# Neuroevolutionary Inventory Control in Multi-Echelon Systems*

S. D. Prestwich[1], S. A. Tarim[2], R. Rossi[3], and B. Hnich[4]

[1]Cork Constraint Computation Centre, Ireland
[2]Operations Management Division, Nottingham University Business School, Nottingham, UK
[3]Logistics, Decision and Information Sciences Group, Wageningen UR, the Netherlands
[4]Faculty of Computer Science, Izmir University of Economics, Turkey
s.prestwich@cs.ucc.ie, armtar@yahoo.com.tr,
roberto.rossi@wur.nl, brahim.hnich@ieu.edu.tr

**Abstract.** Stochastic inventory control in multi-echelon systems poses hard problems in optimisation under uncertainty. Stochastic programming can solve small instances optimally, and approximately solve large instances via scenario reduction techniques, but it cannot handle arbitrary nonlinear constraints or other non-standard features. Simulation optimisation is an alternative approach that has recently been applied to such problems, using policies that require only a few decision variables to be determined. However, to find optimal or near-optimal solutions we must consider exponentially large scenario trees with a corresponding number of decision variables. We propose a neuroevolutionary approach: using an artificial neural network to approximate the scenario tree, and training the network by a simulation-based evolutionary algorithm. We show experimentally that this method can quickly find good plans.

## 1 Introduction

In the area of optimisation under uncertainty, one of the most mature fields is inventory control. This field has achieved excellent theoretical and practical results using techniques such as dynamic programming, but some problems are too large or complex to be solved by classical methods. Particularly hard are those involving *multi-echelon systems*, in which multiple stocking points form a supply chain. In such cases we may resort to simulation-based methods. Simulation alone can only evaluate a plan, but when combined with an optimisation algorithm it can be used to find near-optimal solutions (or plans). This approach is called *simulation optimisation* (SO) and has a growing literature in many fields including production scheduling, network design, financial planning, hospital administration, manufacturing design, waste management and distribution. It is a practical approach to optimisation under uncertainty that can handle problems containing features that make them difficult to model and solve by other methods: for example non-linear constraints and objective function, and demands that are correlated or have unusual probability distributions.

* B. Hnich is supported by the Scientific and Technological Research Council of Turkey (TUBITAK) under Grant No. SOBAG-108K027.

SO approaches to inventory control are typically based on policies known to be optimal in certain situations, involving a small number of reorder points and reorder quantities. For example in $(s, S)$ policies whenever a stock level falls below $s$ it is replenished up to $S$, while in $(R, S)$ policies the stock level is checked at times specified by $R$, and if it falls below $S$ then it is replenished up to $S$. SO can apply standard optimisation techniques such as genetic algorithms to these policies by assigning genes to reorder points and replenishment levels. In more complex situations involving constraints, multiple stocking points, etc, these policies may be suboptimal in terms of expected cost, though they can have other desirable properties such as improved planning stability. But a cost-optimal plan for a multi-stage problem with recourse must specify an order quantity in every possible scenario, so the plan must be represented via a *scenario tree*. The number of scenarios might be very large, or infinite in the case of continuous probability distributions, making the use of SO problematic. Scenario reduction techniques may be applied to approximate the scenario tree, but it might not always be possible to find a small representative set of scenarios.

An alternative form of approximation is to use an artificial neural network (ANN) to represent the policy. For example, the inputs to the ANN could be the current stock levels and time, and the outputs could be the recommended actions (whether or not to replenish and by how much). We must then train the ANN so that its recommendations correspond to a good plan. No training data is available for such a problem so the usual ANN backpropagation training algorithm cannot be applied. Instead we may use an evolutionary algorithm to train the network to minimise costs. This *neuroevolutionary* approach has been applied to control problems [8, 9, 21] and to playing strategies for games such as Backgammon [16] and Go [14], but it has not been extensively applied to inventory control. In this paper we apply neuroevolution to stochastic inventory control in multi-echelon systems. Section 2 presents our method, Section 3 evaluates the method experimentally, Section 4 surveys related work, and Section 5 concludes the paper.

## 2 A neuroevolutionary approach

To approximate the scenario tree, we construct a function whose input is a vector containing the time period and current inventory levels, and whose output is a vector of order quantities (which might be zero). We design the function automatically by simulation optimisation.

