Investigating the Value of Information and Computational Capabilities by Applying Genetic Programming to Supply Chain Management

Scott A. Moore
University of Michigan Business School

Kurt DeMaagd
University of Michigan Business School

Working Paper 03-004

Investigating the value of information and computational capabilities by applying genetic programming to supply chain management

Scott A. Moore
Associate Professor
University of Michigan Business School
samoore@umich.edu

Kurt DeMaagd
Ph.D. Student
University of Michigan Business School
demaagdk@umich.edu

January 10, 2003

Abstract

In this paper we describe a research project centering on experiments in which game-playing evolving agents are used to investigate the value of information. Specifically, in these experiments we define populations of agents whose strategies evolve in order to become better managers of different types of supply chains. The agents evolve their strategies in order to minimize costs (either for themselves or for their value chain). We describe several different experiments in which we will vary the abilities of agents both to gather and to store more information. Part of the results of this project will be related to the value of information and computational capabilities: Is it always better to have more information? If not, what are the conditions under which less information is better? The culminating experiment is one in which evolving agents compete to sell information to other evolving agents playing their roles in a supply chain.

1 Introduction

It is generally believed that having more information available to make a decision is a good thing; however, in today's computing saturated business

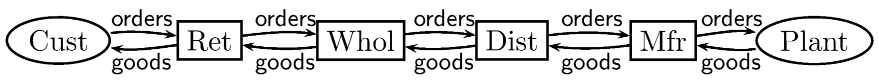


Figure 1: Basic set-up of supply chain

world information overload eventually occurs. Further, given that it costs money to manage information, care should be taken when the decision is made to gather an additional piece of information. The general strategy we are using in the research program described in this paper is to simulate a business problem, varying parameters so that players are placed under differing requirements for information and computational capabilities. (For the rest of the paper, when it will not cause confusion, we will use the term "information" to refer to "information and computational capabilities.") The players who participate in these simulations will be created computationally through the use of a genetic program [5]. Genetic programming is a computational approach to managing the evolution of agents whose fitness to live is based on their ability, relative to other agents in the population, to compute some function. We are using genetic programming in this experiment as a means for investigating the information needs of the participants — the abilities both to gather and to make use of information — as they vary with the simulation's complexity. At the end of the series of experiments we will also look into how effectively evolving agents are able to sell information to the agents competing in the supply chain. A secondary result of this research program will be an increased understanding of how sensitive the genetic program is to its initial parameters, evidence showing how varying certain genetic program-related parameters affect the experimental results, and how well a genetic program can navigate a large search space. The specific business situation we will use in this research program is a multi-level supply chain game (as exemplified by Sterman's beer game [7]). A final result of this project will be insight into how well solutions found by a genetic program compare with solutions found by other methods in the management science and information systems literature. This paper describes the research program we are undertaking on this subject.

In the initial experiments described in this paper, agents evolve in several separate populations that correspond to the different roles in the beer game, namely retailer, wholesaler, distributor, and manufacturer. (See Figure 1.) The standard scenario is one in which an agent can only send orders to one agent who is one step upstream (referring to the flow of goods) and can only send goods to one agent who is one step downstream. Demand is determined by an exogenous customer who both sends orders to the retailer and ultimately receives final delivery of the goods. The times for orders and goods to arrive at their destinations are taken from distributions that are not known by the players. The goods are manufactured at the plant, another exogenous player.

The object of evolution on these populations is to find a strategy for ordering goods, for each player, that minimizes its costs when playing the game with other agents. People are notoriously bad at playing this game even at the simplest possible settings; an average player's performance ends up with costs ten times worse than optimal [8]. Further, analytical techniques can provide solutions to this problem when the settings are simple but are unable to when it becomes more complex [1].

We have several research questions in mind when looking at what evolves in this environment, some related to the genetic programming tool itself and others related to the supply chain management process. We will not summarize them all, but there are generally two classes of questions. The first relates to determining if a genetic program will be able to evolve agents that use optimal or near-optimal re-stocking strategies. We will be playing a great variety of supply chain games, varying many parameters, including the number of players at each level of the supply chain, the information available to the players, the speed at which goods are shipped, and the pattern of customer demand. We believe that some scenarios will prove to be quite simple for the genetic program while others will prove more demanding. Kimbrough & Wu [4] successfully used genetic algorithms (a related technique) to evolve agents to play this game, though the search space for their solutions is much constrained relative to our proposal.

The second set of questions relate to looking at the value of information. We will be looking for situations in which one piece of information ends up determining the eventual success of a player. Conversely, we will also be looking for situations in which the volume of information overwhelms a population and keeps any one member from finding successful strategies. We will also be looking for ways we can value information. In one set of experiments we will put costs on the usage of certain information in order to see how the population reacts. In another set of experiments we will add a competing population that evolves pricing strategies for information in order to determine if the pricing agents can determine appropriate and sustainable prices for information without adversely affecting the performance of the players.

We wrote the program that runs the simulations for this project using

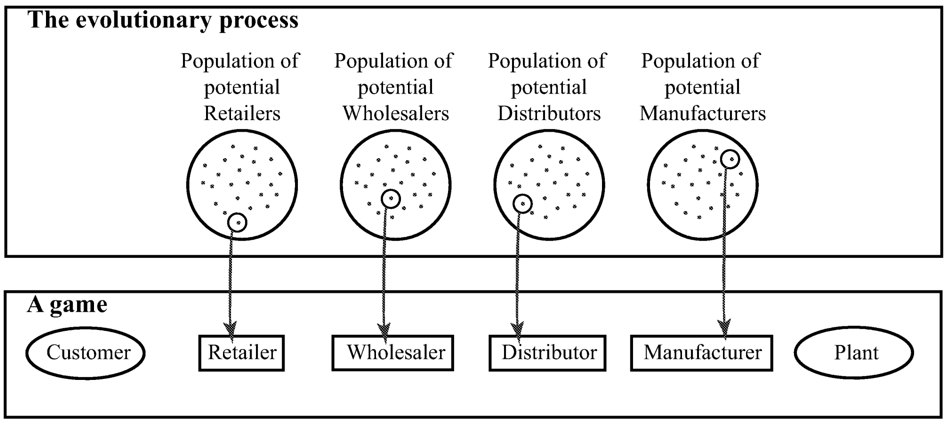

The evolutionary process
Population of potential Retailers
Population of potential Wholesalers
Population of potential Distributors
Population of potential Manufacturers
A game
Customer
Retailer
Wholesaler
Distributor
Manufacturer
Plant

