Mitigating the Bullwhip Effect in Supply Chains using Grammatical Evolution

Michael Phelan
University College Dublin
Centre for Telecommunications Value-Chain-Driven Research[1],
Room D102 (A), Building D,
Michael Smurfit School of Business,
University College Dublin,
Carysfort Avenue, Blackrock,
Co. Dublin, Ireland
Tel: + 353 1 716 4355
michael.phelan@ucd.ie

Seán McGarraghy
University College Dublin
Management Information Systems
Quinn School of Business
University College Dublin
Belfield,
Dublin 4, Ireland.

Tel: +353 1 716 4734
sean.mcgarraghy@ucd.ie

Abstract

This research introduces a relatively new evolutionary algorithm in computer science; Grammatical Evolution (GE), to the field of supply chain dynamics and bullwhip mitigation. As a proof of concept several experiments are conducted to derive optimal ordering policies for agents in a multi-tier supply chain. These results are compared with existing research using Genetic Algorithms (GA) to derive optimal ordering policies using similar simulations. This paper shows that GE can consistently discover the optimal ordering policies similar to the GA approach, and that in several experiments GE outperforms GA.

Keywords: Bullwhip Effect, Grammatical Evolution, Supply Chain, Ordering Policy

Introduction

The objective of supply chain management is to provide a rapid flow of high quality, relevant information that will enable suppliers provide a continuous, timely flow of materials to customers. However, unplanned demand oscillations; including those caused by stockouts in the supply chain execution process, create distortions which can propagate up and down the supply chain. There are numerous causes, often co-existing, that will cause these supply chain distortions to initiate what has become known as the Bullwhip Effect (Lee, Padmanabhan, and Whang 1997, 1997).

This unplanned-for demand can result in a “lumpy” ordering pattern, which may be a minor variance for any one customer, but when it oscillates back through the supply chain can result in large and costly fluctuations at the supplier end. Often, these demand oscillations will lead to a period of over-production in manufacturing with the need to acquire and expedite more raw materials and increase production.

[1] This research was made possible by Science Foundation Ireland (SFI) funding for the Centre for Telecommunications Value-Chain-Driven Research (CTVR) under Grant No. 03/CE3/I405.

While the bullwhip effect has been historically seen as an inevitable aspect of complex supply chains; it can cause vast sums of money (Cooke 1993) to be caught up in the supply chain, leading to dramatic inefficiencies affecting all tiers of the chain. Problems such as excess inventories, quality problems, higher raw material costs, overtime expenses and shipping costs are common. In the worst-case scenario, customer service declines, lead-times increase, sales are lost, costs go up and capacity is adjusted. An important element to operating a smooth-flowing supply chain is to mitigate, and preferably eliminate, the bullwhip effect.

Fig 1. Increasing variability of orders up the supply chain: the bullwhip effect (Lee, Padmanabhan, and Whang 1997, 1997)

This research introduces a relatively new evolutionary algorithm in computer science to the field of supply chain dynamics and bullwhip mitigation. Grammatical Evolution (GE) (O'Neill and Ryan 2001, 2003), (Brabazon and O'Neill 2006) is an evolutionary algorithm that can evolve complete programs (ordering policies) in an arbitrary language using a variable-length binary string. The binary string (genome) determines which production rules in a Backus Naur Form (BNF) grammar definition are used in a genotype-to-phenotype mapping process to generate a program.

A multi-echelon supply chain (Sterman 1989), (Mosekilde and Larsen 1988) is modelled using artificial agents to generate optimal ordering policies that reduce the bullwhip effect in the supply chain. Policies are determined using GE with a Genetic Algorithm (GA) search engine. Conceivably any search engine that can operate on binary or integer strings could employ GE's mapping process to generate a program or policy.

Initial results compare favourably with existing research involving ordering policies generated using a GA (Kimbrough, Wu, and Zhong 2001, 2002), (O'Donnell et al. 2006), finding the same policies as the GA in all experiments, and in some cases GE outperforms GA. Current research is examining more complex grammar structures (incorporating agent memory) leading to more efficient ordering policies.

Literature Review

The four main causes of the bullwhip effect have been identified by Lee et al. (1997, 1997) are; Demand Forecast Updating, Order Batching, Rationing and Shortage Gaming, and Price Variations. The authors provide an insight into the effect each of these causes have on a supply chain and possible methods to mitigate the bullwhip effect once these sources of demand variation are identified.

Due to the complexity of a multi-tier supply chain researchers and practitioners have used the MIT Beer Distribution Game (Sterman 1989), (Mosekilde and Larsen 1988) to simulate a supply chain with four tiers: Retailer, Wholesaler, Distributor and Factory. The Beer Game is used in industry to demonstrate the existence of the bullwhip effect in a simplified supply chain but it is also complex enough (being an intractable 23^{rd} order non-linear difference equation (Sterman 1989)) to be used for simulation purposes and employed for research into bullwhip mitigation techniques.