### 2.1 Scenario tree compression by neural network

An obvious choice for this function is an artificial neural network (ANN), which can approximate any function with arbitrary accuracy given a sufficient number of units. ANNs also come with a ready-made algorithm for optimisation: the well-known backpropagation algorithm. However, there is a problem with this approach: we do not have training data available (this also precludes the use of Support Vector Machines). To obtain training data we would have to solve a set of instances, and there is no known method for solving the harder instances to optimality. Instead we must use an ANN to

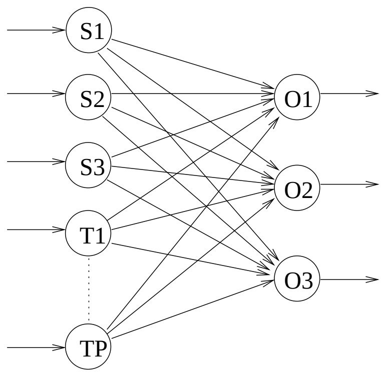

**Fig. 1.** The feedforward ANN used

solve a problem in *reinforcement learning*, in which we must choose its weights in order to maximise reward (in this case to minimise expected cost). Backpropagation cannot be used for this task, but we can instead use an evolutionary algorithm (EA) whose genes are the weights and whose fitness function is minus the cost. This neuroevolutionary approach has been applied to control problems and game learning.

In our experiments we began with a standard three-layer feedforward ANN, which is a universal function approximator: it can approximate any function to arbitrary accuracy given a sufficient number of hidden units. We tried different numbers of hidden units, including multiple hidden layers, with different transfer functions in all the units (including sigmoids, limiter functions and polynomial expressions), and with two alternative representations of time period $t$: as an integer $t = 1 \ldots P$ and using the well-known *unary encoding* which is often used to represent symbolic ANN inputs and gave better results here. In the unary encoding we associate a binary variable with each period, and period $t$ is represented by a vector $(0_1, \ldots, 0_{t-1}, 1, 0_{t+1}, \ldots, 0_P)$. Surprisingly, we obtained best results using an extremely simple network, with no hidden layer and the identity transfer function $f(x) = x$. No bias term is needed because the unary encoding already provides a time-dependent bias.

The ANN corresponding to three stocking points is shown in Figure 1, where Si denotes the $i^{th}$ stock level, Oi the $i^{th}$ order level, and Tj the $j^{th}$ binary variable in the unary time encoding. All units use the identity transfer function. Each arrowed line connecting two units in the diagram has an associated weight, so the ANN has $K(P + K)$ weights, where $K$ is the number of stocking points. This ANN represents a simple set of affine relationships

$$O_j = \sum_i S_i w_{ij} + w_{tj}$$

where $w_{ij}$ is the ANN weight between stock level $S_i$ and order level $O_j$, and $w_{tj}$ is the ANN weight between time $t$ and order level $O_j$. (An affine transformation is a linear transformation followed by a translation.) One would not expect this to yield an efficient or even a sensible policy, but our policy is not yet complete as we have not handled the problem constraints.

## 2.2 Constraint handling

The ANN forms only part of the policy. We also need a way of handling the constraints of the problem, which forbid (i) negative orders (corresponding to selling unused stock back to the supplier), and (ii) negative stock levels. We will train the ANN by an EA and there are several ways of handling constraints in EAs. We use a *decoder* which transforms the (possibly infeasible) ANN solution into one that violates no constraints. Decoders are a way of finding feasible solutions from chromosomes that represent infeasible states. They are problem-specific and ours works as follows. Suppose at period $t$ we have stock levels $S_i$ and the ANN suggests ordering quantities $O_i$. We modify each quantity $O_i$ by

$$O_i \leftarrow \max(O_i, 0)$$

to avoid violating constraints of type (i). Then for any stocking point $i$ that supplies a set of stocking points $X_i$ we modify its order level $O_i$ by

$$O_i \leftarrow \max\left(O_i, \left(\sum_{j \in X_i} O_j\right) - S_i\right)$$

This ensures that each supplier orders sufficient stock to fulfil its deliveries, and avoids violating constraints of type (ii). The policy is now the composition of the ANN and the decoder, which transforms the affine function of the ANN into a continuous piecewise affine function.