Figure 2: The relationship between the evolutionary process and the supply chain games

GNU CLISP [2]. The source code for our program is available at SourceForge [6] and runs on Windows, Linux, and Unix machines. In its current state of development it is able to generate the populations, play the games with a limited set of parameters, breed a new generation based on the performance statistics of the existing generation, and generate some statistics while it is running. Much remains but the foundation of the program is completed.

2 The evolutionary process

In this section we describe the basic set-up of the evolutionary process, the game played by the members of the population for the purposes of determining the fitness of each population member, and the representation of each player's re-stocking strategy.

2.1 Basic set-up

We will vary the experiment somewhat throughout this project but the following describes a standard set-up. (See Figure 2.) This experiment has Φ separate, co-evolving populations of N_ϕ agents for each role in the game; in Figure 2 the dots in the circles represent the members of the population. In each generation of the process, each agent plays at least M games (this is a parameter set by the researcher), and it is the agent's performance in these games that determines its fitness. For any one of these games, a random

```
Create each population of agents
gen = 1
repeat
   for r=1 to R
      Choose agent i of type r at least M times
         Choose 1 agent at random from each of the other agent
    roles
         Play the game for W weeks
         Record the outcome for each agent
      end-loop
   end-for
   Calculate fitness scores for all agents
   Breed the next generation of each population
   gen = gen + 1
```

Figure 3: Evolutionary scenario

member of each population is chosen to play its specified role (as shown in Figure 2). At the end of the game, the agent's score is recorded. We use several different fitness functions but a standard one is to minimize an agent's costs. One game is usually played for $W = 35$ weeks (i.e., *turns*), though we will vary this to keep the agents from over-specifying. After all games are played, the program calculates fitness scores for all the agents in all the populations. The best players in each generation are then chosen to reproduce and create the next generation of players. This whole process is repeated until some termination condition is met. The overall evolutionary scenario is summarized in Figure 3.

2.2 The fitness function

The driver of genetic programming is the fitness function. This determines which agents have a better chance of generating offspring for the next generation. The basic parameters shown in Figure 4 are needed for the following discussion. (Note that all the terms defined in the following figures refer to one generation; for example, G refers to all the games that occur in one generation.) These are set before an evolutionary scenario is begun.

Figure 5 contains some functions that are useful for discussing the genetic programming process. Consider the first function, $\pi(g, \rho)$. This returns the index j of the player who played role ρ in game g. This function enables

$$\begin{aligned}
\Phi &= \text{number of populations} \\
\phi &= \text{any value in } 1 \ldots \Phi\text{; that is, a particular population} \\
\mathcal{P}_\phi &= \text{a certain population } \phi \\
\mathcal{N} &= \text{number of agents in all populations} \\
&= \sum_{\phi \in \Phi} \mathcal{N}_\phi \\
\mathcal{N}_\phi &= \text{number of agents in population } \phi \\
j &= \text{a particular member of some population} \\
\mathcal{R} &= \text{the roles in a game: } R, W, D, M \\
\rho &= \text{any value in } \mathcal{R}\text{; that is, a particular role} \\
W &= \text{number of weeks} \\
w &= \text{any value in } 1 \ldots W\text{; that is, a specific week} \\
G &= \text{number of games played in a particular generation} \\
g &= \text{any value in } G\text{; that is, a particular game} \\
H_{g,\rho} &= \text{inventory holding costs per item in game } g \text{ for the role } \rho \\
P_{g,\rho} &= \text{penalty cost per backordered item in game } g \text{ for the role } \rho
\end{aligned}$$

Figure 4: Definitions of standard parameters

$$\begin{aligned}
\pi(g,\rho) &= \text{the index } j \text{ of the player who played role } \rho \text{ in game } g \\
\phi(j,\rho) &= \text{the set of game numbers in which player } j \text{ played role } \rho \\
\mu(j,\rho) &= \text{rthe number of games in which player } j \text{ played role } \rho \\
\eta(\rho) &= \text{the population number that players in} \\
&\ \text{role } \rho \text{ are taken from}
\end{aligned}$$

Figure 5: Useful functions

us to tie the score earned by a player in a role back to a specific member j of the population — which, in turn, enables us to assign fitness scores and then reproduce a new generation of players. You will see this function π used in many variables. The function $\phi(j, \rho)$ is basically the inverse of the π function: given the index j identifying the population member and the specific role, the function returns a set of game numbers for which this agent fulfilled this role. The next function, $\mu(j, \rho)$, returns the number of games in which player j played role ρ. This will end up being different values for different members of the population in a particular generation because of the way by which members are randomly selected for games.

The last function in this table, $\eta(\rho)$, is useful when there is more than one population of agents (as there is in the scenario we are currently examining; see Figure 2). (If there is only one population, then this function would always return 1.) In our conception of this genetic programming investigation, at the beginning of the evolutionary scenario, each population of agents is created and assigned to a particular role in the game. Thus, if you know the role in the game, you know the population that any player in that role came from. This function captures this information.

