and sets the plan state to NO_PLAN, forcing the trade gene to look for another deal.

Alex2buy:

If there are no unsold CRATES or ITEMS go and buy as prescribed in the plan.

If there are unsold CRATES find a buyer who would give the best price and set the plan state to SELLING.

Else, if there are unsold ITEMS find a buyer who would give the best price and set

the plan state to SELLING.

Alex3buy:

If there are no unsold CRATES or ITEMS go and buy as prescribed in the plan.
If there are unsold CRATES find a buyer who has the capacity to buy the largest portion of unsold CRATES and would give the best price. Set the plan state to SELLING.
Else, if there are unsold ITEMS find a buyer who has the capacity to buy the largest portion of unsold ITEMS and would give the best price.

3.6.5 Sell Gene

The responsibility of the "sell" gene is to move to the buyer prescribed by the plan and sell some commodities there. Currently there are five "sell" strategies.

Petesell:

Go to the buyer prescribed by the plan and sell as much of the commodity of type (ITEM or CRATE) indicated in the plan as possible.

Debsell:

Go to the buyer prescribed by the plan and sell as much of the commodity of type (ITEM or CRATE) indicated in the plan as possible. If after the sale the truck didn't manage to sell all of the commodities of the prescribed kind, try to find another buyer within a distance of 20 steps. If such buyer found, try selling to it. If after 3 attempts to sell, still there is an unsold stock, stop trying and set the plan state from SELLING to NO_PLAN.

Deb2sell:

Go to the buyer prescribed by the plan and sell as much of the commodity of type (ITEM or CRATE) indicated in the plan as possible. If after the sale the truck didn't manage to sell all of the commodities of the prescribed kind, try to find another buyer within a distance of 20 steps. If such buyer found, try selling to it. If after the second selling there still is some stock left, try again to find a buyer within a 20 step distance. Keep trying to sell until there is stock left and able to find a buyer within a 20 step distance.

Alex1Sell:

Alex1Sell checks the buyer's price on each call to it via Phone_Update to make sure

that the buyer's price is still higher than the price bought. If the price bought becomes higher than the buyer's price, the strategy tries to find another more generous buyer.

Alex2sell:

If there is some capital and capacity to buy at least one CRATE or the equivalent to one CRATE amount of ITEMS, set the plan state to NO_PLAN asking the deal gene to find a seller to fill the truck up to the capacity. If the capacity allows to buy, but there is no money to do that, set the borrow state of the plan to BORROW_MONEY, triggering the borrow gene to borrow some money.
Else, go and sell as prescribed by the plan.

3.6.6 Go Gene

The purpose of the "go" gene is to deliver truck to the desired destination.

Petego:

If already at the destination, do nothing.
If not at the destination street, move there.
If not at the destination avenue, move there.
Update dealer info with the information about the dealer at the current location.

3.6.7 Move Gene

The purpose of the move gene is to determine the direction of movement when a truck wanders around.

Petemove:

Move along the current highway. If the end of the highway reached, turn on the perpendicular highway, follow that highway until the end, and then turn on the perpendicular highway again, and so on. Update dealer info with the information about the dealer at the current location.

Jeffmove:

If at the current location there is a buyer, the truck has something to sell, and there is no particular preferred buyer in the truck's plan, then sell to the buyer. Move along the current highway. If the end of the highway reached, turn on the perpendicular highway, follow that highway until the end and then turn on the perpendicular highway again, and so on. Update dealer info with the information about the dealer at the current location.

3.6.8 Deal Gene

The responsibility of the "deal" gene is to find a profitable seller-buyer pair and update the plan with the information about the found pair. Currently there are nine "deal" strategies.

Petedeal:

Check the information on all visited dealers. Find a pair of dealers that have the best difference between the sell and buy prices. Write into the plan the locations of found dealers, as well as the kind of commodity that they sell and buy. Set the plan state to BUY, triggering the "buy" gene to go to the seller and buy there.

Jeffdeal:

First three deals found by this strategy are crate deals, that is we by crates from a producer and sell them to a retailer. All consecutive even deals are item deals (buy from a retailer and sell to a consumer) and odd deals are crate deals. Check the number of the deal. Determine the kind of commodity to look for according to the rule above. Check the information on all visited dealers. Find a pair of dealers that have the best difference between the sell and buy prices an the commodity that we are looking for. Phone both buyer and seller to confirm that prices have not changed to the point where they do not make for a profitable deal. If deal is still profitable, write into the plan the locations of the found dealers, as well as the kind of commodity that they sell and buy. Set the plan state to BUY, triggering the "buy" gene to go to the seller and buy there.

Debdeal:

Check the information on the dealer, which is at the truck's current location. If the truck has a commodity to sell and the dealer buys this kind of commodity, giving the price better than the one paid by the truck, sell it. Else, if the dealer sells commodities, buy them. Else, no particular plan, keep moving around.

Jeff2deal:

Check the information on all visited dealers. Find a pair of dealers that have the best difference between the sell and buy prices. Calculate the distance to the dealers and make sure that the truck has enough simulation time as well as gas left to complete the trade. If gas and time is not a problem, write into the plan the locations of found dealers, as well as the kind of commodity that they sell and buy. Set the plan state to BUY, triggering the "buy" gene to go to the seller and buy there.

Deb2deal:

Check the information on all dealers visited so far. For each kind of commodities find a seller and a buyer with the best buy-sell difference. Pick a commodity with the closest located seller. If the seller is located within 20 steps, go and buy there.

Alex1deal:

For each seller s
For each buyer b
expenses = Money spent on gas to get from here to the
seller and from the seller to the buyer +
Money to buy from the seller. The quantity bought is
computed based on the truck's available capital and
capacity, and seller's available stock.
gain = Money gained from the sail. The money gained
computed based on the buyer's available capacity to buy.
profit = gain - expenses.
Pick a pair of seller s and buyer b that maximizes the profit.

Alex2deal:

For each seller s
For each buyer b
Compute the number of trucks that are closer to the seller
than my truck. Assume that the half of those trucks would buy
from the seller before I would. Thus, they would raise the sell
price of the retailer by the factor of
$1.1^{(numOfTrucksCloserThanMe \div 2)}$.
If the seller "s" is a producer, its sell price would not be affected.
expenses = Money spent on gas to get from here to the seller and
from the seller to the buyer + Money to buy from the seller.
The quantity bought is computed based on the truck's available capital
and capacity, and seller's available stock.
Compute the number of trucks that are closer to the buyer than
my truck. Assume that the half of those trucks would sell to the
buyer before I would. Thus, they would lower the buy price of the
retailer by the factor of
$0.9^{(numOfTrucksCloserThanMe \div 2)}$. If the buyer
"b" is a consumer, its buy price would not be affected.
gain = Money gained from the sail. The money gained computed

based on the buyer's available capacity to buy.
profit = gain - expenses.
Pick a pair of seller s and buyer b that maximizes the profit.

Alex3deal:

Using phone_info system call, update the information about the dealers on as many intersections as possible, leaving enough money to buy gas and to purchase some stock from a seller. Do the update every 150 units of the simulation time.

For each seller s
For each buyer b
Compute the number of trucks that are closer to the seller than my truck. Assume that the half of those trucks would buy from the seller before I would. Thus, they would raise the sell price of the retailer by the factor of $1.1^{(numOfTrucksCloserThanMe \div 2)}$.
If the seller "s" is a producer, its sell price would not be affected.
expenses = Money spent on gas to get from here to the seller and from the seller to the buyer + Money to buy from the seller. The quantity bought is computed based on the truck's available capital and capacity, and seller's available stock.
Compute the number of trucks that are closer to the buyer than my truck. Assume that the half of those trucks would sell to the buyer before I would. Thus, they would lower the buy price of the retailer by the factor of $0.9^{(numOfTrucksCloserThanMe \div 2)}$.
If the buyer "b" is a consumer, its buy price would not be affected.
gain = Money gained from the sail. The money gained computed based on the buyer's available capacity to buy.
profit = gain - expenses.
Pick a pair of seller s and buyer b that maximizes the profit.

Alex4deal:

Using phone_info system call, update the information about the dealers on as many intersections as possible. Phone_info costs money, thus leave enough money to buy gas and to purchase some stock from a seller. Do the update every 150 units of the simulation time.

For each seller s

For each buyer b

expenses = Money spent on gas to get from here to the seller and from the seller to the buyer + Money to buy from the seller. The quantity bought is computed based on the truck's available capital and capacity, and seller's available stock.

gain = Money gained from the sail. The money gained computed based on the buyer's available capacity to buy.

profit = gain - expenses.

Pick a pair of seller s and buyer b that maximizes the profit.

3.6.9 Borrow Gene

The responsibility of Borrow gene is to select the potential creditors from the list of the controllers known to the truck, to decide depending on situation on how much to borrow from a particular creditor, and to ask truck's controller to borrow money from the selected creditors. To actually borrow the money the Borrow gene uses Control::borrow method. If a truck manages to borrow some money from a creditor, the borrow gene must add the creditor to the truck's list of creditors, so that later the Payer_back gene could find the creditor and pay back to it. Currently there are three "borrow" strategies.

Alex1Borrow:

If truck's plan state is BUYING:

From the intersections that the truck has seen so far the method selects all the controllers whose interest rate is lower than the profit rate from buying from plan's seller and selling to plan's buyer. Then, the method tries to borrow as much as possible from each selected creditor. Each creditor that actually lent any money is added to the creditor list. If truck's plan state is BORROW_MONEY:

From the intersections that the truck has seen so far the method selects all the controllers whose interest rate is lower than 1.1 times truck's own interest rate. Then, the method tries to borrow as much as possible from each selected creditor. Each creditor that actually lent any money is added to the creditor list.

Alex2Borrow:

If truck's plan state is BUYING:

From the intersections that the truck has seen so far the method selects all the controllers whose interest rate is lower than the profit rate from buying from plan's seller and selling to plan's buyer. Then, the method tries to borrow as much as possible from one of the selected creditors. As soon as any one of the selected creditors actually lends anything, the truck does not continue to ask the rest of the selected creditors

for money. The creditor that lent the money is added to the creditor list. If truck's plan state is BORROW_MONEY:

From the intersections that the truck has seen so far the method selects all the controllers whose interest rate is lower than 1.02 times truck's own interest rate. Then, the method tries to borrow as much as possible from one of the selected creditors. As soon as any one of the selected creditors actually lends anything, the truck does not continue to ask the rest of the selected creditors for money. The creditor that lent the money is added to the creditor list.

Alex3Borrow:

Do not borrow any money ever.

3.6.10 Lend Gene

The responsibility of the "lend" gene is to decide on how much to lend when asked for money by other truck. Also, the Lend gene updates the interest rate in accordance to some policy. Currently there are three "lend" strategies.

Alex1Lend:

If it is almost the end of simulation or the requester is low on gas, do not lend any money, else if plan state is BUYING and the profit rate from buying and selling is higher than the current interest rate, do not lend any money, else lend truck's capita minus GAS_PRICE * GAS_MOVE * (NUM_AV + NUM_ST).