Note that we must modify the order levels of the stocking points earlier in the supply chain first. This is always possible if the supply chain is in the form of a directed acyclic graph. If lateral transshipments are allowed (orders between stocking points at the same level) or if there are constraints on order sizes or storage capacities then the decoder must be modified; we leave this issue for future work.

## 2.3 The evolutionary algorithm

To train the ANN we use an EA. There are many such algorithms in the literature, and we now describe our choice and the design decisions behind it. Firstly, we decided not to use genetic recombination. When training an ANN by EA one can encounter the well-known *competing conventions* problem (see [20] for example). This is caused by two forms of symmetry: an ANN's hidden units can be permuted without changing its output, and a hidden unit's weights can all be multiplied by $-1$ without changing its output. Thus if there are $h$ hidden units then there are $2^h h!$ equivalent ANNs. Crossover is unlikely to give good results unless the parent chromosomes are aiming for symmetrically similar representations, though it is possible to design crossover operators that handle the symmetries [23]. This problem does not apply to our simple ANN because it has no hidden units, but in experiments crossover did not improve results so we do not use it.

We decided to use a $(\mu + 1)$-Evolution Strategy (ES) because it is almost exactly a steady-state genetic algorithm without crossover, and an efficient method for handling noise in the fitness function is known for a steady-state genetic algorithm (see below). However, we adapted it to a *cellular* ES, in which each chromosome is notionally placed in an artificial space and nearby chromosomes form its neighbourhood. Cellular algorithms can reduce premature convergence, which we found to be a problem with our initial standard ES. In our ES the population size is $\mu$, at each iteration a new chromosome $c'$ is created by mutating a randomly selected chromosome $c$, and if $c'$ is fitter than the least-fit chromosome $c^*$ in the neighbourhood of $c$ then it replaces $c^*$, otherwise $c'$ is discarded. We used a ring topology and define the neighbourhood of a chromosome to be its two adjacent chromosomes.

A common form of mutation adds normally distributed noise to each gene, but we use a method that gave better results in experiments. For each chromosome we generate two uniformly distributed random numbers, $p$ in the range $(0, 1)$ and $q$ in the range $(0, 0.5)$. Then for each allele in the chromosome, with probability $p$ we change it, otherwise with probability $1 - p$ we leave it unchanged. If we do change it then with probability $q$ we set it to 0, otherwise with probability $1 - q$ we add a random number with Cauchy distribution to it. *Cauchy mutation* has been shown to speed up EAs [24]. It can be computed as $s \tan(u)$ where $u$ is a uniformly distributed random variable in the range $(-\pi, \pi)$ and $s$ is a scale factor. For each chromosome we compute a random scale factor, itself with Cauchy distribution and fixed scale factor 100. Finally, if no allele was modified (which is possible for small $p$) then we modify one randomly selected allele as described. This rather complex mutation operator is designed to generate a variety of random moves, with different numbers of modified alleles and different scale factors. All chromosomes initially contain alleles generated randomly using the same Cauchy distribution. We do not use the well-known technique of self-adapting step sizes, because in a $(\mu + 1)$-ES offspring with reduced mutation variances are always preferred [2].

## 2.4 Handling uncertainty

When demand is probabilistic the fitness function of the EA is noisy. In such cases we must average costs over a number of simulations. In some previous SO approaches to inventory control, this problem was tackled by averaging costs over a small number of simulations because the simulations were computationally expensive: for example [13] use 3 samples. The standard deviation of the sample mean of a random variable with standard deviation $\sigma$ is $\sigma/\sqrt{n}$ where $n$ is the number of samples, so a large number of samples may be needed for very noisy fitness functions. Here we use smaller problems than those in [13] so we can afford to use a much larger number of simulations and obtain reliable cost estimates. To do this for every chromosome would be expensive but there are more efficient methods, and we use the *greedy averaged sampling* resampling scheme of [17]. This requires two parameters to be tuned by the user: $U$ and $S$. On generating a new chromosome $c$ it takes $S$ samples to estimate its fitness before placing it into the population. It then selects another chromosome $c'$ (which may be $c$) for *resampling*: another $S$ samples are taken for $c'$ and used to refine its fitness estimate. $c'$ is the chromosome with highest fitness among those with fewer than $U$ samples, so the function of $U$ is to prevent any chromosome from being sampled more times than necessary. If all chromosomes in the population have been sampled $U$ times then no resampling is performed. The algorithm is summarised in Figure 2.

```
train($\mu, S, U$)
  create ANN population of size $\mu$
  evaluate population using $S$ samples
  while not(termination condition)
    select a parent
    breed an offspring O by mutation
    evaluate O using $S$ samples
    if O fitter than locally least-fit chromosome L
      replace L by O
    select globally fittest chromosome F with #samples$< U$
    if F exists
      re-evaluate F using $S$ more samples
  return best chromosome found with #samples$\geq U$
```