Figure 6 describes important weekly status parameters for the supply chain game and how they are related to each other. These values are calculated and stored for each player for each week for each game for each generation. In most games the information for a particular player will be available to only that player. In some games the player will know information about all prior weeks of the game while in other games the player will only know about the current week. All but the last two parameters in this figure are relatively straight-forward. Both of these are related to the problem of coming up with a measure for the whole value chain. The first, $K_{g,w}$, defines a simplistic way of calculating the costs for a value chain for one week of a game: add up everyone's cost. Consider, though, if all of the members of a value chain were under one management so that both inter-agent backorders and the location of inventory were irrelevant; if this were the case, then $\kappa_{g,w}$ would be the preferred measure.

Figure 7 defines various fitness measures whose values are based on these variables. $A_{\pi(g,\rho)}$ defines the total costs for the player in role ρ in game g; it is the sum of the player's costs (defined in Figure 6) for each week of the game. Take a closer look at the name of this variable: $A_{\pi(g,\rho)}$. This uses the function π that we discussed above. When we have a value for a specific $A'_{\pi(g',\rho')}$ — that is, when we are considering a specific role ρ' in a specific game g' — the π function tells us the specific member of the population j'

$$
\begin{aligned}
B_{\pi(g,\rho),w} &= \text{inventory on hand at beginning of week } w \text{ for the player} \\
&\ \text{in role } \rho \text{ in game } g \\
S_{\pi(g,\rho),w} &= \text{shipment received in week } w \text{ by the player in role } \rho \\
&\ \text{in game } g \\
D_{\pi(g,\rho),w} &= \text{demand received (that is, the number of goods ordered)} \\
&\ \text{in week } w \text{ by the player in role } \rho \text{ in game } g \\
I_{\pi(g,\rho),w} &= \text{inventory position in week } w \text{ for the player in role } \rho \\
&\ \text{in game } g \\
&= B_{\pi(g,\rho),w} + S_{\pi(g,\rho),w} - D_{\pi(g,\rho),w} \\
O_{\pi(g,\rho),w} &= \text{backordered items in week } w \text{ for the player in role } \rho \\
&\ \text{in game } g \\
&= \begin{cases} 0 & \text{if } I_{\pi(g,\rho),w} \geq 0 \\ -I_{\pi(g,\rho),w} & \text{if } I_{\pi(g,\rho),w} < 0 \end{cases} \\
C_{\pi(g,\rho),w} &= \text{costs in week } w \text{ for the player in role } \rho \text{ in game } g \\
&= \begin{cases} I_{\pi(g,\rho),w} \times H_{g,\rho} & \text{if } I_{\pi(g,\rho),w} \geq 0 \\ O_{\pi(g,\rho),w} \times P_{g,\rho} & \text{if } I_{\pi(g,\rho),w} < 0 \end{cases} \\
K_{g,w} &= \text{total costs in game } g \text{ in week } w \text{ for a value chain} \\
&= \sum_{\rho \in \mathcal{R}} C_{\pi(g,\rho),w} \\
\kappa_{g,w} &= \text{aggregate costs in game } g \text{ in week } w \text{ for a value chain} \\
&= P_{g,\rho} \times O_{\pi(g,R),w} + \sum_{\rho \in \mathcal{R}} (H_{g,\rho} \times I_{\pi(g,\rho),w})
\end{aligned}
$$

Figure 6: Important status parameters for a week in a game

$$\begin{aligned}
A_{\pi(g,\rho)} &= \text{total costs for the player in role } \rho \text{ in game } g \\
&= \sum_{w=1}^{W} C_{\pi(g,\rho),w} \\
\mathcal{D}_{g,\rho} &= \text{total demand in game } g \text{ for role } \rho \\
&= \sum_{w=1}^{W} D_{\pi(g,\rho),w} \\
\Upsilon_{\pi(g,\rho)} &= \text{normalized average fitness} \\
&= \frac{A_{\pi(g,\rho)}}{\mathcal{D}_{g,\rho}} \\
A^{D}_{g,\rho} &= \text{the total costs for a player in role } \rho \text{ facing the demand} \\
&\quad\ \text{pattern used in game } g \text{ if its strategy is to always order } D \\
\omega_{\pi(g,\rho)} &= \text{relative average fitness} \\
&= \frac{A_{\pi(g,\rho)}}{A^{D}_{g,\rho}} \\
T_g &= \text{total costs in game } g \text{ for a value chain} \\
&= \sum_{w=1}^{W} K_{g,w} \\
\Theta_g &= \text{total aggregate costs in game } g \text{ for a value chain} \\
&= \sum_{w=1}^{W} \kappa_{g,w}
\end{aligned}$$

Figure 7: Various fitness measures for an agent in a game

that played in the game; thus, the total costs A' that the player incurred in that role in that game is able to get recorded with the proper player.

Depending on the demand function used when running the evolutionary process under different parameter settings, it is possible for players in the same population to play games with different demands in different generations. As a result, it can be difficult to use A to compare the fitness scores between different generations. For example, players in two different generations might play similar games except that the first player faces an average demand of 10 while the second player faces an average demand of 100. Both players may have sub-optimal strategies — for example only ordering 50% of demand — but the second player will appear to have a worse strategy. It will probably have backorder costs that are a factor of 10 higher than the first player's.

The *normalized average fitness score*, $\Upsilon_{\pi(g,\rho)}$, resolves this issue. Υ is defined as a player's total costs divided by the sum of its weekly demands. The resulting score is a fitness score per unit of demand. Returning to the previous example, the first player had a backorder cost of 5 and a demand of 10, resulting in a Υ of .5. Likewise, player two had a backorder cost of 50 and a demand of 100, resulting in a Υ of .5. This successfully resolves the problem of differences in demand artificially skewing the fitness scores.

It will also be convenient to have a baseline score against which to measure the quality of a strategy. In this case $A^D_{g,\rho}$ is our baseline. The D strategy of placing order equal to demand is a simple and intuitively appealing baseline strategy which is also the optimal strategy in simplistic scenarios (as pointed out in [4] among others). Given this baseline, it is possible to measure the *relative average fitness*, ω; it is $A_{\pi(g,\rho)}$ divided by the baseline. The resulting ratio will give a comparison of the strategy's effectiveness versus the baseline D strategy. The measure ω will provide a convenient way to compare strategies across generations.