Update interest rate policy:

If number of borrow requests to the truck in the previous time slot is less than the number of borrow requests in the current time slot increase the interest rate by the factor of 1.2

If number of borrow requests to the truck in the previous time slot is greater than the number of borrow requests in the current time slot decrease the interest rate by the factor of 0.9

If the plan state is BUYING and the current interest rate is lower than the profit rate from buying and selling, make the interest rate equal to the profit rate o buying and selling.

Alex2Lend:

If plan state is BUYING, do not lend any money,

else if it's almost the end of simulation or the requester is low on gas, do not lend any money,

else available_money = truck's capital - GAS_PRICE * GAS_MOVE * (NUM_AV + NUM_ST)

if current interest rate less than 0.001, lend 25available_money.

else if current interest rate less than 0.01, lend 50available_money.

else if current interest rate less than 0.1, lend 75available_money.

else lend 100Update interest rate policy:

If number of borrow requests to the truck in the previous time slot is less than the number of borrow requests in the current time slot increase the interest rate by the factor of 1.1

If number of borrow requests to the truck in the previous time slot is greater than the number of borrow requests in the current time slot decrease the interest rate by the factor of 0.8

Alex3Lend:

Do not lend any money ever.

3.6.11 Payer-back Gene

The responsibility of the "payer_back" gene is to pay the debts to the creditors. Currently there are three "payer_back" strategies.

Alex1payer_back:

For all creditors i of the truck sorted in descending by the interest rate order do:

free_capital = truck's capital - GAS_PRICE * GAS_MOVE * (NUM_AV + NUM_ST);

if borrowed from creditor i too recently, do not pay back.

else if plan state is BUYING pay back no more than the half of the free capital.

Else, pay back no more than the free capital.

Alex2payer_back:

For all creditors i of the truck sorted in descending by the interest rate order do:

free_capital = truck's capital - GAS_PRICE * GAS_MOVE * (NUM_AV + NUM_ST);

if borrowed from creditor i too recently, do not pay back.

else if plan state is BUYING do not pay back anything to anybody.

Else pay back no more than the free capital.

Alex3payer_back:

For all creditors i of the truck sorted in descending by the interest rate order do:

free_capital = truck's capital - GAS_PRICE * GAS_MOVE * (NUM_AV + NUM_ST);

if borrowed from creditor i too recently, do not pay back.
Else, pay back no more than the free capital.

3.7 Strengths of Our Approach

The strength of our approach comes from its two properties - any minimal building block, strategy, comprising our organisms is a complete and meaningful entity, and the overall structure of any organism is not random but rather meaningful. These two properties guarantee that any organism would be working, though with different degree of success, and that the algorithm would not spend time on generating totally random and meaningless creatures. In contrast, other genetic algorithms, that have an assembly instruction or a high level language expression as a minimal building block, and an arbitrarily growing parse three as organism's overall structure, waste lots of their time generating nonsense organisms.

3.8 Weaknesses of Our Approach

3.8.1 Inability to Automatically Generate New Strategies

One of the weaknesses of our approach is that it is unable to automatically generate new strategies. All strategies should be designed and hand coded by a programmer. Usually, in order to find a well performing organism, we need to supply our simulation environment with quite a large number of different strategies (42 strategies that we have right now is far from being enough). Writing a large number of strategies could prove to be a laborious task. In addition, it's difficult to come up with a large number of strategies, which are sufficiently different from each other.

One way around this problem is to extrapolate our system from being a two level system (strategy-organism) into a three level system (substrategy-strategy-organism). Similarly to a two level system with a set of hand written strategies and a set of rules and constrains, which allows to automatically generate meaningful organisms from those strategies, we can have a three level system with a set of handwritten substrategies, a set of strategy rules and constrains, which allows to automatically generate meaningful strategies from the substrategies, and a set of organism rules and constrains, which allows to automatically generate meaningful organisms from the strategies. The advantage of dividing strategies further into substrategies is that a substrategy should be simpler than a strategy, and hence should be simpler to design. If we feel that a three level system is not flexible enough, we can extrapolate it further down into a n-level system.

To exemplify the ideas above, consider how our trucks could be designed in a three level system framework. Let truck's state be represented by the following data type (Figure 7):

```
truckState
{
  point position;          //truck's current location

  money capital;           //truck's capital

  goods commodities;       //commodities in truck's possession

  time simTimeLeft;        //simulation time not yet used by the truck

  dealType nextDeal;       //type of the next deal, could be one of the
                           //following: none, buyItem, sellItem,
                           //buyCrate, or sellCrate

  point nextDealLocation; //location of the dealer where truck intends to
                           //make a deal

  mapOfDealers dealerMap; //that's where the truck records information on the
                           //dealers seen so far. The information includes
                           //dealers' locations, prices, and the type of
                           //commodities the dealers buy or sell
}
```

Figure 7: Data type Representing Truck's State

Let the following four function signatures represent the auxiliary functions that help us to manipulate truck's state (Figure 8):

```
truckState GetState(void)
   //returns truck's state

void SetState(truckState)
   //sets truck's state to the state passed as an argument

point GetDealerLocation(truckState)
   //returns the location of the dealer that we want to do business with
   //(field nextDealLocation in the truckState)

pointList concat(pointList, pointList)
   //concatenates two lists of map locations
   //(this function does not manipulate truck's state)
```

Figure 8: Auxiliary Functions for State Manipulation

Let the following 14 function signatures represent the low level genes, or as we call them sub-genes (Figures 9 10 11 12). The three-level system uses gene-level rules to combine the 14 sub-genes into genes. Note, that each of the sub-genes below can have several alternative handwritten implementations called substrategies.

```
//Sub-genes used by the gene-level rules generating "Move" genes

point GeneratePoint(truckState)
        //Given the state of the truck,
        //returns a location on the map.
        //A straightforward implementations of the
        //sub-gene can simply return truck's current
        //location reading "position" field of the truckState.
        //More involved implementations can
        //take into consideration some other factors
        //and return a location different than the current one.

point GeneratePointToTheNorth(point, truckState)
        //Given a location on the map and the state of the truck,
        //generates a location to the North of the given one.
        //One way to create different implementations for this
        //sub-gene is to use different offsets from the given point.
        //Another way is to generate in one implementation a point different
        //than the given one only on every other call, in another
        //implementation on every third call, and so on, producing trucks
        //that move with different speeds.

point GeneratePointToTheSouth(point, truckState)
        //Given a location on the map and the state of the truck,
        //generates a location to the South of the given one.
        //One way to create different implementations for this
        //sub-gene is to use different offsets from the given point.
        //Another way is to generate in one implementation a point different
        //than the given one only on every other call, in another
        //implementation on every third call, and so on, producing trucks
        //that move with different speeds.
```

Figure 9: Sub-genes

```
point GeneratePointToTheWest(point, truckState)
        //Given a location on the map and the state of the truck,
        //generates a location to the West of the given one.
        //One way to create different implementations for this
        //sub-gene is to use different offsets from the given point.
        //Another way is to generate in one implementation a point different
        //than the given one only on every other call, in another
        //implementation on every third call, and so on, producing trucks
        //that move with different speeds.


point GeneratePointToTheEast(point, truckState)
        //Given a location on the map and the state of the truck,
        //generates a location to the East of the given one.
        //One way to create different implementations for this
        //sub-gene is to use different offsets from the given point.
        //Another way is to generate in one implementation a point different
        //than the given one only on every other call, in another
        //implementation on every third call, and so on, producing trucks
        //that move with different speeds.

truckState GoToPoint(point, truckState)
         //Goes from the current location to the map location passed as an
         //argument. Returns a truck state updated with the new location
         //and with the information about dealers that the truck passed by.
         //A straightforward implementations of the sub-gene simply goes to the
         //specified point. More advanced implementations may choose to
         //go or not to go depending on the truck's state. Also, any
         //implementation should decide on what path to take while moving
         //from the source point to the destination one.

//Sub-genes used by the gene-level rules generating  "FindDeal" genes

pointList GenerateListOfProducers(truckState)
         //Retrieves the locations of producers from the "dealer map"
         //stored in the truck's state. The returned list contains only the
         //producers already seen by the truck. Different implementations
         //of this sub-gene may choose, based on some criterion, to
         //retrieve not all but a part of producers.
```

Figure 10: Sub-genes (cont.)

```
point-List GenerateListOfRetailers(truckState)
          //Retrieves the locations of retailers from the "dealer map"
          //stored in the truck's state. The returned list contains only
          //the retailers already seen by the truck. Different implementations
          //of this sub-gene may choose, based on some criterion, to
          //retrieve not all but a part of retailers.


pointList GenerateListOfConsumers(truckState)
          //Retrieves the locations of consumers from the "dealer map"
          //stored in the truck's state. The returned list contains only
          //the consumers already seen by the truck. Different implementations
          //of this sub-gene may choose, based on some criterion, to
          //retrieve not all but a part of consumers.

truckState AddMoreDealersToDealerMapViaPhoneEnquiry(truckState)
          //Using phone, enquires a number of intersections about their
          //dealers. Returns a truckState updated with the dealer info
          //obtained from the enquires. Different implementations should
          //decide on what and how many intersections to enquire.

truckState UpdateInfoOnDealersOnTheListViaPhoneEnquiry(truckState, pointList)
          //Given a list of dealer locations, enquires those locations by
          //phone, updating the "dealer map" in the truck state. Different
          //implementations of the sub-gene should decide based on the phone call
          //cost factor whether to call up the whole list or only a part of it.
```

Figure 11: Sub-genes (cont.)

```
truckState DecideOnDealType(pointList, truckState)
          //Given a list of dealers and the information on those
          //dealers stored in the truck state, decides on which
          //type of deal to perform. Valid deal types are "none",
          //"buyItem", "sellItem", "buyCrate", and "sellCrate".
          //The decision is returned in the "nextDeal" field
          //of the truck state.


truckState FindOptimalDeal(pointList, truckState)
          //Given a list of dealers and the information on those
          //dealers stored in the truck state, decides on the place
          //and the type of transaction. The decision is returned in the
          //"nextDeal" and "nextDealLocation" fields of the truck state.

//Sub-genes used by the gene-level rules generating  "BuyOrSell" genes.
//Note, that "BuyOrSell" generating rules also use "GoToPoint" sub-gene.

truckState PerformTransaction(truckState)
          //Performs at the current location the transaction of the type
          //indicated in the nextDeal field of the truck state. Returns a new
          //truck state with adjusted capital and commodities fields.
          //Different implementations of the sub-gene should decide on the
          //amounts of commodities that they whish to buy or to sell
```

Figure 12: Sub-genes (cont.)