**Fig. 2.** Cellular evolution strategy with resampling

The aim of this resampling method is to obtain chromosomes with good fitness averaged over many samples, while expending a smaller number of samples on less-promising chromosomes. In our experiments we set $U = 10000$ so that cost estimates are obtained over 10000 samples, but by setting $S = 1$ we only expend approximately 200 samples per chromosome on average (this number was found by experiment). As the population size is 50, and $50 \times 200 = 10000$, this implies that a typical chromosome uses little more than one sample before being rejected as unfit. Using small $S$ also has an effect beyond reducing the average number of samples per chromosome: it encourages exploration by preserving less-fit chromosomes for longer. We found this to be a very beneficial effect.

Some points are glossed over in Figure 2 for the sake of readability. Firstly, if $S$ is not a divisor of $U$ then fewer than $S$ samples are needed in the final resampling of any chromosome to bring its total to $U$. Secondly, the termination condition is unspecified, and we simply use a timeout. Thirdly, if no chromosome has $U$ samples on termination then we must choose another chromosome to return. To avoid this, $S$ should be assigned a sufficiently large value so that in experiments there is always a chromosome with $U$ samples on termination. This value must be chosen by experimentation.

## 2.5 Discussion of the method

We refer to our method as NEMUE[1] (Neuro-Evolution for MUlti-Echelon systems). NEMUE is the result of many experiments with alternative versions. We experimented with an array of ANNs, one for each time period. This model has $12P$ weights and

[1] The "lady of the lake" in Arthurian legend.

clearly subsumes the model above: any plan that can be represented by that model can also be represented by this one. The results should therefore be at least as good, but in experiments they were significantly worse. We believe that the ANN array is simply harder to train than a single ANN.

We also tried a non-unary encoding of time, in which order levels are linear functions of stock levels and polynomial functions of time. Fixing the polynomial degree makes the size of the ANN independent of the number of time periods. Using a cubic function of time gave reasonable results but was inferior to the unary encoding.

We used a decoder to handle the problem constraints, but there are other ways of handling constraints in EAs. The simplest is to use a *penalty function* which adds a large artificial cost for each violated constraint. In our problem this forces the ANN to learn to order sufficient stock in order to avoid stockout. We tried a penalty function but it gave inferior results to the decoder.

## 3 Experiments

Ultimately we are interested in solving large, realistic inventory problems with multiple stocking points, stochastic lead times, correlated demands and other features that make classical approaches impractical. Unfortunately there are no known methods for solving such problems to optimality, so there is no way of evaluating our method. Instead we consider more modest problems to test the ability of NEMUE to find good plans.

Our benchmark problems have two multi-echelon topologies: *arborescent* and *serial*. In the arborescent case we have three stocking points A, B and C, with C supplying A and B, while in the serial case C supplies B which supplies A. In both cases we have linear holding costs, linear penalty costs, fixed ordering costs, and stationary probabilistic demands. The closing inventory levels for period $t$ are $I_t^A = I_{t-1}^A + Q_t^A - d_t^A$, $I_t^B = I_{t-1}^B + Q_t^B - d_t^B$ and $I_t^C = I_{t-1}^C + Q_t^C - Q_t^A - Q_t^B$ where $Q_t$ is the order placed in period $t$ and $d_t$ is the demand in period $t$. If $I_t < 0$ then the incurred cost is $-I_t.\pi$, otherwise it is $I_t.h$, where $\pi$ is the penalty cost and $h$ the holding cost. Suppliers are not allowed to run out of stock. We prepared 28 instances of both the arborescent and serial types, with various costs and 2–9 time periods, giving a total of 56 instances with a range of characteristics as follows. The holding costs for A, B and C are 4, 5 and 1 respectively for arborescent instances 1–14; 3, 2 and 1 for arborescent instances 15–28; and 3, 2 and 1 for all serial instances. For the arborescent instances the penalty costs for A and B are 12 and 25 respectively for instances 1–14; and 3 and 6 for instances 15–28. For all the serial instances the penalty cost for instance A is 12. The ordering costs for A, B and C are 150, 130 and 170 respectively for arborescent instances 1–14; 80, 75 and 100 for arborescent instances 15–28; and 75, 80 and 100 for all serial instances. For space reasons we do not specify the demands in detail, but we used 10 patterns for arborescent instances and 4 patterns for serial instances. In each period we specify a deterministic demand which is then multiplied by either $\frac{2}{3}$ with probability 0.25, 1 with probability 0.5, or $\frac{4}{3}$ with probability 0.25. Thus the number of possible scenarios is $3^P$, giving 59,049 scenarios for the largest problems ($P = 10$).