Sterman (1989) studied the effects of human decision making while playing the Beer Game, and suggests reasons for this behaviour. He provides a stock management heuristic to reduce irrationality in ordering patterns employing the widely used forecasting technique; Simple Exponential Smoothing (SES) (Gardner 1985, 1985), which can also cause the bullwhip effect (Chen, Ryan, and Simchi-Levi 2000). Kimbrough et al. (2001, 2002) suggest that this irrationality while playing the Beer Game may be due to lack of incentives for information sharing, bounded rationality, or the consequence of individually rational behaviour that works against the interests of the group. Chen (1999) showed that the bullwhip effect can be reduced using a base stock installation policy, under the assumption that all players in the supply chain work as a team. Base-stock is optimal when facing stochastic demand; but a “one for one” (1 – 1) policy is optimal when facing deterministic demand (Kimbrough, Wu, and Zhong 2001, 2002). This has lead to research in the area of artificial agents replacing humans in the Beer Game. Lee et al. (1997) also identified the main cause of the bullwhip effect as the agents’ overreaction on their immediate orders; using GA and GE, this overreaction is eliminated, resulting in reduced costs throughout the supply chain, and more efficient ordering and stocking policies.

In recent years evolutionary computational techniques have been employed to investigate optimal ordering policies in a simulated multi-tier supply chain. These include Genetic Algorithms (GA) (Kimbrough, Wu, and Zhong 2001, 2002), (O'Donnell et al. 2006), (Chan, Cheng, and Huang 2006), (Strozzi, Bosch, and Zaldivar 2007) and Genetic Programming (GP) (Moore and DeMaagd 2004). This paper adds another evolutionary computational technique: that of Grammatical Evolution (O'Neill and Ryan 2001, 2003), (Brabazon and O'Neill 2006) and compares experimental results with results obtained by Kimbrough et al. (2001, 2002) and O’Donnell et al. (2006).

Methodologies & Implementation

This section describes the application of Grammatical Evolution (GE) to discover optimal ordering policies for artificial agents playing the Beer Game. The simulations used to generate the agents ordering policies and fitness functions are similar to that of

Kimbrough et al. (2001, 2002) and O'Donnell et al. (2006), with the key difference being the evolutionary algorithm employed to generate the ordering policies. The evolutionary algorithm approach proposed by Kimbrough et al. (2001, 2002) used a fixed-length binary string and a GA to determine the optimal ordering policies in several experiments devised to mitigate the bullwhip effect. The GE method described in this section uses a grammar definition representing various ordering policy rules, a fitness function similar to Kimbrough et al. (2001, 2002) and O'Donnell et al. (2006), and a GA search engine. Results in the next section show how, depending on a chosen grammar selection, the GE can discover more complex ordering policies that reduce the agents cost over the Beer Game time horizon.

The basic setup and weekly operation of the Beer Game (Sterman 1989) is as follows. Shipments arrive from upstream players, orders arrive from downstream players, orders are filled and shipped where possible, affecting the inventory and backorders of a player, the player decides how much to order to replenish their inventory, and finally inventory holding costs and backorder costs are calculated for each player every week.

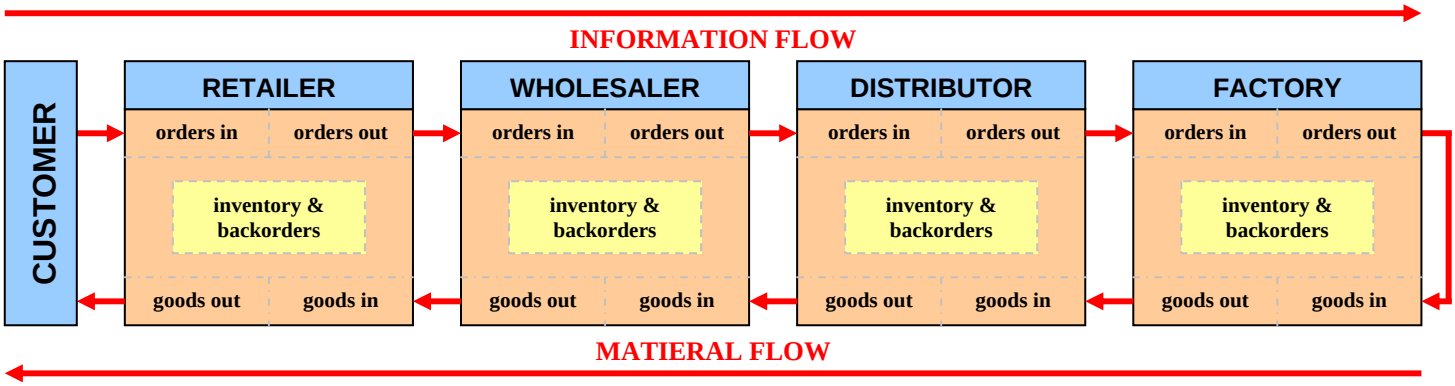

Fig 2. The Beer Game

Kimbrough et al. (2001, 2002) and O'Donnell et al. (2006) employed GAs to determine optimal ordering policies for the Beer Game. Each of the human players Retailer, Wholesaler, Distributor and Factory are replaced with artificial agents playing the Beer Game to see if artificial agents can learn and discover good and efficient ordering policies in supply chains (Kimbrough, Wu, and Zhong 2001, 2002). Each agent's rule is represented with a 6-bit binary string. The leftmost bit represents the sign ("+" or "-") and the next five bits represent (in base 2) how much to order. For example, rule 101001 can be interpreted as "x+9": That is, if demand is x then order x+9. In the case of negative demand each agent ordering policy is determined by: $Max(0, x+y)$ each week, where x is the agents incoming demand for that week, and y is the amount of beer to order as determined by the GA.