The Figures 13 and 14 below give the gene-level rules that generate gene-level strategies from the sub-genes. Though here we give only six rules, it is easy to come up with more.

```
//Rules to generate Move gene strategies

Move1:
  SetState(GoToPoint(GeneratePointToTheEast(
           GeneratePoint(GetState()),GetState()),GetState()));
  //Moves truck eastwards. This rule uses three different sub-genes.
  //If we implement four substrategies for each of the three sub-genes,
  //than the rule can generate 4^3 = 64 different move strategies

Move2:
  SetState(GoToPoint(GeneratePointToTheEast(
           GeneratePointToTheSouth(GeneratePointToTheNorth(
           GeneratePoint(GetState()),GetState()),GetState()),
           GetState()),GetState()));
  //Moves into direction obtained by a summation of three vectors directed
  //northward, southward and eastward . This rule uses four different sub-genes.
  //If we implement four substrategies for each of the four sub-genes,
  //than the rule can generate 4^4 = 256 different move strategies


//Rules to generate FindDeal gene strategies

FindDeal1:
  SetState(FindOptimalDeal(concat(GenerateListOfProducers(GetState()),
           GenerateListOfRetailers(GetState())),
           DecideOnDealType(concat(GenerateListOfProducers(GetState()),
           GenerateListOfRetailers(GetState())),GetState())));
  //Finds a deal either with a producer or with a retailer.
  //This rule uses four different sub-genes.
  //If we implement four substrategies for each of the four sub-genes,
  //than the rule can generate 4^4 = 256 different FindDeal strategies
```

Figure 13: Gene-Level Rules

```
FindDeal2:
  SetState(UpdateInfoOnDealersOnTheListViaPhoneEnquiry(GetState(),
           GenerateListOfProducers(GetState())));

  SetState(UpdateInfoOnDealersOnTheListViaPhoneEnquiry(GetState(),
           GenerateListOfRetailers(GetState())));

  SetState(FindOptimalDeal(concat(GenerateListOfProducers(GetState()),
           GenerateListOfRetailers(GetState())),
           DecideOnDealType(concat(GenerateListOfProducers(GetState()),
           GenerateListOfRetailers(GetState())),GetState())));
  //Makes phone calls to update information on producers and retailers and
  //then finds a deal either with a producer or with a retailer.
  //This rule uses five different sub-genes.
  //If we implement four substrategies for each of the five sub-genes,
  //than the rule can generate 4^5 = 1024 different FindDeal strategies

FindDeal3:
  SetState(AddMoreDealersToDealerMapViaPhoneEnquiry(GetState()));

  SetState(FindOptimalDeal(concat(GenerateListOfRetailers(GetState()),
           GenerateListOfConsumers(GetState())),GetState()));
  //Finds additional dealers via a phone enquiry and then finds a deal
  //either with a retailer or with a consumer.This rule uses
  //four different sub-genes. If we implement four substrategies for each
  //of the four sub-genes, than the rule can generate 4^4 = 256 different
  //FindDeal strategies.

//Rules to generate FindDeal gene strategies

BuyOrSell:
  SetState(GoToPoint(GetDealerLocation(GetState()),GetState()));

  SetState(PerformTransaction(GetState()));
  //Goes to the dealer's location indicated in the truck's state and then
  //performs the transaction indicated in the state. The transaction could
  //be one of the following: none, buyItem, sellItem, buyCrate, or sellCrate.
  //This rule uses two different sub-genes.If we implement four substrategies
  //for each of the two sub-genes, than the rule can generate
  //4^2 = 16 different BuyOrSell strategies.
```

Figure 14: Gene-Level Rules (cont.)

As can be seen from Figures 9 10 11 12 and Figures 13 and 14, with 14 sub-genes, where each of the sub-genes has 4 different implementations, and 6 gene-level rules, the three-level system can generate $64 + 256 + 256 + 1024 + 256 + 16 = 1872$ different gene-level strategies. With a three-level system, in order to get those 1872 gene strategies a programmer is required to write just $14 * 4 = 56$ simple routines or substrategies. The same or grater amount of programming effort in a two-level system would produce only 56 gene-level strategies.

To generate organisms from the gene-level strategies generated by the gene-level rules a three-level system uses the organism-level rules. Figure 15 gives us the organism-level rules that generate our trucks.

```
Truck1:
  Move;
  FindDeal;
  BuyOrSell;
```

Figure 15: Organism-Level Rules

Now, as we've seen in Figures 7 through 15 a way of generating a truck in a three-level system, we are still left to show a way of applying a genetic algorithm to this kind of system, and more importantly a way of encoding an organism in a chromosome as well as a way of doing a cross-over. Figure 16 shows how to do this.

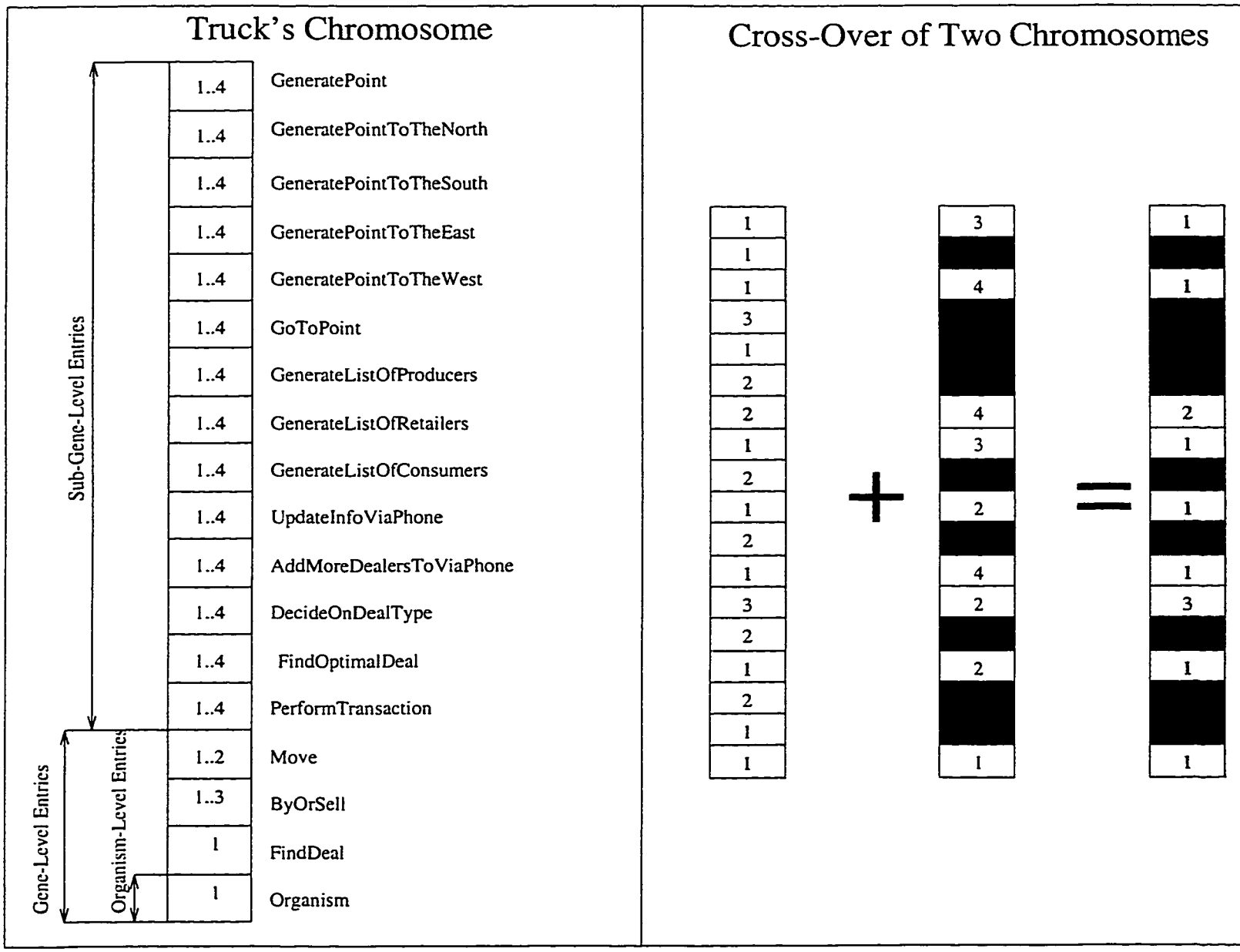


Figure 16: Chromosome and Cross-Over in a Three-Level System.

As can be seen from the Figure 16 , a chromosome string consists of three parts - a sub-gene partition, gene partition, and organism partition. Since we have 14 sub-genes, the sub-gene partition has 14 entries. In a system where, for example, each of the sub-genes has

four different implementations or substrategies, a sub-gene partition entry can take a value from one to four. In our three-level trucking system we have three genes, Move, FindDeal, and BuyOrSell, which translate into three entries in the gene partition. Since we have two Move rules, three FindDeal rules, and one BuyOrSell rule, the Move entry can take a value from one to two, the FindDeal entry from one to three, and the value of BuyOrSell is always one. The organism partition has only one entry. Since we have only one organism-level rule, the value of that entry is always one. With this type of chromosome encoding cross-over is done by randomly selecting nine entries from the first parent chromosome and copying them into the child chromosome. The remaining nine entries are copied from the second parent chromosome.

3.8.2 Small Population Size.

Another weakness of our algorithm is that we try to operate in a moderate size search space using a small population (usually 50 or 100 trucks). The common opinion in GA literature suggests that for a good evolution in a large search space one needs a population of hundreds or even thousands. One of the directions for the future development in "Object-Oriented Trucking" would be simulation runs with populations of several thousands.

Chapter 4

Conducted Experiments

In the previous chapter we defined our framework, "Object-Oriented Trucking", and pointed out its strong and weak sides. In this chapter we will describe the experiments conducted under the framework.

4.1 Theoretical Maximum

First, let's find out if our search space has a maximum. Namely, we are interested in the maximum of the objective function $\sum_{i=1}^{N} f(chr_i, \ldots)$, where $f(chr, s, s_{env}, t, chr_1, \ldots, chr_N, s_1, \ldots, s_N)$ are the earnings of an organism o_i at the end of a simulation run. That is, we want to know what the best possible performance of a population is. By knowing the performance maximum, we would know what kind of performance we should aim in our simulation runs. In our analysis, inspired by [Gro97], we will make a use of the initial state and principal parameters described on page 34. The environment initial state, given by the seven variables is the following:

$k = 20\frac{items}{crate}$ - number of items in a crate.

$n_c = 20$ - number of consumers

$n_p = 5$ - number of producers

$r_c = 50\frac{items}{min}$ - maximum possible rate at which a consumer purchases items. This rate is determined by the rate at which producers supply the system. The rate is computed by the following formula: $r_c = \frac{r_p n_p}{n_c} = \frac{2005}{20} = 50\frac{items}{min}$, where $r_p = 200\frac{items}{min}$ is the rate at which a producer produces items. The rate r_c could be less than $50\frac{items}{min}$ if the trucks are unable to deliver at such a high rate.

$w = 1kg$ - weight of one item

$a = \frac{\$0.8}{item}$ - amount per item paid by a producer-retailer (PR) truck to a producer

$b = \frac{\$1.8}{item}$ - amount per item paid by a consumer to a retailer-consumer (RC) truck

The next seven variables are the principal parameters.