We solved these problems in two ways: using Stochastic Programming (SP) [3] and NEMUE. SP is a field of Operations Research designed to solve optimisation problems under uncertainty via scenario reduction techniques: a representative subset of all possible scenarios is selected and used to generate a deterministic equivalent optimisation problem, which is then typically solved using integer linear programming. We use the SP results to evaluate the quality of plans found by NEMUE. The optimal replenishment plans are obtained using the following Stochastic Integer Programming model:

$$\begin{array}{l}
\min \mathsf{E}[C] = \sum_{t=1}^{N} \sum_{p \in P} \left( a_p \delta_{pt} + h_p I^{+}_{pt} + \pi_p I^{-}_{pt} \right) \\
\text{s.t. } t = 1, \ldots, N \text{ and } p \in P \\
I_{pt} = I_{p,t-1} + Q_{pt} - Q_{P_p,t} - d_{pt} \\
I_{pt} = I^{+}_{p,t} - I^{-}_{p,t} \\
Q_{pt} \leq M \delta_{pt} \\
\delta_{pt} \in \{0, 1\} \quad Q_{pt} \geq 0
\end{array}$$

where

| | |
|---|---|
| $C$ : | total holding and ordering/set-up cost of the system over $N$ periods; |
| $a$ : | fixed ordering/set-up cost; |
| $h$ : | proportional inventory holding cost per period; |
| $P$ : | the set of all stocking points; |
| $P_p$ : | the set of stocking points supplied directly by the stocking point $p$; |
| $d_{pt}$ : | random demand at stocking point $p$, in period $t$; |
| $\delta_{pt}$ : | a binary variable that takes the value of 1 if a replenishment occurs |
| : | at stocking point $p$ in period $t$ and 0 otherwise; |
| $I_{pt}$ : | the inventory level at the end of period $t$ at stocking point $p$; |
| $Q_{pt}$ : | the order quantity at the beginning of period $t$ at stocking point $p$; |

and $I^+$ and $I^-$ denote positive and negative closing inventory levels. Except for the lowest echelon stocking points, $I^-$ is zero. $M$ is some large positive number. In this stochastic model a *here-and-now* policy is adapted: all decision variables are set before observing the realisation of the random variables. The certainty equivalent model is obtained using the compiler described in [22] and solved with CPLEX 11.2.

Results comparing SP and NEMUE are shown in Table 1. All SP runs were terminated after one hour and all NEMUE results after 30 minutes on a 2.8 GHz Pentium (R) 4 with 512 RAM, each figure being the best of six five-minute runs. The NEMUE parameters used were $S = 1, U = 10000$ and $\mu = 50$. SP runs that were aborted because of memory problems are denoted by "—". (In the few cases that SP found and proved optimality, this sometimes took much less than one hour.) The columns marked "%opt" denote the optimality gap: a reported cost $c$ and gap $g$ means that SP proved that the optimal solution cannot have cost lower than $c' = c(100 - g)/100$ (this does not imply the existence of a solution with cost $c'$). In several cases NEMUE finds superior plans to those found by SP, showing that on larger instances SP fails to find optimal plans. In a few cases NEMUE appears to find plans that are slightly better than optimal: this is of course impossible, and is a consequence of the empirical nature of the data. In such cases we assume that NEMUE found an optimal plan.