The cost function used by GE to determine the fitness of an individual is similar to that of Kimbrough et al. (2001, 2002) and O'Donnell et al. (2006) where the absolute fitness function is the negative of the total cost. The total cost of the Beer Game is calculated by:

$$\sum_{A=1}^{4}\sum_{T=1}^{N}\left(Inventory_{AT}\times InventoryCost\right)+\left(Backorders_{AT}\times BackorderCost\right)$$

A	Agents (Retailer, Wholesaler, Distributor and Factory)
T	Current week being played
N	Total number of weeks (35, 100)
$Inventory_{AT}$	The amount of inventory on hand for agent *A* in week *T*
InventoryCost	The cost of holding 1 case of inventory per week is €1
$Backorder_{AT}$	The amount of backorders for agent *A* in week *T*
BackorderCos	The backorder cost for 1 case of inventory per week is €2

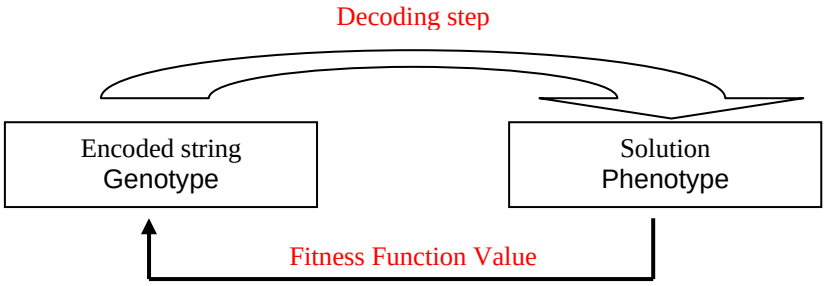

Fig 3. Decoding of genotype to phenotype into a solution in order to calculate fitness

The best ordering policies (phenotype) after playing the game for a number of weeks (35 or 100) are retained for evolving better policies in future generations. A new technique for evolving optimal ordering policies – GE is now described.

Evolutionary algorithms have been successfully used for the automatic generation of programs. Genetic programming (GP), particularly the version popularised by Koza (1992) using Lisp as the target language, has been very successful in this area. GP is fundamentally different to GA in to two ways. Firstly a GA evolves binary strings which are a direct encoding of a potential solution whereas in GP the solution itself is evolving where the solutions could be a computer program or an agents ordering policy. Secondly in GA the length of the binary string is fixed but in GP the length is variable recognising that the length of a solution program or policy may not be known a priori and as such the number of genes must also be open to evolution. In GP solutions (programs, ordering policies etc…) are represented as syntax trees (Fig. 4). It is to these trees that the evolutionary search operators of crossover and mutation are applied.

(+ (cos x) (* x 3.14) (/ y x)

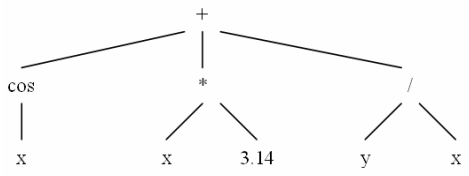

Fig 4. Example S-expression (left) and corresponding syntax tree (right). The syntax tree decodes into the expression (cos *x*) + (3.14*x*) + (*y*/*x*) where x and y are predefined constants.

Moore and DeMaagd (2004) use GP to investigate heuristics for supply chain recording policies and examine the effects of various parameter settings used in their GP model.

Evolutionary algorithms have also been used to generate other languages using a grammar definition of the target language. Grammatical Evolution is an evolutionary algorithm developed by Ryan and O'Neill (2001, 2003) that can evolve computer

programs, sentences in any language or for the purpose of this research: Beer Game ordering policies. Unlike GP where solutions are represented as syntax trees, in GE a linear genome representation is used in conjunction with a grammar. Similar to GP, each individual or genotype is a variable length binary string that contains codons (groups of 8 bits) used to select production rules from the grammar (O'Neill and Ryan 2001, 2003), (Brabazon and O'Neill 2006).

The language or in this case ordering policies to be generated, are described using a Backus Naur Form (BNF) grammar definition, consisting of a series of production rules mapping non-terminal symbols to the terminals which appear in the language. BNF is a notation for expressing the grammar of a language, or in this example, ordering policies in the form of production rules. BNF grammars consist of *terminals*, which are items that can exist in the language, e.g. +, -, *, / etc… and *non-terminals*, which can be expanded into one or more terminals or non-terminals.

The GE system is inspired by the biological process of generating a protein from the genetic material of an organism. Proteins are responsible for traits such as eye colour or height and are fundamental to the development and operation of all living organisms. For a more detailed discussion of the biological analogy the reader is referred to O'Neill and Ryan (2001, 2003) and Brabazon and O'Neill (2006). Figure 5 compares the mapping process employed by both GE and biological organisms. The binary string of GE is analogous to the double helix of DNA, each guiding the formation of the phenotype. In GE this occurs with the application of production rules from a BNF grammar to generate the terminals of the program or ordering policy. In biological terms the phenotypic protein is determined by the order and type of protein subcomponents (amino acids) that are joined together.