$n_{pr} = 25$ - number of PR trucks. The total number of trucks in our simulation is 50. We assume that a half of those trucks specializes in the producer-retailer transactions.

$n_{rc} = 25$ - number of RC trucks. The total number of trucks is 50. We assume that a half of those trucks specializes in the retailer-consumer transactions.

$n_r = 25$ - number of retailers.

$t = 60min$ - average time to journey from a buyer to a seller and back to another buyer. The length of the country is ten steps, thus, on average it takes five steps to go from a seller to a buyer, and another five step to go from a seller to a buyer, which gives us a total of ten steps per a buy-sell journey. Each step takes six minutes, thus, the average journey time is $10 \times 6 = 60$ minutes.

$p_r = 0.71$ - retailer's profit, is determined by the following expression: $p_r = \frac{p_{rmin}+p_{rmax}}{2}$, where $p_{rmin} = \frac{d_{min}-c_{max}}{c_{max}}$ and $p_{rmax} = \frac{d_{max}-c_{min}}{c_{min}}$. At the beginning of the simulation, retailer's buy price c and retailer's sell price d are set to their max $c = c_{max} = \$1.2$ and min $d = d_{min} = \$1.4$ values. As a high-performance simulation run goes, the trucks buy and sell with a high rate, lowering retailers' buy prices and raising retailers' sell prices. Since we are trying to estimate the upper bound on trucks' performance, we have to assume a high-performance simulation run with the retailers lowering their buy prices and raising their sell prices. In the extreme case, the retailer's buy price can drop to the producer's sell price ($c = c_{min} = a = \frac{\$0.8}{item}$), and the retailer's sell price can raise to the consumer buy price ($d = d_{max} = b = \frac{\$1.8}{item}$). Thus, $p_{rmin} = \frac{d_{min}-c_{max}}{c_{max}} = \frac{1.4-1.2}{1.2} = 0.17$, $p_{rmax} = \frac{d_{max}-c_{min}}{c_{min}} = \frac{1.8-0.8}{0.8} = 1.25$, and $p_r = \frac{p_{rmin}+p_{rmax}}{2} = \frac{0.17+1.25}{2} = 0.71$.

$p_{rc} = 0.14$ - RC truck's profit. The profit p_{rc} is determined by the following expression: $p_{rc} = \frac{p_{rcmax}+p_{rcmin}}{2}$. $p_{rcmin} = \frac{b-d_{min}}{d_{min}} = \frac{1.8-1.4}{1.4} = 0.28$. $p_{rcmax} = \frac{b-d_{max}}{d_{max}} = \frac{1.8-1.8}{1.8} = 0$. Thus, $p_{rc} = \frac{p_{rcmax}+p_{rcmin}}{2} = \frac{0.28+0}{2} = 0.14$.

$p_{pr} = 0.15$ - PR truck's profit. The amount payed by a consumer to an RC truck is derived from the following expression: $b = a(p_{pr} + 1)(p_r + 1)(p_{rc} + 1)$. How we obtained this expression will become clear when we will be calculating the earnings of retailers, PR trucks,and RC trucks(see page 62). Substituting in the expression above p_r, p_{rc}, a, and b by the correspondent values we would obtain: $1.8 = 0.8(p_{pr} + 1)(0.71 + 1)(0.14 + 1) \Rightarrow p_{pr} = \frac{1.8}{(0.71+1)(0.14+1)0.8} - 1 = 0.15$

As we saw above, the maximum possible rate at which a consumer purchases items is $r_c = 50\frac{items}{min}$. However, we did not take into account that this rate can not be higher

than the rate at which the RC trucks deliver items to the consumers. To estimate r_c more realistically, let's find the truck's maximum delivery rate. The time that a truck needs to buy from a retailer, to sell to a consumer, and to return to another retailer is $60min$. The total simulation time is $5000min$. Thus, a truck has enough time to make no more than $\frac{5000}{60} = 83$ deliveries. The maximum number of items that a truck can deliver in one trip equals to truck's maximum capacity, which is 1800 items. Thus, a truck can deliver $83 \times 1800 = 149400$ items over the 5000 min time interval. All 25 RC trucks during the simulation can deliver $149400 \times 25 = 3735000$ items. Since we have 20 consumers, each consumer over the simulation time receives $\frac{3735000}{20} = 186750$ items, or $\frac{186750}{5000} = 37\frac{items}{min}$. Thus, the maximum consumption rate can not be higher than $37\frac{items}{min}$. Therefore, we obtain:

$$r_c = 37\frac{items}{min}.$$

To maintain this consumption rate, the rate at which trucks deliver items form producers to retailers, $r_{pr}n_{pr}$, should be the same as the rate at which the retailers buy $r_r n_r$, and the rate at which trucks deliver from retailers to consumers, $r_{rc}n_{rc}$ should be the same as the rate at which consumers buy $r_c n_c$. That is, $r_{pr}n_{pr} = r_r n_r = r_{rc}n_{rc} = r_c n_c \Rightarrow r_r = \frac{r_c n_c}{n_r} = \frac{37\times 20}{25} = 29.6\frac{items}{min}$ and $r_{pr} = \frac{r_c n_c}{n_{pr}} = \frac{37\times 20}{25} = 29.6\frac{items}{min}$ and $r_{rc} = \frac{r_c n_c}{n_{rc}} = \frac{37\times 20}{25} = 29.6\frac{items}{min}$, where r_r is the rate at which an individual retailer buys, r_{pr} is the rate at which an individual PR truck delivers, and r_{rc} is the rate at which an individual RC truck delivers. Thus,

$$r_c = 37\frac{items}{min},$$

$$r_r = 29.6\frac{items}{min},$$

$$r_{pr} = 29.6\frac{items}{min},$$

$$r_{rc} = 29.6\frac{items}{min}.$$

The throughput r, which is the number of items that pass through the system every minute is $r = r_c n_c = 37 \times 20 = 740\frac{items}{min}$.

4.1.1 Earnings Calculations

Now we will calculate the earnings of retailers, PR trucks, and RC trucks. The maximum throughput of the system r, which shows haw many items are passed by trucks through the system every minute, is the dominant factor in our calculations. Since we are using the maxim possible throughput $r = 740$, what we are going to calculate are the maximum possible earnings. Here are the calculations:

$ap_{pr} = 0.8 \times 0.16 = \frac{\$0.13}{item}$ - PR truck's earnings per item

$ap_{pr}\frac{r}{n_{pr}} = 0.8 \times 0.13 \times 740 \div 25 = \frac{\$3.78}{min}$ - PR truck's earnings per minute

$ap_{pr}\frac{r}{n_{pr}}t_{simulation} = 3.78 \times 5000 = \$18944$ - PR truck's total earnings

$ap_{pr}\frac{r}{n_{pr}}t_{simulation}{}^{n_{pr}} = 18944 \times 25 = \$473600$ - total earnings of all PR trucks

$a(p_{pr}+1) = 0.8(0.13+1) = \frac{\$0.9}{item}$ - amount per item paid by a retailer to a PR truck

$a(p_{pr}+1)p_r = 0.9 \times 0.71 = \frac{\$0.64}{item}$ - retailer's earnings per item

$a(p_{pr}+1)p_r\frac{r}{n_r} = 0.64 \times 740 \div 25 = \frac{\$18.94}{min}$ - retailer's earnings per minute

$a(p_{pr}+1)p_r\frac{r}{n_r}t_{simulation} = 18.94 \times 5000 = \$94720$ - total retailer's earnings

$a(p_{pr}+1)p_r\frac{r}{n_r}t_{simulation}{}^{n_r} = 95720 \times 25 = \$2368000$ The earnings of all retailers

$a(p_{pr}+1)(p_r+1) = 0.8(0.13+1)(0.71+1) = \frac{\$1.57}{item}$ - amount per item paid by a RC truck to a retailer

$a(p_{pr}+1)(p_r+1)p_{rc} = 1.57 \times 0.14 = \frac{\$0.23}{item}$ - amount per item earned by a RC truck

$a(p_{pr}+1)(p_r+1)p_{rc}\frac{r}{n_{rc}} = 0.23 \times 740 \div 25 = \frac{\$6.8}{min}$ - RC truck's earnings per minute

$a(p_{pr}+1)(p_r+1)p_{rc}\frac{r}{n_{rc}}t_{simulation} = 6.8 \times 5000 = \$34040$ - RC truck's total earnings

$a(p_{pr}+1)(p_r+1)p_{rc}\frac{r}{n_{rc}}t_{simulation}{}^{n_{rc}} = 34040 \times 25 = \$851000$ - total earnings of all RC trucks

$a(p_{pr}+1)(p_r+1)(p_{rc}+1) = b = \frac{\$1.8}{item}$ - amount per item paid by a consumer to a RC truck

Now, let's compute the amount that the trucks spend on gas.It takes six minutes to make a move. The simulation runs for 5000 minutes, thus a truck has enough time to make $\frac{5000}{6} = 833$ moves. Each move takes 1 liter of gas and costs on average \$1. Thus, to make 833 moves, a truck spends \$833on gas. All 50 trucks spend $833 \times 50 = \$41650$. The total earnings of PR and RC trucks are $473600 + 851000 = 1324600$ and the maximum amount that a generation of trucks can end-up with is $1324600 - 41650 = \$1282950$.The amount of \$1282950 is a theoretical maximum under our initial conditions and the assumptions that we've made. However, in the real simulation runs we achieve much lower numbers because of the system's throughput r that are much lower than $740\frac{item}{min}$.

4.2 Non-Cooperating Trucks Experiment

The first experiment that we run was the Producer-Retailer-Consumer-Truck experiment with non-cooperating trucks.

The trucks cooperate by borrowing from and lending money to each other. The process of borrowing and lending is controlled by "borrow", "lend", and "payerback" genes. Disabling calls to these three genes would disable cooperation. Thus, the trucks in our "non-cooperating trucks" experiment had their "borrow", "lend", and "payerback" genes disabled. This "non-cooperating trucks" experiment was run for 32 generation with the

following search parameters:

- Percentage to Retain = 40%. This parameter determines how many trucks would live into the next generation. In our case, the top-performing 40% of trucks of the current generation are copied into the next generation.

- Percentage of Offsprings = 40%. This parameter determines how many trucks of the next generation are the offsprings of the current generation. In our case, 40% of trucks of the next generation are the offsprings of the current generation. When producing the offsprings, we use the following scheme - each pair of parents contributes the number of offsprings proportional to parents' fitness.

- Percentage of Randomly Generate Trucks = 20%. This parameter determines the number of trucks with randomly generated chromosomes. In our case, 20% of trucks are randomly generated. Random truck generation is an equivalent of mutation in the traditional genetic algorithms. If our search converges on some small local peaks, randomly generated trucks can take the search into another part of the search space with possibly higher peaks.