SP was unable to find provably optimal plans for all but the smallest instances. We believe that for the medium-sized instances SP finds optimal plans but does not prove optimality before timeout. For the largest instances SP ran out of memory, though we

| arborescent | | | | | | serial | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | SP | | NEMUE | | | | SP | | NEMUE | |
| # | periods | cost | %opt | cost | %opt | # | periods | cost | %opt | cost | %opt |
| 1 | 4 | 2507 | 0 | 2573 | 2.6 | 1 | 4 | 995 | 0 | 993 | 0 |
| 2 | 5 | 3124 | 1.4 | 3180 | 3.1 | 2 | 5 | 1269 | 0.7 | 1298 | 2.9 |
| 3 | 6 | 3657 | 2.7 | 3775 | 5.7 | 3 | 6 | 1493 | 1.8 | 1491 | 1.7 |
| 4 | 7 | 4214 | 5.6 | 4250 | 6.4 | 4 | 7 | 1794 | 7.4 | 1797 | 7.6 |
| 5 | 8 | 4654 | 8.2 | 4722 | 9.5 | 5 | 8 | 2087 | 12.0 | 1987 | 7.6 |
| 6 | 9 | 5472 | 16.9 | 5164 | 11.9 | 6 | 9 | 2741 | 25.7 | 2295 | 11.3 |
| 7 | 10 | — | ? | 5590 | ? | 7 | 10 | — | ? | 2603 | ? |
| 8 | 4 | 2100 | 0 | 2169 | 3.2 | 8 | 4 | 1311 | 0.2 | 1306 | 0.0 |
| 9 | 5 | 2626 | 0.6 | 2722 | 4.1 | 9 | 5 | 1598 | 2.2 | 1594 | 2.0 |
| 10 | 6 | 3311 | 1.8 | 3409 | 4.6 | 10 | 6 | 1833 | 4.3 | 1832 | 4.2 |
| 11 | 7 | 4065 | 2.5 | 4153 | 4.6 | 11 | 7 | 2024 | 6.7 | 2024 | 6.7 |
| 12 | 8 | 4454 | 3.4 | 4542 | 5.3 | 12 | 8 | 2160 | 9.3 | 2142 | 8.5 |
| 13 | 9 | 5158 | 10.3 | 5115 | 9.5 | 13 | 9 | 2678 | 25.1 | 2264 | 11.4 |
| 14 | 10 | — | ? | 5432 | ? | 14 | 10 | — | ? | 2407 | ? |
| 15 | 4 | 1342 | 0.2 | 1340 | 0.1 | 15 | 4 | 1104 | 0 | 1104 | 0 |
| 16 | 5 | 1657 | 1.8 | 1671 | 2.6 | 16 | 5 | 1417 | 2.1 | 1423 | 2.5 |
| 17 | 6 | 1930 | 2.2 | 1938 | 2.6 | 17 | 6 | 1759 | 4.1 | 1763 | 4.3 |
| 18 | 7 | 2180 | 4.5 | 2192 | 5.0 | 18 | 7 | 2057 | 5.4 | 2055 | 5.3 |
| 19 | 8 | 2428 | 6.1 | 2393 | 4.7 | 19 | 8 | 2266 | 6.6 | 2258 | 6.3 |
| 20 | 9 | 2853 | 13.9 | 2617 | 6.1 | 20 | 9 | 2706 | 17.7 | 2479 | 10.2 |
| 21 | 10 | — | ? | 2864 | ? | 21 | 10 | — | ? | 2627 | ? |
| 22 | 4 | 1086 | 0 | 1096 | 0.9 | 22 | 4 | 828 | 0 | 828 | 0 |
| 23 | 5 | 1334 | 0.2 | 1330 | 0.0 | 23 | 5 | 931 | 0 | 934 | 0.3 |
| 24 | 6 | 1680 | 0.6 | 1677 | 0.4 | 24 | 6 | 1259 | 1.3 | 1265 | 1.8 |
| 25 | 7 | 2055 | 0.7 | 2051 | 0.5 | 25 | 7 | 1633 | 2.4 | 1639 | 2.8 |
| 26 | 8 | 2219 | 1.1 | 2220 | 1.1 | 26 | 8 | 1757 | 2.7 | 1766 | 3.2 |
| 27 | 9 | 2479 | 2.0 | 2531 | 4.0 | 27 | 9 | 1983 | 3.9 | 2000 | 4.7 |
| 28 | 10 | — | ? | 2665 | ? | 28 | 10 | — | ? | 2150 | ? |

**Table 1.** Experimental results

use the state-of-the-art CPLEX solver and a powerful machine (an Intel Core 2 Duo CPU E7200 with 2.53 GHz and 3GB of RAM). On the largest instances for which SP did not run out of memory, it was unable to prove optimality even within several days. Thus our benchmark problems straddle the borderline of solvability by classical methods.