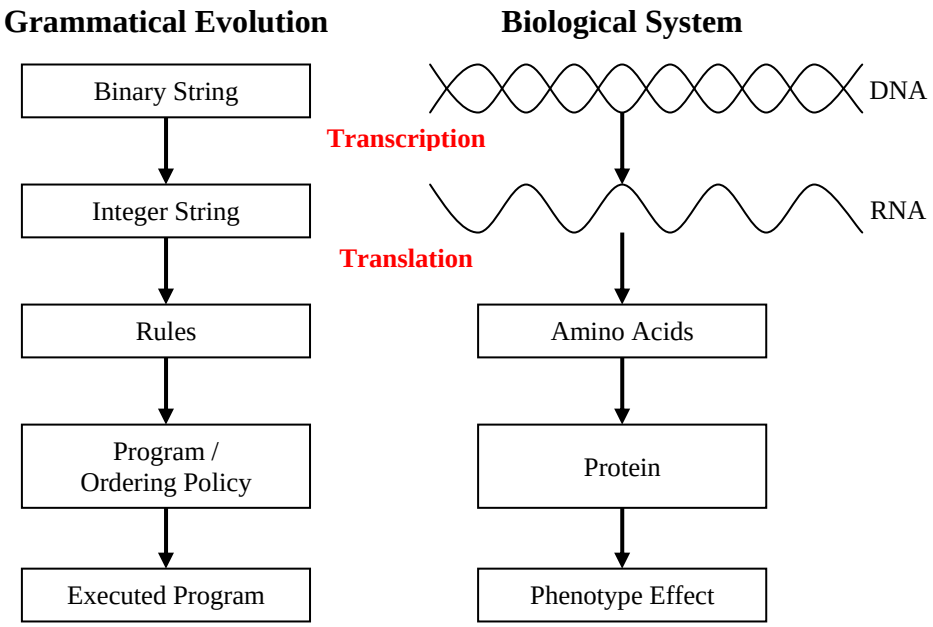

Fig 5. Comparison of GE system and biological genetic system

When embarking upon a problem in GE, a suitable BNF grammar definition must first be defined (Brabazon and O'Neill 2006). The BNF can either be the specification of an entire language or a subset of the language geared towards the problem. In this case, a BNF grammar is defined, representing ordering policy choices that an agent can have. A grammar can be represented by the tuple {*N*, *T*, *P*, *S*}, where *N* is the set of non-terminals, *T* is the set of terminals, *P* is a set of production rules and *S* is a start symbol which is a member of *N*. When there are multiple productions that can be applied to an element of *N*, the choices are delimited with the '|' symbol. Below is an example of a BNF grammar used to determine the optimal ordering policies for the artificial agents.

N = {<agent>, <policy>, <op>, <int>, <var>}
T = {x, +, -, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20}
S = {<agent>}

And *P* can be represented as:

```
<agent>  ::= <policy><policy><policy><policy>
<policy> ::= <var>|<var><op><int>|<policy><op>(<var><op><int>)
<op>     ::= +|-
<int>    ::= 0|1|2|3|4|5|6|7|8|9|10|11|12|13|14|15|16|17|18|19|20
<var>    ::= x
```

Which can be rewritten as:

```
(A)   <agent>  ::=   <policy><policy><policy><policy>        (0)
(B)   <policy> ::=   <var>                                   (0)
                   | <var><op><int>                          (1)
                   | <policy><op>(<var><op><int>)            (2)
(C)   <op>     ::=   +                                       (0)
                   | -                                       (1)
(D)   <int>    ::=   0                                       (0)
                   | 1                                       (1)
                   | 2                                       (2)
                     :                                        :
                     :                                        :
                   | 19                                      (19)
                   | 20                                      (20)
(E)   <var>    ::=   x                                       (0)
```

Table 1 shows the productions rules and the choices available for each rule.

Rule	Number of Choices
A	1
B	3
C	2
D	21
E	1

Table 1. Summary of production rules and number of choices available for each rule. E.g. production rule B has 3 possible choices (1) <var>, (2) <var><op><int> and (3) <policy><op>(<var><op><int>).

The genotype is used to map the start symbol onto terminals by reading codons of 8 bits to generate a corresponding integer value, from which an appropriate production rule is selected by the following mapping function:

$$Rule = c \bmod r$$

where c is the codon integer value, and r is the number of rule choices for the current non-terminal symbol.

During the genotype to phenotype mapping process it is possible for individuals to run out of codons, and in this case the *wrap* operator is applied which reuses the codon string, starting with the left-most codon to determine the next rule to be selected. In GE, each time the same codon is expressed it will always generate the same integer value, but depending on the current non-terminal to which it is applied, it may result in the selection of a different production rule. Each time that a particular individual is mapped from its genotype to its phenotype the same output is generated, as the same choices are made each time. It is possible that an incomplete mapping can occur even after several wrappings, due to the integer codon values returning the same production rules each time. This is similar to an infinite loop in programming and requires an upper bound to be placed on the number of wrappings that can occur (e.g. *MAX_WRAP* = 10).