- Adaptive search parameters = enabled. When the flag "Adaptive search parameters" is enabled, the three search parameters, "percentage to retain", "Percentage of Offsprings", and "percentage of randomly generate trucks", are allowed to change from their initial values of 40%, 40%, and 20% in the following way: if the performance of the current generation is better that the performance of the previous generation, decrease the percentage of randomly generated trucks in the next generation by 1% and increase the percentage of offsprings by 1%. Conversely, if the performance of the current generation is worse than the performance of the previous generation, decrease the percentage of offsprings by one and increase the percentage of random trucks by one. The motivation for the "adaptive search parameters" is the following: if the performance of a generation gets improved, we probably converge on a large peak. We help the search to concentrate on that peak by reducing the number of random jumps into different regions of the search space. If, on the other hand, the performance of a generation does not get improved, there is probably no large peaks around. Therefore, explore other regions of the search space.

The choice of values 40%, 40%, 20%, and "adaptive parameters enabled" for the search parameters described above seems to produce the best results. Whenever we tried to reduce the percentage of random trucks below 20% or to increase the "percentage to retain" above 40%, the search was converging on smaller peaks (about $5000 of the total earnings of all

trucks in a generation), and missing larger peaks (about $40000)), which we were able to find when the search parameters were set to 40%, 40%, 20%, and "adaptive parameters enabled". Whenever we tried to increase the percentage of random trucks above 20%, the search was resembling a random walk that jumped from one region of the search space to another, not converging on a single large peak. Figure 17 shows the performance of 32 generations of non-cooperating trucks, with search parameters of 40%, 40%, 20%, and "adaptive parameters enabled". The x axis in the Figure represents generations. The y axis represents the total earnings of all trucks in a single generation. Figure 18 gives the performance of the best trucks in our 32-generations. The x axis in the Figure represents generations. The y axis represents the earnings of the best-performing truck in a single generation.

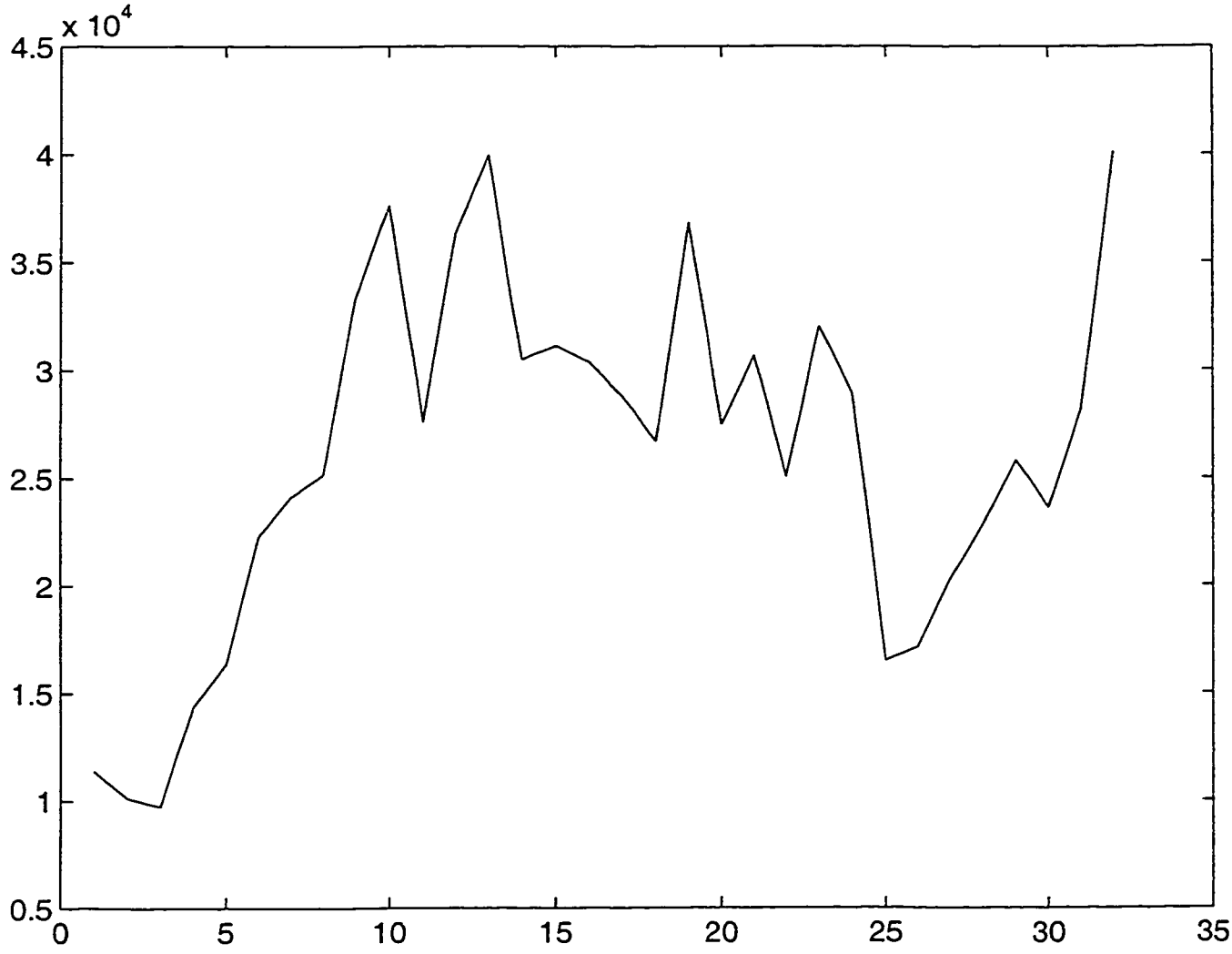


Figure 17: Population Performance (32 generations).

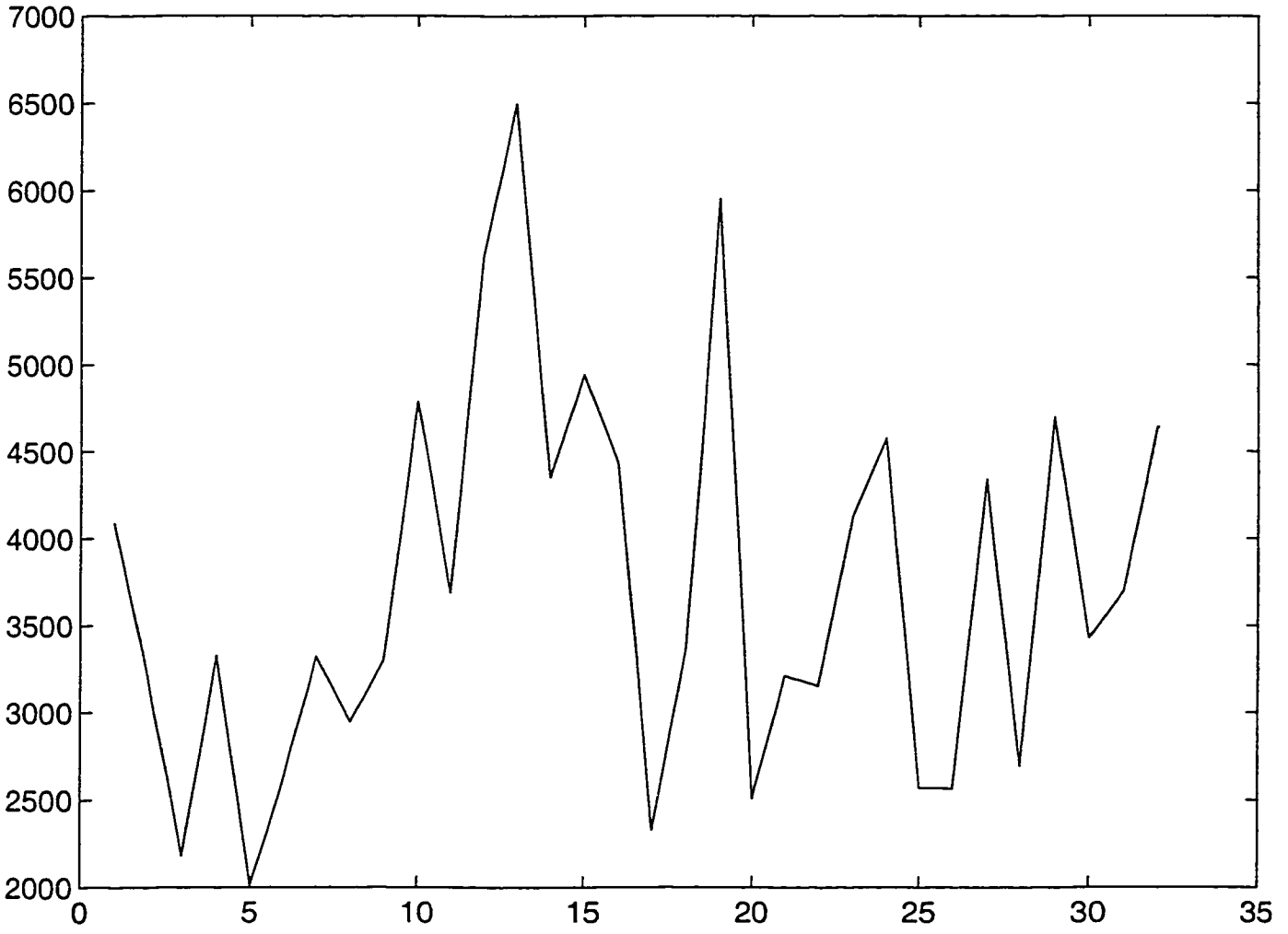


Figure 18: Best Trucks (32 generations).

As can be seen from Figure 17, the best generation of trucks is generation 32, which earned $40133.12. The amount of goods in this run that the PR trucks delivered to retailers over the simulation time is $188400 items$. Thus, the rate of PR truck delivery is $188400 \div 5000 = 37.68 \frac{items}{min}$. The amount of goods that the RC trucks delivered to retailers over $5000 min$ is $115800 items$. Thus, the rate of RC trucks' delivery is $115800 \div 5000 = 23.16 \frac{items}{min}$, and the system throughput $r = minimum(37.68, 23.16) = 23.16 \frac{items}{min}$. The average producer price in this run is $\frac{\$0.8}{item}$, the average retailer buy price is $\frac{\$0.93}{item}$, the average retailer sell price is $\frac{\$1.62}{item}$, and the average consumer price is $\frac{\$1.8}{item}$. The prices above give us the following profit rates: $p_{pr} = 0.16, p_r = 0.43, p_{rc} = 0.1$. These experimental profit rates are comparable with the profit rates $p_{pr} = 0.15, p_r = 0.71, p_{rc} = 0.14$ that we used in our derivations of a theoretical performance maximum. However, the experimental system throughput $r_e = 23.16 \frac{items}{min}$ is much lower than the theoretical maximum $r_t = 740 \frac{items}{min}$. The main cause of the low throughput is the unbalance between the PR and RC trucks. Since at the beginning the profit rate p_{pr} is higher than the profit rate $p_r c$, there are more trucks that try to perform a Producer-Retailer transactions than the trucks that try to perform Retailer-Consumer ones. Retailers keep buying and almost don't sell anything. Eventually, the retailers get filled to the capacity and can not buy anymore. However, most of the truck "deal" strategies don't check the retailers capacity, but rather rely on the profit rate alone. The trucks run around, loaded to the capacity with unsold crates, unsuccessfully trying to sell them to over-flown retailers. This situation wastes the simulation time and lowers the throughput. Since there are more PR transactions that RC transactions, eventually the p_{pr} profit rate becomes lower than the $p_r c$ profit rate. When this happens, all trucks try to buy items from retailers. The huge number of requests to buy immediately raises the retailer sell prices and lowers the profit rate p_{rc}. Even though the retailers received a huge number of requests to sell, not much was sold because the trucks were filled with the unsold crates and did not have much capacity to buy. Thus, we again arrive to the situation when the profit rate p_{pr} is better that the profit rate p_{rc}, and the retailers are still over-flown. This dead-lock like situation repeats till the end of the simulation, causing a low, comparing to the theoretical maximum, throughput.