Despite the simplicity of the policy and the large number of scenarios (at least on the larger instances) the NEMUE results are remarkably good. On 13 of the 28 arborescent instances and 19 of the 28 serial instances, NEMUE found plans that were at least as good as those found by SP. On the three serial instances for which SP found provably optimal plans, NEMUE found equally good plans. On most of the larger instances NEMUE found better plans than SP. These results show that: (i) a relatively simple, continuous, piecewise affine function can closely approximate a large policy tree for multi-echelon systems; (ii) such a function can be effectively represented by an affine function followed by a decoder function; (iii) the affine function can be learned in a reasonable time by evolutionary search; (iv) that our approach is more scalable than SP.

It is tempting to speculate that with improved heuristics and longer runtimes we might find optimal strategies for *all* instances. But there is no guarantee that all scenario trees can be well-approximated in this way, and in more extensive experiments on arborescent instance 1 (for example) we have been unable to find an optimal plan. Nevertheless, the results are very promising.

## 4 Related work

Though simulation was originally used only to evaluate solutions found by other means, the field of SO has recently become more popular — see the survey of [5]. SO may be *recursive* or *non-recursive*. In the non-recursive approach an approximate cost function is learned during a simulation phase, then this function is minimised using an optimisation algorithm during a second phase. NEMUE is an example of recursive SO in which simulation and optimisation alternate and inform each other.

A tutorial and survey of the application of SO to inventory control is given in a recent paper [12]. Relatively little work has been done on applying SO to multi-echelon systems, and we have been unable to find any other work on inventory control via neuroevolution, though several papers use EAs to evolve plans (for example [1, 13, 15, 18]). Another difference of NEMUE is that it aims to approximate optimal policy trees, whereas most SO methods aim to find parameters for special policies such as $(s, S)$. A different way of using ANNs for inventory control is to solve a set of training instances by some other method, then train an ANN to learn how to find good solutions from new instances (for example [6]). But we then need another algorithm to solve the problems, which is the aim of NEMUE. A related approach to neuroevolution is *genetic programming*, in which an EA is used to evolve an algorithm for solving the problem, instead of an ANN. This approach has also been applied to inventory control [11].

Another interesting approach to sequential decision problems such as those in inventory control is the field variously referred to as *neuro-dynamic programming*, *temporal difference learning* and *approximate dynamic programming*. This blend of dynamic programming and simulation has been applied to many problems including inventory control: see for example [4, 7, 10, 19]. A drawback is that special techniques are needed to cope with the well-known “curse of dimensionality”: the vast number of states that result from a simple discretisation of the continuum of states in these problems. In contrast, neuroevolution can directly handle a continuum of states.

## 5 Conclusion

We have proposed what seems to be the first neuroevolutionary method for approximating optimal plans in multi-echelon stochastic inventory control problems. Large or infinite scenario trees are approximated by a neural network, which is trained by an evolutionary algorithm with resampling, while problem constraints are handled by a decoder. Because the method is simulation-based and uses general-purpose techniques such as evolutionary algorithms and neural networks, it does not rely on special properties of the problem and can be applied to inventory problems with non-standard features. We showed experimentally that the method can find near-optimal solutions. In future work we will extend the method to handle problem features such as capacity constraints.