Take for example an individual with three codons and of which specify rule (0) from the following grammar:

```
(A)   <policy> ::=   (<policy>)<op>(<policy>)          (0)
                   | <var><op><int>                    (1)
                   | <var>                             (2)
```

Even after wrapping the mapping process would be incomplete and would carry on infinitely unless *MAX_WRAP* is used to terminate the process after 10 wrappings. This occurs because the non-terminal `<policy>` is mapped recursively by production rule (0), e.g. `(<policy>)<op>(<policy>)` becomes
`((<policy>)<op>(<policy>))<op>(<policy>)` which becomes
`(((<policy>)<op>(<policy>))<op>(<policy>))<op>(<policy>)` which would continue infinitely unless a maximum limit is placed on the number of wrappings. In this case the genotype to phenotype mapping process is incomplete and the individual is termed invalid and not included in the population.

Mapping process example

Consider the following randomly generated binary string (genotype) after mutation and crossover have taken place:

11111000 11110111 **00100111** 00101111 **10110101** 10000101 **00010010** (*DNA*)

Transcribing the binary string into an integer codon string where each codon is 8 bits:

248 247 **39** 47 **181** 133 **18** (*RNA*)

Using the BNF grammar outlined above an ordering policy will be generated.

As there is only one production rule for the start symbol <agent>, it is automatically replaced and no codons are used.

The ordering policy becomes:

<policy><policy><policy><policy>

Taking the left most non-terminal <policy> (representing the ordering policy for the Retailer agent) there are 3 possible replacements:

<var> | <var><op><int> | <policy><op>(<var><op><int>)

The codon being read is 248 then applying the mapping function:

248 *mod* 3 = 2 (selects 2 from rule B) → <policy><op>(<var><op><int>)

The ordering policy becomes:

<policy><op>(<var><op><int>)<policy><policy><policy>

Again, reading the left most non-terminal <policy> and reading the next codon to the right then

247 *mod* 3 = 1 (selects 1 from rule B) → <var><op><int>

The ordering policy becomes:

<var><op><int><op>(<var><op><int>)<policy><policy><policy>

The next left most non-terminal <var> has only one possible choice *x* giving:

x<op><int><op>(<var><op><int>)<policy><policy><policy>

The next left most non-terminal <op> has 2 possible choices "+", "-" and selecting the next available codon value

39 *mod* 2 = 1 (selects 1 from rule C) → "-"

The ordering policy becomes:

x-<int><op>(<var><op><int>)<policy><policy><policy>

The next left most non-terminal <int> has 21 possible choices 0 to 20 and selecting the next available codon value

47 mod 21 = 5 (selects 5 from rule D) → 5

The ordering policy becomes:

x-5<op>(<var><op><int>)<policy><policy><policy>

The next left most non-terminal <op> has 2 possible choices "+", "-" and selecting the next available codon value

181 *mod* 2 = 1 (selects 1 from rule C) → "-"

The ordering policy becomes:

x-5-(<var><op><int>)<policy><policy><policy>

The next left most non-terminal <var> has only one possible choice *x* giving:

x-5-(x<op><int>)<policy><policy><policy>

The next left most non-terminal <op> has 2 possible choices "+", "-" and selecting the next available codon value

133 *mod* 2 = 1 (selects 1 from rule C) → "-"

The ordering policy becomes:

x-5-(x-<int>)<policy><policy><policy>

The next left most non-terminal <int> has 21 possible choices 0 to 20 and selecting the next available codon value

18 mod 21 = 18 (selects 18 from rule D) → 18

The ordering policy becomes:

x-5-(x-18)<policy><policy><policy>

At this point the mapping process has generating an ordering policy (partial phenotype) for the Retailer agent, and all other agents are generated in a similar fashion resulting in the complete phenotype (ordering policies for all agents). In the generation of the Retailer policy all of the codon values were used. The *wrap* operator is applied and the next codon used is the left-most codon (initial codon value) of 248.

Agent	**Ordering Policy**
Retailer	x-5-(x-18)
Wholesaler	x-5-(x-18)
Distributor	x-5-(x-18)
Factor	x-5-(x-18)

Table 2. Ordering policy (phenotype) generated from genotype in mapping example recommends that 13 units are ordered each week by each of the agents.
This is only a sample policy and not the best policy discovered by the GE.

As mentioned; any search engine that can operate on binary or integer strings could employ GE's mapping process to generate a program or policy. For example; particle swarm and differential evolution algorithms have been used to create grammatical swarm and grammatical differential evolution algorithms respectively (Brabazon and O'Neill 2006). The GE algorithm described in this paper employed a variable length GA as the search engine.

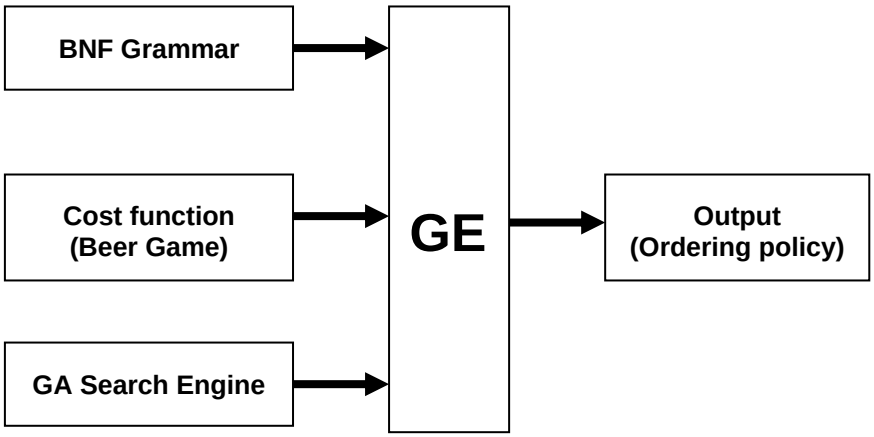

Fig 6. Overview of the modular structure of Grammatical Evolution as applied in this paper

Results

This section of the paper presents the results of the experiments carried out using GE to evolve optimal ordering policies for agents playing the beer game. As a proof of concept two experiments with varying parameters were carried out and the results compared with results described by Kimbrough et al. (2001, 2002) and O'Donnell et al. (2006). For each experiment the GE was run 50 times to test the robustness of the solutions.