One way to remedy this situation, would be to add more truck "deal" strategies that verify not only the profit rates when looking for deals, but also the retailers' capacities. Another way to remedy this situation is to rewrite the retailers' code, so that a retailer changes its price based on the amount of goods that it sold rather than based on the number of buy or sell requests received.

4.3 Cooperating Trucks Experiment

To see if the performance of the trucks can be improved if we allow inter-truck cooperation, we added to our trucks an ability to borrow money from, lend money to, and pay money back to each other. We hoped, that by doing this we would prevent the trucks from running under-loaded, increasing the system's throughput. A truck becomes under-loaded if it does not have enough money to fill itself to the capacity whenever a buy transaction is performed. However, a buying truck can afford to get an additional load of goods if it borrows some money from the other trucks that do not plan to buy at the moment. Later, the money can be payed back with some interest.

Indeed, our expectations of better performance were confirmed. We were able to get our best-performing population of trucks just after 20 generations, which is 12 generations faster, comparing to the non-cooperating trucks experiment. In addition, our best population of cooperating trucks managed to earn \$50189.70, whereas the best non-cooperating trucks earned only \$40133.12. The system's throughput went from $23.16\frac{items}{min}$ of non-cooperating trucks experiment to $26\frac{items}{min}$. Though the rate $26\frac{items}{min}$ is better than the rate $23.16\frac{items}{min}$, it's still much lower than the theoretical maximum. The cooperation among trucks helps to avoid under-loaded trucks, but it does not held to avoid the described above problem of unbalance between the PR and RC trucks.

Figure 19 gives the performance of 20 generations of cooperating trucks. The x axis represents generations, the y axis gives the total amount earned by a generation of trucks. Figure 20 gives the performance of the top-performing trucks in those 20 generations. The x axis represents generations, the y axis gives the total amount earned by the best-performing truck in a given generation.

The results depicted in Figures 19 and 20 were obtained from the simulation run with the following search parameters: percentage to retain = 40%, percentage of offsprings = 40%, percentage of randomly generate trucks = 20%, and the adaptive search parameters = enabled. As with the non-cooperating trucks, this setting of the search parameters seems to produce the best results for the cooperating trucks as well.

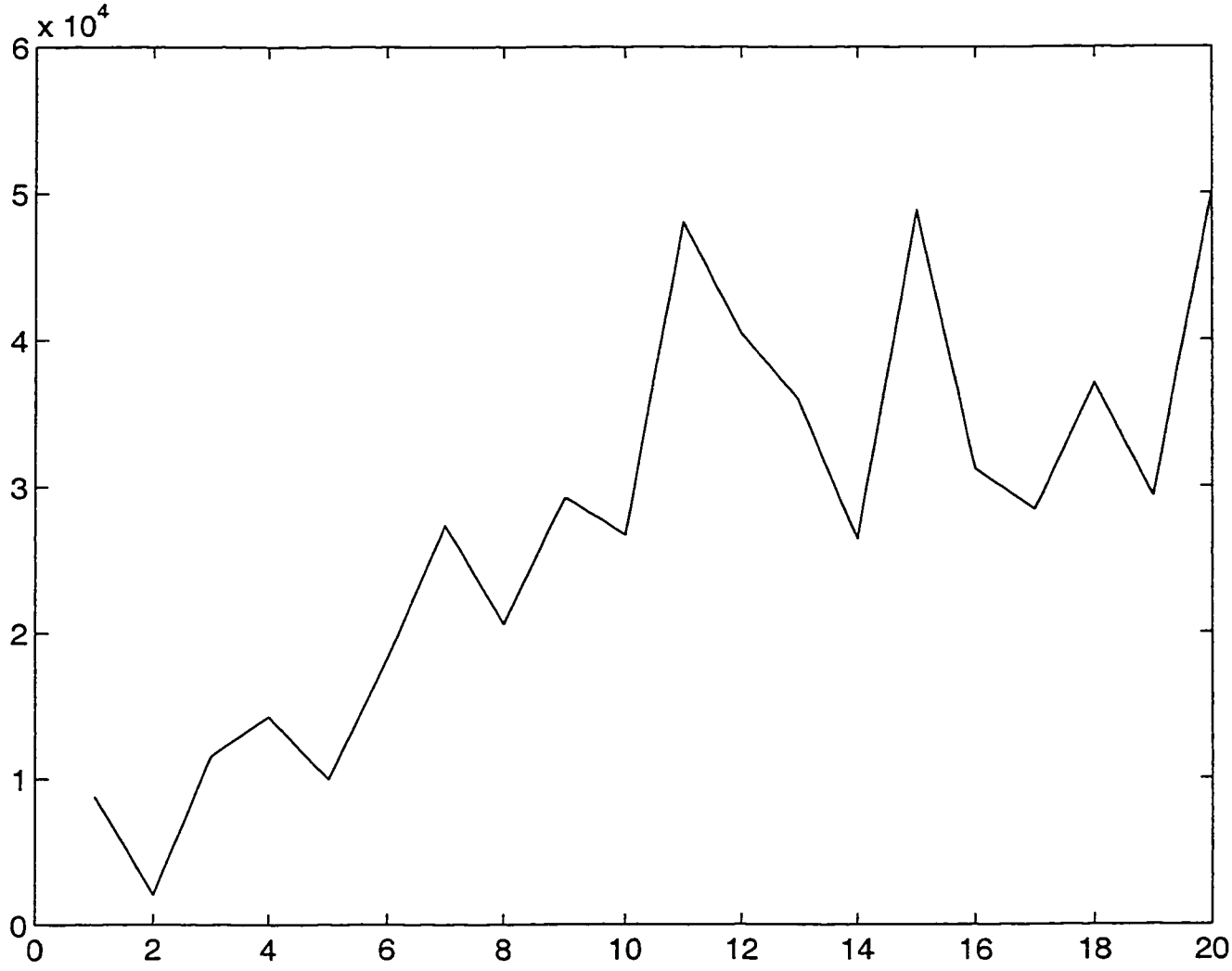


Figure 19: Population Performance (20 generations).

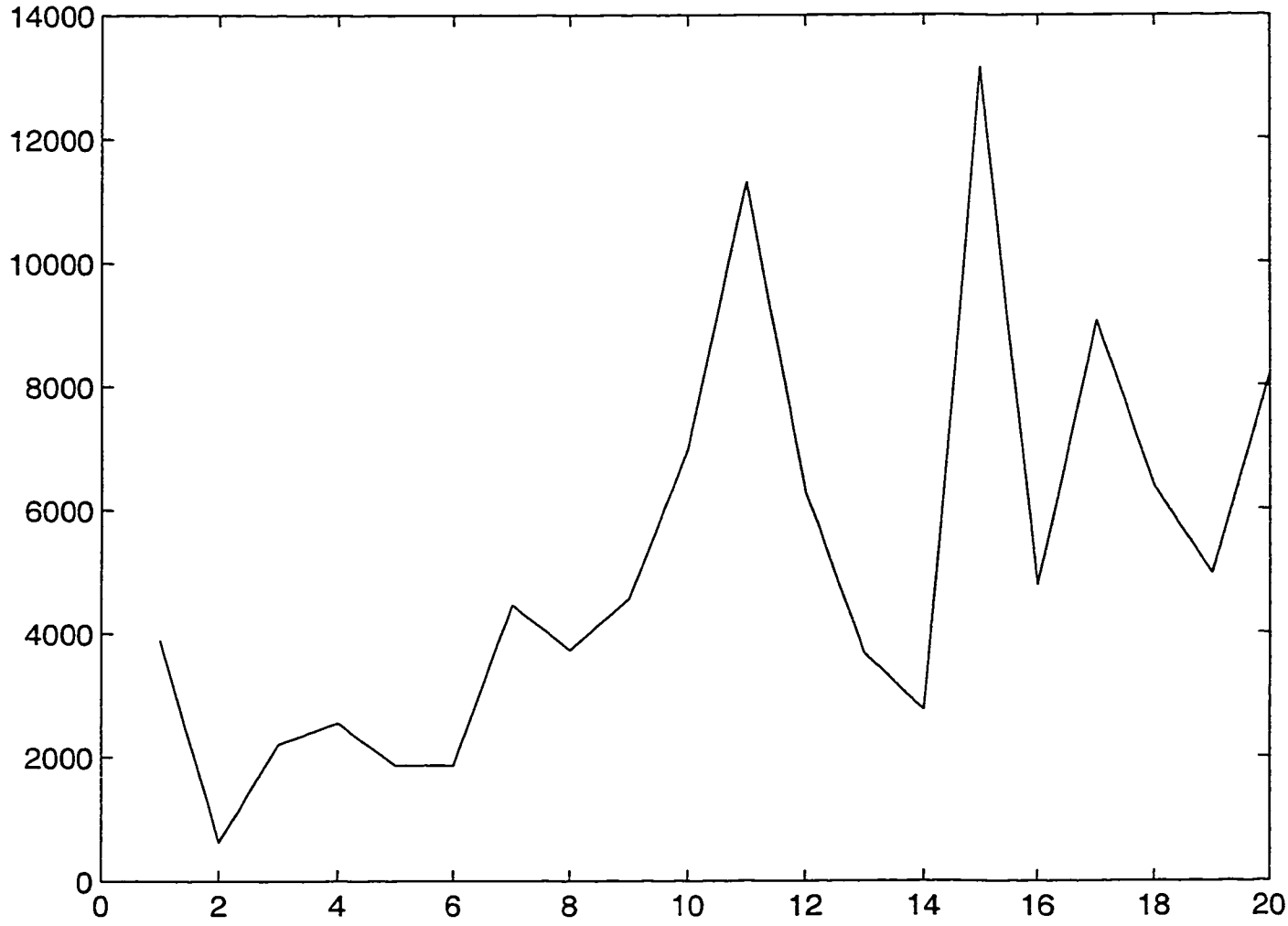


Figure 20: Best Trucks (20 generations).