## References

1. J. Arnold, P. Köchel. Evolutionary Optimisation of a Multi-Location Inventory Model With Lateral Transshipments. *9th International Working Seminar on Production Economics*, Igls 1996, Preprints 2, pp. 401–412.
2. T. Bäck, F. Hoffmeister, H.-P. Schwefel. A Survey of Evolution Strategies. *4th International Conference on Genetic Algorithms*, 1991.
3. J. R. Birge, F. Louveaux. Introduction to Stochastic Programming. Springer, New York, 1997.
4. S. K. Chaharsooghi, J. Heydari, S. H. Zegordi. A Reinforcement Learning Model for Supply Chain Ordering Management: an Application to the Beer Game. *Decision Support Systems* 45(4):949–959, 2008.
5. M. C. Fu. Optimization for Simulation: Theory vs Practice. *INFORMS Journal of Computing* 14:192–215, 2002.
6. L. K. Gaafar, M. H. Choueiki. A Neural Network Model for Solving the Lot-Sizing Problem. *Omega* 28(2):175–184, 2000.
7. I. Giannoccaro, P. Pontrandolfo. Inventory Management in Supply Chains: a Reinforcement Learning Approach. *International Journal of Production Economics* 78(2):153–161, 2002.
8. F. Gomez, J. Schmidhuber, R. Miikkulainen. Efficient Non-Linear Control Through Neuroevolution. *Journal of Machine Learning Research* 9:937–965, 2008.
9. N. M. Hewahi. Engineering Industry Controllers Using Neuroevolution. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing* 19(1):49–57, 2005.
10. C. Jiang, Z. Shenga. Case-Based Reinforcement Learning for Dynamic Inventory Control in a Multi-Agent Supply-Chain System. *Expert Systems with Applications* 36(3 part 2):6520–6526, 2009.
11. P. Kleinau, U. W. Thonemann. Deriving Inventory-Control Policies With Genetic Programming. *OR Spectrum* 26(4):521–546, 2004.
12. P. Köchel. Simulation (Optimisation) in Inventory Theory. Tutorial, *8th ISIR Summer School on New and Classical Streams in Inventory Management: Advances in Research and Opening Frontiers*, 2007.

13. P. Köchel, U. Nieländer. Simulation-Based Optimisation of Multi-Echelon Inventory Systems. *International Journal of Production Economics* 1:503–513, 2005.
14. A. Lubberts, R. Miikkulainen. Co-Evolving a Go-Playing Neural Network. *Genetic and Evolutionary Computation Conference,* Kaufmann, 2001, pp. 14–19.
15. A. L. Olsen. An Evolutionary Algorithm for the Joint Replenishment of Inventory with Interdependent Ordering Costs. *Genetic and Evolutionary Computation Conference, Lecture Notes in Computer Science* vol. 2724, Springer, 2003, pp. 2416–2417.
16. J. B. Pollack, A. D. Blair. Co-Evolution in the Successful Learning of Backgammon Strategy. *Machine Learning* 32(3):225–240, 1998.
17. S. D. Prestwich, S. A. Tarim, R. Rossi, B. Hnich. A Steady-State Genetic Algorithm With Resampling for Noisy Inventory Control. *10th International Conference on Parallel Problem Solving From Nature, Lecture Notes in Computer Science* vol. 5199, Springer, 2008, pp. 559–568.
18. S. D. Prestwich, S. A. Tarim, R. Rossi, B. Hnich. A Cultural Algorithm for POMDPs from Stochastic Inventory Control. *5th International Workshop on Hybrid Metaheuristics, Lecture Notes in Computer Science* vol. 5296, Springer, 2008, pp. 16–28.
19. B. Van Roy, D. P. Bertsekas, Y. Lee, J. N. Tsitsiklis. A Neuro-Dynamic Programming Approach to Retailer Inventory Management. *Proceedings of the IEEE Conference on Decision and Control*, 1997.
20. J. Schaffer, D. Whitley, L. Eshelman. Combinations of Genetic Algorithms and Neural Networks: A Survey of the State of the Art. *International Workshop on Combinations of Genetic Algorithms and Neural Networks*, 1992, pp. 1–37.
21. K. O. Stanley, R. Miikkulainen. Evolving Neural Networks Through Augmenting Topologies. *Evolutionary Computation* 10(2):99–127, 2002.
22. S. A. Tarim, S. Manandhar, T. Walsh. Stochastic Constraint Programming: A Scenario-Based Approach. *Constraints* 11:53–80, 2006.
23. D. Thierens. Non-Redundant Genetic Coding of Neural Networks. *International Conference on Evolutionary Computation*, Nagoya, Japan, 1996, pp. 571–575.
24. X. Yao, Y. Liu, G. Lin. Evolutionary Programming Made Faster. *IEEE Transactions on Evolutionary Computation* 3(2):82–102, 1999.