The parameters used for all experiments are the parameters used by Kimbrough et al. (2001, 2002) and O'Donnell et al. (2006) unless otherwise stated. A summary of the parameters for the Beer Game and the evolutionary algorithms is given in Table 3.

Beer Game Parameters	
Number of weeks played	35 / 100
Inventory holding cost	€1 per case per week
Backorder cost	€2 per case per week
GE & GA Parameters	
Number of runs	50
Maximum generations	10
Population size	20
Crossover rate	0.87
Mutation rate	0.03
Pruning rate *	0.01
Duplication rate *	0.01
Generation gap *	0.9
Codon size (bits) *	8
Minimum number of codons *	1
Maximum number of codons *	10

Maximum number of wrappings *	10
* Additional parameters not used in GA experiments	

Table 3. Experimental parameters

Grammar Selection

The grammars used to generate the ordering policies are given below. The first grammar was designed to find the optimal "*x+y*" ordering policies as described by Kimbrough et al. (2001, 2002) and O'Donnell et al. (2006). The second grammar was designed to find (a) "1-1" policy, (b) "*x+y*" policy or (c) discover if the GE artificial agents could discover more complex ordering policies resulting in reduced costs.

BNF Grammar 1

```
<agent>     ::= <policy><policy><policy><policy>
<policy>    ::= <var><op><int>
<op>        ::= +|-
<int>       ::= 0|1|2|3|4|5|6|7|8|9|10|11|12|13|14|15|16|17|18|19|20
<var>       ::= x
```

BNF Grammar 2

```
<agent>     ::= <policy><policy><policy><policy>
<policy>    ::= <var>|<var><op><int>|<policy><op>(<var><op><int>)
<op>        ::= +|-
<int>       ::= 0|1|2|3|4|5|6|7|8|9|10|11|12|13|14|15|16|17|18|19|20
<var>       ::= x
```

Experiment 1 – deterministic demand

The first experiment uses the demand data for the classical MIT Beer Distribution Game where the customer orders 4 cases of beer for the first 4 weeks and 8 cases of beer for the remainder of the game and lead times are fixed at 2 weeks. The optimal ordering policy for all agents using GA is the "1-1" policy (Kimbrough, Wu, and Zhong 2001, 2002), resulting in a total cost over 35 or 100 weeks of €360 (O'Donnell et al. 2006).

BNF Grammar 1

After 50 runs GE using grammar 1 discovered the optimal ordering policy every time.

Min Cost	€360.00
Max Cost	€360.00
Avg Cost	€360.00

Table 4. Experiment 1, Grammar 1, Runs 50, Generation 10, Population 20, Weeks 100

BNF Grammar 2

After 50 runs GE using grammar 2 discovered the optimal ordering policy every time.

Min Cost	€360.00
Max Cost	€360.00
Avg Cost	€360.00

Table 5. Experiment 1, Grammar 2, Runs 50, Generation 10, Population 20, Weeks 100

Both grammars were tested on the deterministic demand and consistently discovered the optimal ordering policies over all runs with the same population size and number of generations as Kimbrough et al. (2001, 2002).

Experiment 2 – known stochastic demand

The second experiment uses known stochastic demand e.g., uniformly distributed between [0, 15] (Kimbrough, Wu, and Zhong 2001, 2002) and fixed lead times. For this experiment Kimbrough et al. (2001, 2002) sets the number of generations to 30 and the population size is increased to 1000. The optimal ordering policy using GA for all agents playing for 35 weeks (Kimbrough, Wu, and Zhong 2001, 2002) is the "*x*, *x*+1, *x*, *x*+1" policy resulting in a total cost of < €2000 (around €1926) as opposed to €2736 if the "1-1" policy is chosen. However, if the game is played over 100 weeks the "*x*, *x*+1, *x*, *x*+1" policy yields a total cost of €11220, which is worse than the "1-1" policy resulting in a total cost over 100 weeks of €8474 (O'Donnell et al. 2006). The agents are smart enough to adopt different policies over varying time horizons (Kimbrough, Wu, and Zhong 2001, 2002).

BNF Grammar 1 – 35 Weeks

After 50 runs GE discovered several ordering policies, finding a best policy with a cost of €1926 in 3 of the 50 runs and finding an average cost of €2176.52 over the 50 runs. The worst policy found in 1 of the 50 runs resulted in a cost of €3637.

Min Cost	€1926
Max Cost	€3637
Avg Cost	€2176.52

Table 6. Experiment 2, Grammar 1, Runs 50, Generation 10, Population 20, Weeks 35

Cost	**Frequency**
€1926	3
€2091	41
€2736	5
€3637	1

Table 7. Frequency of cost policies discovered with Generation 10, Population 20, Weeks 35

Increasing the population from 10 to 100 with the number of generations remaining at 10, GE discovered a best policy with a cost of €1926 in 17 of the 50 runs, and found an

average cost of €2026.72 over the 50 runs. As the population size increases, the quality of the policies improves, and the best policy is discovered more frequently.