In addition to looking at the performance of individual players, we also can examine the performance of the entire supply chain for a whole game. The two ways to do this, as represented by T_g and Θ_g, are based on the two different ways of measuring weekly value chain performance, $K_{g,w}$ and $\kappa_{g,w}$, respectively.

Figure 8 describes several additional fitness measures as defined by Koza [5]. The simplest measure is τ', the game-based raw fitness. The value of this can come from the fitness measures defined in Figure 8, such as A, Υ, ω, T, or Θ. However, an agent's raw fitness for a generation, $\tau_{j,\rho}$, can be a bit more complicated than simply using an agent's raw fitness score from one game. There are some cases in which the agent has the opportunity to play

$$
\begin{aligned}
{\tau'}_{\pi(g,\rho)} &= \text{game-based raw fitness; this can be any one of } A, \\
&\ \Upsilon, \omega, T, \text{ or } \Theta \\
\tau_{j,\rho} &= \text{an agent } j\text{'s raw fitness in a role } \rho \text{ over all of its} \\
&\ \text{games in this generation} \\
&= \frac{1}{\mu(j,\rho)} \sum_{g \in \phi(j,\rho)} {\tau'}_{\pi(g,\rho)} \\
\beta_\rho &= \text{best score for any player in role } \rho \text{ in this generation} \\
&= \min_{j \in \mathcal{P}_{\eta(\rho)}} \tau_{j,\rho} \\
\sigma_{j,\rho} &= \text{standardized fitness for the player } j \text{ in role } \rho \\
&= A_{j,\rho} - \beta_\rho \\
\alpha_{j,\rho} &= \text{adjusted fitness for agent } j \text{ in role } \rho \\
&= \frac{1}{1 + \sigma_{j,\rho}} \\
\nu_{j,\rho} &= \text{normalized fitness for agent } j \text{ in role } \rho \\
&= \frac{\alpha_{j,\rho}}{\sum_{j \in \mathcal{P}_{\eta(\rho)}} \alpha_{j,\rho}}
\end{aligned}
$$

Figure 8: Koza's standard fitness measures

multiple games. For example, in the evolutionary scenarios in which there are multiple populations, we will sometimes have the agent play games with multiple different, random opponents. The hypothesis is that this will reduce the chances that a high-fitness agent will be dropped from the population because it played in a game with a poor value chain partner.

The genetic program needs standardized fitness, $\sigma_{j,\rho}$, to have values such that smaller numbers are better, and the best score is zero [5, p. 96]. All of the measures defined in Figure 7 are based on costs, which should be minimized. Therefore, as lower is already better, standardized fitness is equal to raw fitness (with the β_ρ adjustor needed to set the best score to 0).

The adjusted fitness for the player, α, converts the standardized fitness into a number between 0 and 1. As in the case of the standardized fitness, this will result in a score where smaller values are better. It also tends to create a more pronounced distinction between scores with similar values. Normalized fitness, ν, is similar to the adjusted fitness, except larger scores are better. Like adjusted fitness, the values will range from 0 to 1 and it tends to further emphasize the difference between values.

When evolving agents, we will use all of the possibilities for the game-based raw fitness, τ', to calculate an agent j's score for any particular game, and will use the normalized fitness based on that τ', ν, as the agent's fitness value for that generation. We hypothesize that ω will be the most useful of all of our τ' possibilities. We will report all of the τ' values for every generation no matter whether or not that specific τ' is being used as the basis.

2.3 Specifying a strategy

We have discussed how the game is played, how the evolution proceeds, and how agents are measured. We now examine how strategies are defined and how new ones evolve from old ones.

2.3.1 Defining a strategy

Each member's strategy is represented by an S-expression. An S-expression is a set of terms composed into a tree. Consider the example shown in Figure 9. The standard mathematical expression can be translated in a fairly straight-forward manner into the S-expression; this can, in turn, be mapped onto a tree representation. The program that we have written manipulates S-expressions but the tree representation can sometimes be easier to use in exposition so we will generally use that representation in this paper. The

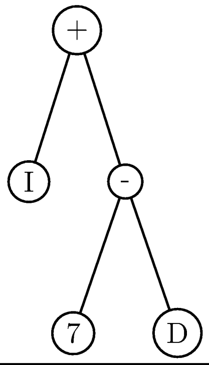
Standard mathematical representation	`I + (7 - D)`
S-expression	`(+ I (- 7 D))`
Equivalent tree representation	

Figure 9: Equivalent representations

terms in the S-expression can be either terminals or functions. Terminals are either constants or variables, such as 187 or D. A function takes a specified number of parameters, performs an operation on those parameters, and then returns the result. For example, the function `(sum 3 4)` will add 3 and 4 and return the result. Parameters of a function can be both terminals and other functions.

When coming up with a set of functions to include in a genetic program, it is desirable that every function satisfy the *closure* property [5, p. 86]. To meet this requirement, all functions must accept as a parameter any possible value in the set composed of all terminals and any value returned by any function. Thus, no matter what allowed transformations that might be performed on this function (see the discussion of genetic operations below), it should not encounter errors in execution. The classic example is division by zero. Since this would generate an error, a special division operator must be created to return a pre-specified value if the denominator is zero.

Functions must also satisfy the *sufficiency* property if we want the genetic program ultimately to find the answer to the problem at hand. This states that functions and terminals provided must be able to express a solution to the problem. For example, in our experiments with the beer game and MIT demand, we know that the optimal solution is based on the demand. If we failed to provide the demand as a terminal, the program would at best be able to find only an approximation of the optimal solution.