Chapter 5

Conclusions

5.1 Summary

Genetic algorithms are highly efficient and robust search methods that guide their searches by exploiting the similarity information among the population of strings encoded over a finite alphabet. Since the genetic algorithms do not work directly with the search space parameters but rather work with the parameters encoded in a string over a finite alphabet, the search space encoding schemes are the central topic in the research on genetic algorithms. Presently, there are two major types of encoding - a machine code encoding and a parse tree encoding.

With the machine code encoding, mostly used in artificial life systems, each organism is represented by a fragment of a machine code (see [Ray90], [Ray95]). The machine code used should possess the following property: random flip of a bit in a machine instruction should produce, with a high probability, a new meaningful instruction. A machine code organism replicates by copying itself into the RAM of the computer where it "lives". While copying, random errors are possible. These errors make the copy slightly different from its parent. In addition, with low probability, some bits in the copy may be flipped (mutated). Random errors and mutation populate the environment (computer memory) with a variety of organisms. Those types of organisms that can replicate the most efficiently prevail the environment. Since the errors during copying and mutations occur at random, at any given moment a large fraction of the computer memory is populated by the organisms consisting of totally random combinations of instructions and unable to perform the task at hand. Those nonsense organisms slow down the genetic search.

With the parse tree encoding, each organism is a fragment of code represented by a parse tree written using Lisp's S-expressions (see [Koz92],[Koz90]). The parse tree of two

individuals can be combined by the genetic algorithm together to produce a new individual. There are very few constraints on the way how the parse tree can be combined, so the individuals generated by the combination can take virtually any shapes and sizes. The majority of the generated organisms have nonsense structures, and therefore unable to perform the task at hand. Those few that are able to perform the task are still not optimal - only a small portion of the parse tree code does a useful work, the rest is a "dead" code resulted from the random recombination.

To overcome the problem of "nonsense" organism generation, in our approach, "Object-Oriented Trucking" we introduced the following encoding scheme: Each organism is encoded by an eleven entries long chromosome. Each entry in the chromosome represents a base C++ class, which is responsible for one particular type of organism's behavior. For example, entry seven represents the Move base class responsible for moving behavior, and entry nine represents the Deal base class responsible for dealing behavior. The value inside the chromosome entry selects one of the several programmer-written subclasses or strategies derived from the base class. To generate a new organism, the algorithm takes two chromosomes from two parents, selects at random six entries in the first chromosome and five entries in the second. Than, the selected six and five values are copied into the child's chromosome. Since all the organisms with this type of encoding have the same eleven-gene structure and all strategies are hand-written by a programmer in a meaningful way, no nonsense organisms with random structures are possible.

5.2 The Effects of my Contributions

- Three new genes (borrow, lend, and payer_back) — to allow us to observe the effects of the truck collaboration.
- Eighteen new strategies — the simulation is not able to find interesting results when too few strategies are available. Adding a number of new strategies enabled the simulation to find generations of trucks with high performance levels.
- Statistics module — provided in order to better understand what is going on in the simulation.
- Adaptive Search Parameters Algorithm — makes the compromise between the smooth and noisy search. That is, the randomness (noise) is added when the performance goes down, and the randomness is reduced when the performance goes up.

5.3 Future Work

Though proofed against generation of meaningless organisms, our system has a number of problems caused by not having a sufficient number of strategies that are significantly different from each other. For example, the problem of unbalance between the PR and RC trucks, described on the page 68, caused by the fact that we don't have enough deal strategies that take into account dealers' capacities. One of the items on our list of future enhancements would be to remedy this problem and to write mode deal strategies that take into account dealers' capacities.

Writing a few extra deal strategies may be easy, however writing a sufficient number of strategies for a more or less complex problem could be a laborious task. Instead of writing strategies, it would be nice to have a system that can automatically generate new strategies from a set of simple hand-written primitives. It also desirable in such a system to avoid a generation of meaningless organisms. The tree-level system, described on the pages 47 - 59 is an example of such a system. Implementing the tree-level system and making experiments with it would be another item on our list of future enhancements.

Bibliography

[Gol89] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*, chapter 1. Addison-Wesley Publishing Company, Inc, 1989.

[Gro97] P. Grogono. The truckin' project - the producer/retailer/consumer model. Technical report, Concordia University, Department of Computer Science, 1997.

[Hol68] J.H. Holland. Hierarchical description of universal spaces and adaptive systems. Technical report, Ann Arbor: University of Michigan, Department of Computer and Communication Sciences, 1968.

[Hol75] J. H. Holland. *Adaptation in Natural and Artificial Systems.* Ann Arbor: The University of Michigan Press, 1975.

[Koz90] J. R. Koza. A genetic approach to economic modeling. Technical report, Sixth World Congress of the Economic Society. Barcelona, Spain, 1990.

[Koz92] J. R. Koza. Genetic evolution and co-evolution of game strategies. Technical report, Stanford University, Department of Computer Science, 1992.

[Ray90] T. S. Ray. An approach to the synthesis of life. Technical report, Santa Fe Institute Studies in the Science of Complexity, 1990.

[Ray95] T. S. Ray. Artificial life. Technical report, ATR Human Information Processing Laboratories., 1995.

[Smi80] S. F. Smith. *Learning System Based on Genetic Adaptive Algorithms.* PhD thesis, University of Pittsburg, 1980.

Appendix A

Base Gene Classes

```
//Base class representing "Init" gene
class Init {
public:
  //Gene's interface function
  virtual void init ();
protected:
  Truck *truck;
};

//Derived strategy subclass of the "Init" gene
class Peteinit : public Init {
public:
  //auxiliary function
  Peteinit (Truck *tr) { truck = tr; }
private:
  //Implements gene's interface function
  void init ();
};

//Derived strategy subclass of the "Init" gene
class Debinit : public Init {
public:
  //auxiliary function
  Debinit (Truck *tr) { truck = tr; }
private:
  //Implements gene's interface function
  void init ();
};


//Base class representing "Gas" gene
class Gas {
public:
  //Gene's interface function
  virtual void check_gas (int& done_it);
protected:
  Truck *truck;

};
```

Figure 21: Base classes representing genes along with the derived from them subclasses representing strategies.

```
//Derived strategy subclass of the "Gas" gene
class Petegas : public Gas {
public:
  //auxiliary function
  Petegas (Truck *tr) { truck = tr; }
private:
  //Implements gene's interface function
  // Method controlling gas seeking and buying behavior.
  // Set done_it = 1 if in midst of moving to gas station, otherwise 0.
  void check_gas (int& done_it);
};

//Derived strategy subclass of the "Gas" gene
class Debgas : public Gas {
public:
  //auxiliary function
  Debgas (Truck *tr) {truck = tr; origin_valid = 0; gas_in_progress = 0;}
private:
  //Implements gene's interface function
  // Method controlling gas seeking and buying behavior.
  // Set done_it = 1 if in midst of moving to gas station, otherwise 0.
  void check_gas (int& done_it);
  Place origin; int origin_valid; int gas_in_progress;
};

//Derived strategy subclass of the "Gas" gene
class Jeffgas : public Gas {
public:
  //auxiliary function
  Jeffgas (Truck *tr) { truck = tr; }
private:
  //Implements gene's interface function
  // Method controlling gas seeking and buying behavior.
  // Set done_it = 1 if in midst of moving to gas station, otherwise 0.
  void check_gas (int& done_it);

  //auxiliary function
  // Performs the actual buying of the gas
  void BuyGas (int& done_it);
};
```

Figure 22: Base classes representing genes along with the derived from them subclasses representing strategies. (cont.)

```
//Derived strategy subclass of the "Gas" gene
class Jeff2gas : public Gas {
public:
  //auxiliary function
  Jeff2gas (Truck *tr) { truck = tr; }
private:
  //Implements gene's interface function
  // Method controlling gas seeking and buying behavior.
  // Set done_it = 1 if in midst of moving to gas station, otherwise 0.
  void check_gas (int& done_it);
};



//Base class representing "Trade" gene
class Trade {
public:
  //Gene's interface function
  virtual void trade (int& done_it);
protected:
  Truck *truck;
};

//Derived strategy subclass of the "Trade" gene
class Petetrade : public Trade {
public:
  //auxiliary function
  Petetrade (Truck *tr) { truck = tr; }
private:
  //Implements gene's interface function
  void trade (int& done_it);
};
```

Figure 23: Base classes representing genes along with the derived from them subclasses representing strategies. (cont.)

```
//Derived strategy subclass of the "Trade" gene
class Jefftrade : public Trade {
public:
  //auxiliary function
  Jefftrade(Truck *tr){ truck = tr; }
private:
  //Implements gene's interface function
  void trade (int& done_it);
};

//Derived strategy subclass of the "Trade" gene
class Debtrade : public Trade {
public:
  //auxiliary function
  Debtrade(Truck *tr){ truck = tr; }
private:
  //Implements gene's interface function
  void trade (int& done_it);
};

//Derived strategy subclass of the "Trade" gene
class Deb2trade : public Trade {
public:
  //auxiliary function
  Deb2trade(Truck *tr){ truck = tr; }
private:
  //Implements gene's interface function
  void trade (int& done_it);
};
```

Figure 24: Base classes representing genes along with the derived from them subclasses representing strategies. (cont.)

```
//Base class representing "Buy" gene
class Buy {
public:
  //Gene's interface function
  virtual void go_buy_it (int& done_it);
protected:
  Truck *truck;
};

//Derived strategy subclass of the "Buy" gene
class Petebuy : public Buy {
public:
  //auxiliary function
  Petebuy(Truck *tr){ truck = tr; }
private:
  //Implements gene's interface function
  void go_buy_it (int& done_it);
};

//Derived strategy subclass of the "Buy" gene
class Jeffbuy : public Buy {
public:
  //auxiliary function
  Jeffbuy(Truck *tr){ truck = tr; }
private:
  //Implements gene's interface function
  void go_buy_it (int& done_it);
};
```

Figure 25: Base classes representing genes along with the derived from them subclasses representing strategies. (cont.)

```
//Derived strategy subclass of the "Buy" gene
class Debbuy : public Buy {
public:
  //auxiliary function
  Debbuy(Truck *tr){ truck = tr; }
private:
  //Implements gene's interface function
  void go_buy_it (int& done_it);
};

//Derived strategy subclass of the "Buy" gene
class Alex1buy : public Buy {
public:
  //auxiliary function
  Alex1buy(Truck *tr){ truck = tr; }
private:
  //Implements gene's interface function
  void go_buy_it (int& done_it);
};

//Derived strategy subclass of the "Buy" gene
class Alex2buy : public Buy {
public:
  //auxiliary function
  Alex2buy(Truck *tr){ truck = tr; }
private:
  //Implements gene's interface function
  void go_buy_it (int& done_it);

  //auxiliary function
  void do_buy(int& done_it);
};
```