Min Cost	€1926
Max Cost	€2091
Avg Cost	€2026.72

Table 8. Experiment 2, Grammar 1, Runs 50, Generation 10, Population 100, Weeks 35

Cost	**Frequency**
€1926	17
€2000	1
€2038	6
€2091	26

Table 9. Frequency of cost policies discovered with Generation 10, Population 100, Weeks 35

Further increasing the population from 100 to 1000, and increasing the number of generations from 10 to 30, GE found a policy with a cost of €1924 in 1 of the 50 runs, with the other 49 runs returning policies yielding a cost of €1926. Again, by increasing the population size and number of generations the optimality of the ordering policies improve.

Min Cost	€1924
Max Cost	€1926
Avg Cost	€1925.96

Table 10. Experiment 2, Grammar 1, Runs 50, Generation 30, Population 1000, Weeks 35

BNF Grammar 1 – 100 Weeks

After 50 runs GE discovered the optimal ordering policy in all but one run.

Min Cost	€8474
Max Cost	€51794
Avg Cost	€9340.40

Table 11. Experiment 2, Grammar 1, Runs 50, Generation 10, Population 20, Weeks 100

Increasing the population from 10 to 100 (10 times smaller than the GA in Kimbrough et al. (2001, 2002)) with the number of generations remaining at 10, after a further 50 runs GE discovered the optimal ordering policy every time.

Min Cost	€8474
Max Cost	€8474
Avg Cost	€8474

Table 12. Experiment 2, Grammar 1, Runs 50, Generation 10, Population 100, Weeks 100

BNF Grammar 2 – 35 Weeks

Grammar 2 was designed to test if the artificial agents could discover more complex ordering policies than the "*x*+*y*" policy. Again, the experiment was run over a 35 week and 100 week time horizon. After 50 runs GE found a best policy with associated cost of €532 by ordering 10 cases every week e.g. "*x* – (*x* - 10)", regardless of the incoming orders for each agent and worst case GE found the "1 – 1" ordering policy. The average cost found was €1708.28 over the 50 policies which is an improvement on the leading cost of €1926 found by grammar 1 over the same period.

Min Cost	€532
Max Cost	€2736
Avg Cost	€1708.28

Table 13. Experiment 2, Grammar 2, Runs 50, Generation 10, Population 20, Weeks 35

Increasing the population size to 100 and maintaining a generation size of 10, after 50 runs the worst case policy found is the leading policy found by grammar 1. The average policy cost has also been considerably reduced from €1708.28 to €1163.54.

Min Cost	€532
Max Cost	€1926
Avg Cost	€1163.54

Table 14. Experiment 2, Grammar 2, Runs 50, Generation 10, Population 100, Weeks 35

BNF Grammar 2 – 100 Weeks

After 50 runs GE discovered the optimal ordering policy or better every time. In 41 out of 50 runs GE discovered an ordering policy resulting in a total cost over 100 weeks of €8474 and in 9 (18%) runs found a more favourable policy than that of the GA approach as shown in Table 15. Taking the average cost over the 50 runs shows a 10% improvement on the GA results and an optimal cost of €1,717 using the ordering policy *x*-(*x*-9) for each agent where *x* is the incoming order received by the agent. This policy indicates that each agent should order 9 (the mean of the demand data (uniform [0, 15] over 100 weeks is 8.96) cases each week regardless of any incoming orders. Given that the distribution is Uniform (stationary data) it is optimal to use the mean of the data for the ordering policy (Makridakis, Wheelwright, and Hyndman 1998, 136).

Run	**Cost**
38	€1,717.00
26	€2,739.00
28	€2,739.00
45	€2,775.00
22	€2,851.00
27	€3,712.00
35	€3,712.00
42	€4,782.00
2	€4,996.00

Table 15. Optimal GE ordering policies, Runs 50, Generation 10, Population 20, Weeks 100

Min Cost	€1717.00
Max Cost	€8474.00
Avg Cost	€7549.14

Table 16. Experiment 2, Grammar 2, Runs 50, Generation 10, Population 20, Weeks 100

If the population is increased from 20 individuals to 100 individuals and the number of generations remains at 10 there is a significant improvement with 27 out of 50 (54%) runs finding policies with costs less than €8474. The optimal policy with cost of €1717 is now found in 3 runs and the average cost is lower.

Min Cost	€1717.00
Max Cost	€8474.00
Avg Cost	€5976.20

Table 17. Experiment 2, Grammar 2, Runs 50, Generation 10, Population 100, Weeks 100

Again, if the population is increased from 100 individuals to 1000 individuals, and the number of generations increase from 10 to 30 there is a dramatic improvement. Please note that only 30 runs were completed for this experiment due to the processing time required for a larger population with more generations. As can be seen from Table 18 there is a dramatic improvement in the number of runs (22 out of 30) finding the optimal policy and the remaining 8 runs found policies yielding a cost of €2739 which is also significantly lower than that of the GA policies.