The terminals and functions are all listed in §A. The terminals consist of a set of integers plus information that the agent knows about itself and the game: amount of its own inventory, amount that it has on order, its own current demand, plus the current week number of the game. There is nothing that says that the appropriate terminals or functions have to be easy to find, or that they have to be combined in some straight-forward way, or that the researcher even knows *how* the terminals and functions should be combined in order to reach the best solution. In fact, part of what this research program is attempting to do is to determine how difficult the answer can be to find while still enabling the genetic program to succeed. Further, some of the problems that we will pose to the genetic program have no known optimal solution so we will merely be searching for "best that we can find." In this case, we will investigate the value of information and computational capabilities and how well the agents are able to take advantage of them.

The functions currently under consideration come in three groups (see Appendix A): mathematical, logical, and informational. The mathematical set includes fairly standard operations: addition, subtraction, multiplication, integer division, minimum, maximum, and sine. The logical functions are also the ones you might expect: and, or, not, if/then, if/then/else, greater than, less than, and equal to.

The informational functions are a bit more interesting. The function `(mydem y)` retrieves a player's own demand `y` weeks ago while `(myinv y)` does the same thing for inventory and `(myord y)` for orders. The functions `(dem x y)`, `(inv x y)`, and `(ord x y)` are generalizations of these three functions that provide access to this information for other players. These functions will not be available to all functions in all scenarios. Some of our experiments will be directed at discovering how useful the players find these functions, and whether or not some functions allow some population members to dominate over other population members.

2.3.2 Evolving a new strategy

Genetic programs generate new strategies using many of the same principles of Darwinian evolution. It is essentially a matter of selecting certain individuals with superior fitness, performing a set of genetic operations, and adding them to a new population of individuals. In this way, the genetic program should eventually evolve better solutions to the problem.

The first problem is to select superior individuals. In §2.2 we described the different means of rating an agent. The next problem is to select individuals with higher rankings without destroying the diversity of the genetic material that may contain components of the optimal solution. Three primary methods are used in genetic programming: fitness proportionate, rank, and tournament.

Fitness proportionate selection is the most popular method for selection [3]. This method effectively treats the fitness score of the individual as the probability that it will be selected. The easiest means of calculating this probability for an agent j is to set it equal to the normalized fitness, $\nu_{j,\rho}$. The normalized fitness is already a measure between 0 and 1 with greater scores being better, and the sum of all νs is 1.0.

While popular, this has a significant weakness. If a particular agent or small group of agents has a relatively high fitness, they may quickly dominate the entire population. This can reduce the diversity of the population which may limit the ability to find an optimal solution. One means of resolving this is to use *rank* selection. Each individual is assigned a rank based on its score, ν. New members of the population are selected from the set of highly ranked members. This has the effect of reducing the premature convergence of fitness proportionate selection while still favoring strong individuals.

Tournament selection is the computational equivalent of two individuals competing for a mate. Two agents are selected at random from the population. The agent with a superior fitness score is selected. Once the agent is selected, one of several genetic operations is performed on it. The goal of this phase is to create a new population that, ideally, will be superior to the previous population. Three of the most common operations used in genetic programming are reproduction, crossover, and mutation.

The simplest one of these is *reproduction*. The selected individual is simply copied into the new population. This results in the stronger elements of the previous population becoming the new population. While this will create a stronger population, the system will never find any members with superior fitness than those in the original population. To resolve this issue, members of the population must be modified in some way.

The most common method of evolving the individuals is *crossover*. This is the genetic programming equivalent of sexual reproduction. It takes material from two parents and uses it to create two new children. The first step is to select two individuals, using one of the selection methods described above. These two agents will be the parents.

The next step is to perform the actual crossover. Recall that each parent can be represented as a tree (Figure 9). To perform the crossover, each parent swaps part of its tree with the other parent. This is done by selecting a random point in each parent. These points become the crossover points. The subtrees under these points are swapped between the two parents (see Figure 10). The resulting new trees are then inserted into the new popu-

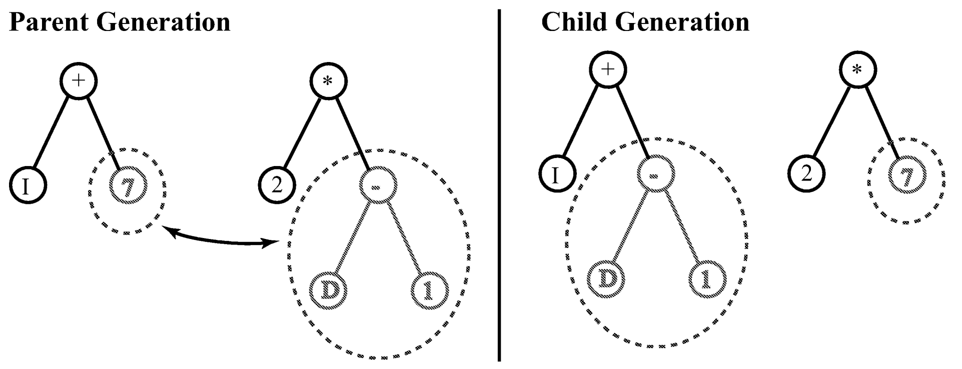


Figure 10: An example of crossover

lation. It is possible to select the same tree twice so that it serves as both parents. In such a case, however, the resulting children will most likely not be clones of the parents. To get a clone, the crossover operation would have to select the same crossover point on both trees. Since all points are equally likely to be selected, the probability of this occurring is minimal.

While the trees shown in Figure 10 are relatively simple, this same process occurs even with more complex trees. This crossover operation can grow and shrink the size of the tree. In the event that the tree grows to a depth greater than the maximum depth allowed, the crossover operation is cancelled and the first parent is inserted into the new population.

The final option is *mutation.* When a particular population appears to be prematurely converging on a solution, it may be desirable to inject some diversity into the system. As the title implies, mutation introduces a random change into the system. It does this by selecting a random point in a selected tree, removing that point and all elements below it, and then inserting a new random tree into its place. The crossover operation in genetic programming, however, is highly effective at finding new solutions. Therefore, mutation is very rarely used.