Figure 26: Base classes representing genes along with the derived from them subclasses representing strategies. (cont.)

```
//Derived strategy subclass of the "Buy" gene
class Alex3buy : public Buy {
public:
  //auxiliary function
  Alex3buy(Truck *tr){ truck = tr; }
private:
  //Implements gene's interface function
  void go_buy_it (int& done_it);

  //auxiliary function
  void do_buy(int& done_it);
};


//Base class representing "Sell" gene
class Sell {
public:
  //Gene's interface function
  virtual void go_sell_it (int& done_it);
protected:
  Truck *truck;
};

//Derived strategy subclass of the "Sell" gene
class Petesell : public Sell {
public:
  //auxiliary function
  Petesell(Truck *tr){ truck = tr; }
private:
  //Implements gene's interface function
  void go_sell_it (int& done_it);
};
```

Figure 27: Base classes representing genes along with the derived from them subclasses representing strategies. (cont.)

```
//Derived strategy subclass of the "Sell" gene
class Debsell : public Sell {
public:
  //auxiliary function
  Debsell(Truck *tr){ truck = tr; Nbr_of_tries_to_sell = 0; }
private:
  //Implements gene's interface function
  void go_sell_it (int& done_it);
  int Nbr_of_tries_to_sell;
};

//Derived strategy subclass of the "Sell" gene
class Deb2sell : public Sell {
public:
  //auxiliary function
  Deb2sell(Truck *tr){ truck = tr; Nbr_of_tries_to_sell = 0; }
private:
  //Implements gene's interface function
  void go_sell_it (int& done_it);
  int Nbr_of_tries_to_sell;
};


//Derived strategy subclass of the "Sell" gene
class Alex1sell : public Sell {
public:
  //auxiliary function
  Alex1sell(Truck *tr){ truck = tr; }
private:
  //Implements gene's interface function
  void go_sell_it (int& done_it);
};
```

Figure 28: Base classes representing genes along with the derived from them subclasses representing strategies. (cont.)

```
//Derived strategy subclass of the "Sell" gene
class Alex2sell : public Sell {
public:
  //auxiliary function
  Alex2sell(Truck *tr){ truck = tr; }
private:
  //Implements gene's interface function
  void go_sell_it (int& done_it);

  //auxiliary function
  void do_sell (int& done_it);
};

//Base class representing "Go" gene
class Go {
public:
  //Gene's interface function
  virtual int go_to (Place dest, int& done_it);
protected:
  Truck *truck;
};

//Derived strategy subclass of the "Go" gene
class Petego : public Go {
public:
  //auxiliary function
  Petego(Truck *tr){ truck = tr; }
private:
  //Implements gene's interface function
  int go_to (Place dest, int& done_it);
};
```

Figure 29: Base classes representing genes along with the derived from them subclasses representing strategies. (cont.)

```
//Base class representing "Move" gene
class Move {
public:
  //Gene's interface function
  virtual void move (int& done_it);
protected:
  Truck *truck;
};

//Derived strategy subclass of the "Move" gene
class Petemove : public Move {
public:
  //auxiliary function
  Petemove (Truck *tr) { truck = tr; }
private:
  //Implements gene's interface function
  void move (int& done_it);
};

//Derived strategy subclass of the "Move" gene
class Jeffmove : public Move {
public:
  //auxiliary function
  Jeffmove (Truck *tr) { truck = tr; }
private:
  //Implements gene's interface function
  void move (int& done_it);
};
```

Figure 30: Base classes representing genes along with the derived from them subclasses representing strategies. (cont.)

```
//Base class representing "Deal" gene
class Deal {
public:
  //Gene's interface function
  virtual void look_for_deal (int& done_it);
protected:
  Truck *truck;
};


//Derived strategy subclass of the "Deal" gene
class Petedeal : public Deal {
public:
  //auxiliary function
  Petedeal (Truck *tr) { truck = tr; }
private:
  //Implements gene's interface function
  void look_for_deal (int& done_it);
};

//Derived strategy subclass of the "Deal" gene
class Jeffdeal : public Deal {
public:
  //auxiliary function
  Jeffdeal (Truck *tr) { truck = tr; Deal_Number = 0; }
private:
  //Implements gene's interface function
  void look_for_deal (int& done_it);
  int Deal_Number;
};
```

Figure 31: Base classes representing genes along with the derived from them subclasses representing strategies. (cont.)

```
//Derived strategy subclass of the "Deal" gene
class Debdeal : public Deal {
public:
  //auxiliary function
  Debdeal (Truck *tr) { truck = tr; }
private:
  //Implements gene's interface function
  void look_for_deal (int& done_it);
};

//Derived strategy subclass of the "Deal" gene
class Jeff2deal : public Deal {
public:
  //auxiliary function
  Jeff2deal (Truck *tr) { truck = tr; }
private:
  //Implements gene's interface function
  void look_for_deal (int& done_it);
};


//Dereved strategy subclass of the "Deal" gene
class Alex1deal : public Deal {
public:
  //auxiliary function
  Alex1deal (Truck *tr) { truck = tr; init(); }
private:
  //Implements gene's interface function
  void look_for_deal (int& done_it);

  //auxiliary function
  void init();
  Place idx[NUM_AV*NUM_ST];
  int num_intr;
};
```

Figure 32: Base classes representing genes along with the derived from them subclasses representing strategies. (cont.)

```
//Dereved strategy subclass of the "Deal" gene
class Alex2deal : public Deal {
public:
  //auxiliary function
  Alex2deal (Truck *tr) { truck = tr; init(); }
private:
  //Implements gene's interface function
  void look_for_deal (int& done_it);

  //auxiliary function
  //Returns the number of trucks that are within the
  //distance "dist" from the intersection "target"
  int num_clossest_trucks(long dist, Place target);

  //auxiliary function
  void init();
  Place idx[NUM_AV*NUM_ST];
  float sell_factor[MAX_TRUCKS+1]; float buy_factor[MAX_TRUCKS+1];
  int num_intr;
};

//Dereved strategy subclass of the "Deal" gene
class Alex3deal : public Deal {
public:
  //auxiliary function
  Alex3deal (Truck *tr) { truck = tr; last_update = -10000L; init(); }
private:
  //Implements gene's interface function
  void look_for_deal (int& done_it);

  //auxiliary function
  //Returns the number of trucks that are within the
  //distance "dist" from the intersection "target"
  int num_clossest_trucks(long dist, Place target);

  //auxiliary function
  //Updates the information about dealers, updating as
  //many intersections as the available capital and slot time left allow.
  void update_dealer_info();

  //auxiliary function
  void init();
  Place idx[NUM_AV*NUM_ST];
  int num_intr;
  float sell_factor[MAX_TRUCKS+1]; float buy_factor[MAX_TRUCKS+1];
  long last_update; //time of the last dealer_info update
};
```

```
//Dereved strategy subclass of the "Deal" gene
class Alex4deal : public Deal {
public:
  //auxiliary function
  Alex4deal (Truck *tr) { truck = tr; last_update = -10000L; init(); }
private:
  //Implements gene's interface function
  void look_for_deal (int& done_it);
  //auxiliary function
  //Updates the information about dealers, updating as
  //many intersections as the available capital and slot time left allow.
  void update_dealler_info();
  //auxiliary function
  void init();
  Place idx[NUM_AV*NUM_ST]; int num_intr;
  long last_update; //time of the last dealer_info update
};

//Base class representing "Borrow" gene
class Borrow {
public:
  //Gene's interface function
  virtual void borrow (int& done_it);
protected:
  Truck *truck;
};

//Dereved strategy subclass of the "Borrow" gene
class Alex1borrow : public Borrow {
public:
  //auxiliary function
  Alex1borrow(Truck *tr){ truck = tr; }
private:
  //Implements gene's interface function
  void borrow (int& done_it);
};

//Dereved strategy subclass of the "Borrow" gene
class Alex2borrow : public Borrow {
public:
  //auxiliary function
  Alex2borrow(Truck *tr){ truck = tr; }
private:
  //Implements gene's interface function
  void borrow (int& done_it);
};
```

Figure 34: Base classes representing genes along with the derived from them subclasses representing strategies. (cont.)

```
//Dereved strategy subclass of the "Borrow" gene
class Alex3borrow : public Borrow {
public:
  //auxiliary function
  Alex3borrow(Truck *tr){ truck = tr; }
private:
  //Implements gene's interface function
  void borrow (int& done_it);
};


//Base class representing "Lend" gene
class Lend {
public:
  //Gene's interface function
  virtual void lend(Control* requester, long &amount);
  //Gene's interface function
  virtual void update_rate();
protected:
  Truck *truck;
};

//Dereved strategy subclass of the "Lend" gene
class Alex1lend : public Lend {
public:
  //auxiliary function
  Alex1lend(Truck *tr){ truck = tr; last_num_requests=0; last_date=0;
                        last_waiting_period=1; cur_num_requests=0; }
private:
 //Implements gene's interface function
 void lend(Control* requester, long &amount);
 //Implements gene's interface function
 void update_rate();
 long last_num_requests, last_date, last_waiting_period, cur_num_requests;
};
```

Figure 35: Base classes representing genes along with the derived from them subclasses representing strategies. (cont.)

```
//Dereved strategy subclass of the "Lend" gene
class Alex2lend : public Lend {
public:
  //auxiliary function
  Alex2lend(Truck *tr) { truck = tr; last_num_requests=0; last_date=0;
                         last_waiting_period=1; cur_num_requests=0; }
private:
 //Implements gene's interface function
 void lend(Control* requester, long &amount);
 //Implements gene's interface function
 void update_rate();
 long last_num_requests, last_date, last_waiting_period, cur_num_requests;
};

//Dereved strategy subclass of the "Lend" gene
class Alex3lend : public Lend {
public:
  //auxiliary function
  Alex3lend(Truck *tr){ truck = tr; }
private:
  //Implements gene's interface function
  void lend(Control* requester, long &amount);
  //Implements gene's interface function
  void update_rate();
};

//Base class representing "Payer_back" gene
class Payer_back {
public:
  //Gene's interface function
  virtual void pay_back  (int& done_it);
protected:
  Truck *truck;
};
```

Figure 36: Base classes representing genes along with the derived from them subclasses representing strategies. (cont.)

```
//Dereved strategy subclass of the "Payer_back" gene
class Alex1payer_back : public Payer_back {
public:
  //auxiliary function
  Alex1payer_back(Truck *tr) { truck = tr; }
private:
  //Implements gene's interface function
  void pay_back (int& done_it);
};

//Dereved strategy subclass of the "Payer_back" gene
class Alex2payer_back : public Payer_back {
public:
  //auxiliary function
  Alex2payer_back(Truck *tr) { truck = tr; }
private:
  //Implements gene's interface function
  void pay_back (int& done_it);
};

//Dereved strategy subclass of the "Payer_back" gene
class Alex3payer_back : public Payer_back {
public:
  //auxiliary function
  Alex3payer_back(Truck *tr) { truck = tr; }
private:
  //Implements gene's interface function
  void pay_back (int& done_it);
};
```

Figure 37: Base classes representing genes along with the derived from them subclasses representing strategies. (cont.)