Min Cost	€1717.00
Max Cost	€2739.00
Avg Cost	€1989.53

Table 18. Experiment 2, Grammar 2, Runs 30, Generation 30, Population 1000, Weeks 100

Summary & Future Work

This research introduces Grammatical Evolution, a relatively new evolutionary algorithm in computer science to the field of supply chain dynamics and bullwhip mitigation. The initial experimental results (deterministic demand and known stochastic demand with fixed lead times) compare favourably with existing research involving ordering policies generating using a GA (Kimbrough, Wu, and Zhong 2001, 2002), (O'Donnell et al. 2006).

In all experiments the GE approach using 2 grammars of differing complexities found similar optimal ordering policies as the GA and when using grammar 2 the GE consistently found more favourable policies for the artificial agents. This is possible because GE provides the policy rules (e.g. “building blocks”) and allows the agents the freedom to configure more complex ordering policies than the GA “$x+y$” policies. In several runs the GE found the optimal policy for all agents was to simply order the mean of the customer demand data every week regardless of incoming orders. This is an interesting result as it suggests that for demand data following a uniform distribution [0, 15] the optimal ordering strategy is to order the mean of the demand each week. Future work will examine if this holds for other distributions drawn from real data.

The policies and resulting lower costs attained by GE using the 2 grammars are encouraging and offer another evolutionary algorithm to the field of bullwhip mitigation in supply chain management.

Current research is developing more complex grammars that allow the agents track previous orders for their own tier and other agent tiers. Some basic forecasting models such as moving averages will also be provided for the agents and conceivably agents could construct their own forecasting models while searching for optimal ordering policies.

As mentioned the Beer Game is an excellent teaching aid and first step at simulating a supply chain. The next step is to introduce stochastic lead times into the model and compare results with those of Kimbrough et al. (2001, 2002) and O'Donnell et al. (2006). Future work will also examine a "real-world" supply chain with different demand distributions, lead times and several penalty metrics; it is not always possible to quantify the backorders cost in monetary terms so other metrics will be examined.

In relation to GE, other search engines will be employed and alternatives to the binary encoded genomes such as Gray codes will be investigated to improve the efficiency of the algorithm.

Acknowledgements

The authors would like to thank Dr. Anthony Brabazon for his insightful conversations into Grammatical Evolution and its possibilities and Tina O'Donnell for all her advice and assistance in Beer Game simulations.

References

Brabazon, A., and M. O'Neill. 2006. *Biologically Inspired Algorithms for Financial Modelling, Natural Computing Series*: Springer.

Chan, C. C. H., C. B. Cheng, and S. W. Huang. 2006. Formulating ordering policies in a supply chain by genetic algorithm. *Int. J. Model. Simul.* 26 (2):129-136.

Chen, F., J. K. Ryan, and D. Simchi-Levi. 2000. The impact of exponential smoothing forecasts on the bullwhip effect. *Naval Research Logistics* 47 (4):269-286.

Cooke, J. A. 1993. The $30 Billion Promise. *Traffic Management* 32:57-59.

Gardner, E. S. 1985. Exponential Smoothing - the State of the Art. *Journal of Forecasting* 4 (1):1-28.

———. 1985. Exponential Smoothing - the State of the Art - Response. *Journal of Forecasting* 4 (1):37-38.

Kimbrough, S. O., D. J. Wu, and F. Zhong. 2001. Computers play the beer game: can artificial agents manage supply chains? In *34th Hawaii International Conference on System Sciences*.

———. 2002. Computers play the beer game: can artificial agents manage supply chains? *Decision Support Systems* 33 (3):323-333.

Lee, H. L., V. Padmanabhan, and S. Whang. 1997. The bullwhip effect in supply chains. *Sloan Management Review* 38 (3):93-102.

———. 1997. Information distortion in a supply chain: The bullwhip effect. *Management Science* 43 (4):546-558.

Makridakis, S., S. Wheelwright, and R.J. Hyndman. 1998. *Forecasting: methods and applications*. 3rd ed: John Wiley and Sons: New York.

Moore, S. A., and K. DeMaagd. 2004. Using a Genetic Program to Search for Supply Chain Reordering. In *Genetic Programming Theory and Practice {II}*, edited by U. M. O'Reilly, T. Yu, R. L. Riolo and B. Worzel: Springer.

Mosekilde, E., and E. Larsen. 1988. Deterministic Chaos in the Beer Production-Distribution Model. *System Dynamics Review* 4 (1-2):131-147.

O'Donnell, T., L. Maguirez, R. McIvor, and P. Humphreys. 2006. Minimizing the bullwhip effect in a supply chain using genetic algorithms. *International Journal of Production Research* 44 (8):1523-1543.

O'Neill, M., and C. Ryan. 2001. Grammatical Evolution. *IEEE Transactions on Evolutionary Computation* 5 (4):349-358.

———. 2003. *Grammatical Evolution: Evolutionary Automatic Programming in a Arbitrary Language*. Vol. 4, *Genetic programming*: Kluwer Academic Publishers.

Sterman, J. D. 1989. Modeling Managerial Behavior - Misperceptions of Feedback in a Dynamic Decision-Making Experiment. *Management Science* 35 (3):321-339.

Strozzi, F., J. Bosch, and J. M. Zaldivar. 2007. Beer game order policy optimization under changing customer demand. *Decision Support Systems* 42 (4):2153-2163.