3 Plan of investigation

In this research project we are completing four series of experiments that will generally build upon the results of each prior series. The first set of experiments will focus on fine-tuning the parameters of the beer game and the genetic program so that the evolution process will go better. The second set of experiments will focus on parameters of the genetic program that might significantly affect how well the evolutionary process progresses in our later, more complex, experiments. The third set of experiments are the centerpiece of our investigation into information value and information overload. The fourth set of experiments will build on this last set of experiments. We will use what we learn from the third set to define future investigations, but we list here a reasonable set of possibilities.

3.1 Given parameters

In the following experiments we will use default values for some parameters: probability of permutation (set to 0) [5, p. 107], probability of editing (0) [5, p. 108], and the probability of encapsulation (0) [5, p. 110]. Koza discussed each of these but used them little, if at all, in his experiments. The *variety* of a population is "the percentage of individuals for which no exact duplicate exists elsewhere in the population" [5, p. 93]. Our program checks for duplicates so the variety will be 100%. We do this so as to ensure the greatest amount of genetic building blocks enter our population with the hopes that this will increase the genetic program's probability of success. As is typical among this type of research, we are only interested in attaining 100% variety in the initial population; after that, duplicates are something that the genetic program naturally creates.

Maximum number of generations is a parameter that is generally set to keep the genetic program from running forever. We have no initial plan for the value of this parameter. Some very difficult problems require many generations (> 500) while simpler problems take far less time to solve (< 50). We will just have to see how effective and efficient the program is at solving this problem.

3.2 Fine-tuning parameters

The following parameters are not central to our investigation but may potentially have some impact on the performance of the evolutionary process. We will perform some cursory experiments to determine how the process performs at the different settings; in these experiments we will take the settings of the parameters in §3.1 as given. We will be looking at two performance measures: 1) **effectiveness**: the effect of the setting on the process's ability to find the best performing solutions, and 2) **efficiency**: the effect of the setting on the speed of the process in its search for the best performing solutions. Given that the process is able to do the first, we will prefer settings that enable more efficient searches. One of the findings that we expect from this research program is that we should be able to describe more completely the trade-off between effectiveness and efficiency in this problem.

Selection method As we discussed above in §2.3.2, the three possible means of selecting individuals for reproduction are fitness proportionate, rank, and tournament.

Probability of crossover This is the percentage of the population on which crossover is performed. Koza uses the value of 0.90 [5, p. 114]. For that portion of the population on which crossover is not performed, reproduction is performed (that is, those members are directly copied into the new population). The purpose of investigating this is to see what combination of crossover and reproduction makes the evolutionary process perform best. The possible values that we will investigate are 0.2, 0.5, 0.8, 0.9, 0.95, and 1.

Percentage of internal crossover points In an S-expression the internal points are the function nodes. If crossover is performed on two external (terminal) points, then this mimics the mutation operation on terminals. We want to encourage the creation of more complex trees so we will investigate relatively large values for this parameter: 0.7, 0.8, 0.9, 0.95, and 1. Koza uses a value of 90% in his investigations [5, p. 114].

Probability of mutation Koza generally does not use the mutation operation [5, p. 114]. We hypothesize that the mutation operation might be useful for helping the genetic program avoid local minima in its search process. We will investigate possible mutation probabilities of 0, 0.01, 0.05, 0.1, and 0.25.

This requires a $3 \times 6 \times 5 \times 5$ (450) completely randomized experimental design be employed in order to assess the impact of the above variables on the effectiveness and efficiency of the genetic program. We will choose a moderately difficult problem for which an optimal (or heuristic) solution is already known. This will allow us to stop the evolutionary process when the solution is found or when the best value gets within some arbitrarily close range of this answer. Effectiveness will be measured for this test with a ratio between the best answer found by the genetic program and the optimal answer. Efficiency will be measured in three ways: 1) time elapsed before the stopping point was reached, 2) the number of generations completed before the stopping point was reached, and 3) the total number of weeks played in all games in all generations.

We will use these results in the following set of investigations.

3.3 Basic investigation

The follow parameters are also not central to our investigation but may potentially significantly affect the performance of the evolutionary process (as opposed to the above parameters that we do not believe should have more than a marginal affect on the process's effectiveness though they might end up having a measurable impact on efficiency). We will use the settings of the parameters in §3.1 as given and will use the values of the parameters that we determined in §3.2.

Number of populations (Φ) As the problem is described in §2, each role is assigned one population and *vice versa.* In those instances in which the parameters for each of the roles are the same, it would be computationally more efficient if all of the players came from one population. Further, it would be even more efficient if one population member were selected and then assigned to all four roles. We want to determine if this parameter's setting has an effect on the process's effectiveness. Possible values of 1 (one selection plays all the players), 1 (one selection for each player), and 4 (one population per player).

Members per population ($\mathcal{N}_\phi$) One of the parameters that can have a significant effect on the process's effectiveness is the number of members within a population. Koza uses a population size of 500 about $\frac{2}{3}$ of the time [5, p. 98]. This parameter is also one of the primary determinants of the process's efficiency. The values for this parameter that we are going to investigate are 50, 200, 500, and 2000.

Games per member per generation (M) If an agent in a role is playing in a value chain whose other players are being fulfilled by different agents, then we need to define a minimum number of games per generation that need to be played by each member. For comparability across different population sizes, possible values for this parameter are specified in terms of the percentage of other members that the member will play in each generation. The values that we are going to investigate are 5%, 10%, and 20%. This only applies to the first two "number of populations" choices above.

Turns per game (W) The number of turns per game has at least two important effects on the scores that players end up earning in the game. First, longer games help minimize the effects of the player's initial inventory position, focusing the scoring more tightly on the player's re-stocking strategy. Second, longer games allow for a more complete realization of the negative effects of a bad re-stocking strategy; if it has not become obvious by 35 weeks, then it almost certainly would become obvious by 100 weeks. The possible values that we are going to investigate are 35 and 100. (We chose 35 because that is the length of the MIT Beer Game, and we chose 100 because it seems suitably large.) In both cases we will not use 35 (or 100) *per se*; we will use a random number distributed around 35 (or 100) in order to keep the agents from over-adapting to the number of weeks.

Priming of initial population Priming of an initial population means putting relatively higher fitness agents into the population [5, p. 94]. Koza proposes, and we agree, that it makes sense either to prime the whole population with relatively equally capable agents or to not prime at all; thus, possible values for this parameter are either "yes" or "no." This brings up the obvious next question of how to perform this priming. We hypothesize that a useful approach would be to start with a high population (maybe 2000), run the simulation for one generation, choose the best 500, and then run the evolutionary process as normal with this smaller population. However, this is only a hypothesis. A full investigation of this parameter will have to wait until we understand more about this problem.

We are not going to be looking at determining the "best" value for each of these parameters. Here we simply want to gain insight into 1) the effect that these factors have on the effectiveness and efficiency of the search, and 2) the dependencies among these factors. We will use insight gathered from this process in the investigations outlined in the next section.

3.4 Initial investigation into information value

The parameters discussed in this section form the heart of our investigation. We looked into the parmeters discussed in the previous sections because we wanted to ensure that the investigations into the parameters in this section were performed as efficiently and effectively as possible. Further, the parameters in this section can take many values and have many interdependencies so this will be a complex and time-consuming series of experiments.

Items in the function and terminal sets We have currently defined 21 functions: mathematical (7), logical (8), and informational (6). We can add the functions in each of these three sets to the function set either as a group or individually in order to see what effect they have. We have a smaller set of terminals — four plus the set of integers. Certainly one question relates to how small we can make the set of terminals and still enable the evolutionary process to proceed.

These are some further questions that we have: Which functions and terminals are used by higher-performing agents? We can investigate this question from two different perspectives. 1) One way is to manipulate the demand distribution and other parameters and observing which functions and terminals end up in the higher-performing agents. Do certain of the components appear in higher-performing agents? 2) Another way is to hold the functions and terminals fixed in three of the populations and manipulate the function and terminal sets. Do certain components lead to absolute dominance in a population? Another question relates to investigating whether or not certain fitness measures require that successful agents have certain terminals or functions.

In short, the questions in this section relate to examining questions related to valuing information.

Type of demand function This is one of the primary ways in which we can manipulate the difficulty of a scenario. Depending on how successful the genetic program is on simpler problems for which an optimal solution is already known, we could run the evolutionary process with one of many values: standard MIT demand function, uniform distribution, Poisson distribution, cyclical variations, and non-stationary variations. Other parameters which can be used to make the problem more difficult: shipment lead time (determinstic or stochastic), information lead time (0, 1, 2). If the genetic program is successful at finding solutions for difficult problems, then we will compare the findings of this process with the findings of management science and information systems.

Inventory holding costs and penalty costs We will manipulate the values of these two parameters to determine if the genetic program is sophisticated enough to respond to this type of control. That is, will it be able to find low-inventory-level solutions when the first is relatively higher and high-inventory-level costs when the second is relatively higher.

Fitness measure As we discussed in §2.2, several different fitness measures are available for this problem. Three (based on A, Υ, and ω) are appropriate for encouraging an agent to focus on its own costs; two others (based on T and Θ) encourage an agent to focus on the costs of the whole value chain. We want to investigate how much the different incentives actually affect the performance of the agents in the game in different scenarios. In order to address this problem, we will have to determine how to assign credit to members of a value chain that perform well. We have discussed using T and Θ, and it might be as simple as dividing the cost by the number of participants in the supply chain — but it might not.

Maximum depth overall The most effective way of controlling the complexity of the genetic programming problem is to limit the maximum depth of the tree representing the agent's re-stocking strategy. This places a limit on the number of different functions that the genetic program has to explore. We will manipulate this parameter as a means of exploring how agents can handle information overload. The possible values that we will look at are 6, 12, 17 (Koza's standard value), and 25.

Minimum depth of initial population This parameter provides a way of making the genetic programming process easier or harder. If the solution to one of the scenarios is simple, then removing all the simple structures from the initial population (by setting this parameter to 5) would make the genetic program have to discover the structure by itself, rather than having it be created in the initial population. Possible values that we will look at are 1 (Koza's standard value), 2, and 5.

3.5 Further investigations into valuing and pricing information

While the above certainly describes an ambitious research agenda, it by no means exhausts the possibilities. A much more complex supply chain scenario is one in which multiple players can play each level of the supply chain, and a player must decide among them when placing an order. This would add a significant amount of difficulty to the genetic program.

To this point we have considered information and computational capabilities to be costless. For example, when a player uses the `sum` or `inv` functions, it cost him nothing. We propose that some additional cost should be placed on the agent for every additional piece of information that it uses in the calculation of its re-stocking strategy. Thus, the value of the information will be reflected not only in a positive sense — if the information is valuable (useful), then the re-stocking strategy will be more effective and the agent's possibility of reproducing into the next generation will go up — but in a negative sense as well — if the information is not valuable but is being used anyway, then the agent will have to bear the burden of the cost of collecting and using that information. We will change the raw fitness function for this set of investigations to reflect the cost of using information, and then we will see how this changes the genetic program's process.

The most interesting possibility lies in adding another type of competitor to this scenario. Above we discussed adding information cost to the player's fitness function. Here we propose that a population of information sellers be added to the scenario as a competitor for all the players in all of the roles. The fitness function of the information sellers will reflect its ability to sell information for as much as it can. The result of this experiment should be better insight into which information is most valuable to the agents in determining their re-stocking strategies.

4 Summary

This paper describes an ambitious research program that will use genetic programming to investigate re-stocking strategies in a multi-level supply chain. After describing the basics of the evolutionary process (§2), we describe our overall plan of investigation (§3). In this investigation we will be looking at questions related to the value of information, the usefulness of the genetic programming approach to this problem, and the supply chain management problem itself. We describe the design of an early experiment (§3.2) that we will perform to help determine some settings for the evolutionary scenario. We next discuss (§3.3) some further investigations into parameters of the process that will have a large effect on the efficiency and effectiveness of the process. We follow this up (§3.4) with a discussion of our proposed investigations into the value of information and computational capabilities, into the ability of the genetic program to respond to changes in fitness measures, and into the ability of the genetic program to handle larger search spaces. We finish the explanation of our research project (§3.5) with a discussion of alternative approaches for investigating the valuing and pricing information, the most exciting of which involves sellers of information co-evolving with the agents evolving in the supply chain game.

We have great hopes for what we can learn about valuing information, genetic programming, and supply chain management. Many researchers have addressed these problems separately, but we think much is to be gained by looking at them together. All that remains is to find out what that is.

A Terminals and functions

A.1 Terminals

The following are all the terminals we used in our experiments:

1. `j` $\in$ set of integers
2. `I`: the amount of the good that the agent has in stock
3. `O`: the amount of the good that the agent has on order
4. `D`: the current demand for this agent
5. `C`: the current week number of the game

A.2 Functions

The following are the functions that can be used in the construction of an agent's strategy:

A.2.1 Mathematical

1. `(sum X Y)`: returns $X + Y$
2. `(- X Y)`: returns $X - Y$
3. `(% X Y)`: returns `round(X Y)`, the result of rounding $\frac{X}{Y}$ to the nearest integer; if $Y = 0$, returns 100000.
4. `(mult X Y)`: returns $X * Y$
5. `(min X Y)`: returns the minimum of `X` and `Y`
6. `(max X Y)`: returns the maximum of `X` and `Y`
7. `(sin X)`: returns $\sin X$, where `X` is in degrees

A.2.2 Logical

1. `(ifThen X Y)`: if `X` does not evaluate to `0`, returns value of `Y`, otherwise returns `0`
2. `(ifThenElse X Y Z)`: if `X` does not evaluate to `0`, returns value of `Y`, otherwise returns `Z`
3. `(not X)`: if `X` does not evaluate to `0`, returns `0`, otherwise returns `1`
4. `(and X Y)`: if `X` and `Y` are true, returns `1`, otherwise returns `0`
5. `(or X Y)`: if `X` or `Y` are true, returns `1`, otherwise returns `0`
6. `(gt X Y)`: if $X > Y$, returns `1`, otherwise returns `0`
7. `(lt X Y)`: if $X < Y$, returns `1`, otherwise returns `0`
8. `(equal X Y)`: if $X = Y$, returns `1`, otherwise returns `0`

A.2.3 Informational

1. `(MYDEM Y)`: the player's own demand `Y` weeks ago. The function uses the modulo function on `Y`; the demand is actually calculated for `mod(Y, W)` weeks ago (where `W` is the number of weeks in a game).
2. `(MYINV Y)`: the player's own inventory `Y` weeks ago. The function uses the modulo function on `Y`; the inventory is actually retrieved for `mod(Y, W)` weeks ago (where `W` is the number of weeks in a game).
3. `(MYORD Y)`: the player's order that it placed `Y` weeks ago. The function uses the modulo function on `Y`; the order is actually retrieved for `mod(Y, W)` weeks ago (where `W` is the number of weeks in a game).
4. `(DEM X Y)`: player `X`'s demand `Y` weeks ago. The function uses the modulo function on `X` to map from integers to game players. For example, if `mod(X, 4)=3`, then the *retailer* would be chosen. The function also uses the modulo function on `Y`; the demand is actually retrieved for `mod(Y, W)` weeks ago (where `W` is the number of weeks in a game).
5. `(INV X Y)`: player `X`'s inventory `Y` weeks ago. The function uses the modulo function on `X` to map from integers to game players. For example, if `mod(X, 4)=3`, then the *retailer* would be chosen. The

function also uses the modulo function on `Y`; the inventory is actually retrieved for `mod(Y, W)` weeks ago (where `W` is the number of weeks in a game).

6. `(ORD X Y)`: player `X`'s order that it placed `Y` weeks ago. The function uses the modulo function on `X` to map from integers to game players. For example, if `mod(X, 4)=3`, then the *retailer* would be chosen. The function also uses the modulo function on `Y`; the order is actually retrieved for `mod(Y, W)` weeks ago (where `W` is the number of weeks in a game).

Endnote

The inspiration for this paper comes from reading Kimbrough, Wu, and Zhong's paper from FMEC2000 as well as interesting conversations held after the paper was presented. We have also benefitted from several long discussions concerning GAs and GPs with Michael Gordon.

References

[1] Chen, F. Decentralized supply chains subject to information delays. *Management Science 45*, 8 (August 1999). 1076–1090.

[2] Free Software Foundation, Inc. GNU CLISP — an ANSI Common Lisp. http://clisp.cons.org/.

[3] Holland, J. H. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence.* MIT Press, 1992.

[4] Kimbrough, S. O., Wu, D., and Zhong, F. Computers play the beer game: can artificial agents manage supply chains? *Decision Support Systems 33* (2002). 323–33.

[5] Koza, J. R. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* A Bradford Book/The MIT Press, 1992 (7th printing, 2000). ISBN 0-262-11170-5.

[6] Moore, S. A., and Demaagd, K. Beer game genetic program. http://sourceforge.net/projects/beergame/.

[7] Sterman, J. Modeling managerial behavior misperceptions of feedbach in a dynamic decision making experiment. *Management Science 35*, 3 (1989). 321–339.

[8] Sterman, J. D. Teaching takes off: Flight simulators for management education. http://web.mit.edu/jsterman/www/SDG/beergame.html.