A Single Queen Single Worker Honey–Bees Approach to 3-SAT

Hussein A. Abbass
School of Computer Science,
University of New South Wales,
University College, ADFA Campus,
Northcott Drive, Canberra ACT, 2600, Australia,
h.abbass@adfa.edu.au

Abstract

Modelling the behavior of social insects has attracted much research recently. Although honey–bees exhibit many features that encourage their use as models for intelligent behavior, up to our knowledge, no attempt for using honey-bees as a basis for optimization has been made in the literature. Some of the features that distinguish honey–bees are division of labor, communication on the individual and group level, and cooperative behavior.

This paper presents a novel search algorithm inspired by the marriage process in honey–bees. The algorithm is applied to a special class of the propositional satisfiability problems (SAT) known as 3–SAT, where each clause contains exactly three literals. Two versions of the proposed algorithm, which incorporate each a well known heuristic for SAT, are developed. The two heuristics employed for each version are GSAT and random walk. The objective of this paper is to scrutinize the algorithm and compare its behavior on 3–SAT against both heuristics alone. The analysis is undertaken using two parameters gleaned from biological concepts; these are, the colony size and the amount of time devoted for brood–care.

1 Introduction

Modelling the behavior of social insects, such as ants and bees, and using these models for search and problem solving are the context of the emerging area of *swarm intelligence* [2]. A successful swarm–based approach to optimization is *ant colony optimization*, where the search algorithm is inspired by the behavior of real ants [2, 5]. This method proved successful in solving many complex combinatorial problems. However, up to our knowledge, there has not been any attempt to model the marriage process of honey–bees and use this model for optimization and search.

In this paper, we present an attempt to model the marriage behavior of honey–bees to produce an optimization search algorithm, which we call *marriage in honey–bees optimization* (MBO). The main objective of this paper is to study the behavior of the MBO algorithm on the *propositional satisfiability problem* (SAT) as an example of an NP–hard combinatorial optimization problem. The type of the SAT problem used in this paper is called 3–SAT where each clause (constraint) contains exactly three literals (variables). We undertake the analyzes using parameters purely inspired by their biological encounters, which, therefore, will give some insight into the underlying relation between the biological and the computational model. Therefore, important biological parameters that will influence our parameters' choice are underlined for clarity purposes.

The paper is organized as follows: in Section 2, the marriage process in honey–bees is discussed along with some background materials and a general version of the MBO algorithm is formulated. We then introduce the 3–SAT problem in Section 3. The MBO algorithm applied to 3–SAT is then discussed in Section 4, followed by experimental setup and results in Section 5. Conclusions are then drawn in Section 6.

2 Marriage in honey–bees

MBO is inspired by the phylogenetic of sociality in Hymenoptera, such as bees, ants, and wasps, and the mating process in honey–bees. Hymenoptera are normally found as *eusocial insects*. Eusocial insects are characterized by three main features, viz cooperation among adults in brood care and nest construction, overlapping of at least two generations, and reproductive division of labor. Insects without these attributes are termed *solitary* and those lack one or two of these attributes are termed *presocial* [4]. The evolutionary steps of eusociality were driven by the contin-

uous series of nests, starting with entirely solitary colony to reaching high eusociality. Two sequences can be distinguished in establishing this series of nests; *subsocial* (familial or altruistic cooperation) and *parasocial* (mutualistic cooperation) [4]. In the literature of behavioral genetics, the former series is more common than the latter and is the assumed model in this paper. That is, we will start with a single queen without broods (*ie.* solitary colony) then we evolve the colony - using concepts of marriage in honey–bees - for several generations to end with a presocial colony. The two main features of presociality appear in our proposed algorithm as follows:

Overlapping of at least two generations The model always has two generations: the mother and her broods.

Reproductive division of labor Division of labor appears when the queen specializes in egg laying, and acts as the only reproductive member in the population, whereas workers specialize in brood care.

The missing feature to qualify this model to reach eusociality is cooperation among adults, which can be achieved if we use two or more workers. However, in this paper, we concentrate on a single worker colony to identify the power of the MBO algorithm. This point will be discussed later in the paper.

A colony can be founded in two different methods [4]. The first method is called *independent founding*, where a colony starts with one or more reproductive females that construct the nest, produce eggs, and feed the larvae. The first brood is reared alone until they emerge and take over the work of the colony. Subsequently, division of labor starts to take place, where the queen specializes in egg laying and the workers in brood care. The second method is called *swarming* where the colony is founded by a single queen (*haplometrosis*) or more (*pleometrosis*) in addition to a group of workers from the original colony. Division of labor starts from the beginning where queens specialize in egg laying and workers in brood care. If the colony contains one queen during its life–cycle, it is called a *monogynous* colony; otherwise a *polygynous* colony. In this paper, we take a swarming presocial approach assuming a pleometrosis monogynous honey–bees colony.

Before we present a summary of the mating–flight and scrutinize the MBO algorithm, the structure of a normal honey–bees colony is discussed in the following sub–section.

2.1 Colony structure

Each normal honey–bees' colony consists of the queen(s), drones, worker(s), and broods. Queens represent the main reproductive individuals in some types of honey–bees - such as the European Apis Mellifera - and specialize in egg laying [9]. Drones are the sires or fathers of the colony. Drones are haploid and act to amplify their mothers' genome without alteration of their genetic composition except through mutation. Therefore, they are considered as agents that propagate one of their mother's gametes and function to enable females to act genetically as males.

Workers specialize in brood care and sometimes lay eggs. Broods arise either from fertilized or unfertilized eggs. The former represent potential queens or workers, whereas the latter represent prospective drones.

2.2 The mating–flight

Each bee performs sequences of actions which unfold according to genetic, environmental, and social regulations [12]. The outcome of each action itself becomes a portion of the environment and greatly influences the subsequent actions of both a single bee and her hive mates. The marriage process represents one type of action that was difficult to study because the queens mate during their mating–flight far from the nest. Consequently, the mating process was hard to observe.

A mating–flight starts with a dance performed by the queens who then start a mating–flight during which the drones follow the queens and mate with them in the air. In a typical mating–flight, each queen mates with seven to twenty drones [1]. In each mating, sperm reaches the spermatheca and accumulates there to form the genetic pool of the colony. Each time a queen lays fertilized eggs, she retrieves at random a mixture of the sperms accumulated in the spermatheca to fertilize the egg [10].

2.3 The artificial analogue model

The mating–flight can be visualized as a set of transitions in a state–space (the environment) where the queen moves between the different states in the space in some speed and mates with the drone encountered at each state probabilistically. At the start of the flight, the queen is initialized with some energy–content and returns to her nest when the energy is within some threshold from zero or when her spermatheca is full.

In this paper, we will restrict the functionality of workers to brood care and therefore, a worker is in effect a heuristic which acts to improve (take care of) the broods. Moreover, we will assume that the colony has a single queen and a single worker. This represents the simplest division of labor ever; therefore, one can examine the usefulness of the algorithm in its simplest form.

A drone mates with a queen probabilistically using the fol-

lowing equation

$$prob(Q, D) = e^{\frac{-\Delta(f)}{S(t)}} \quad (1)$$

where, $prob(Q, D)$ is the probability of adding the sperm of drone D to the spermatheca of queen Q; that is, the probability of a successful mating, $\Delta(f)$ is the absolute difference between the fitness of D (*ie.* $f(D)$) and the fitness of Q (*ie.* $f(Q)$), and $S(t)$ is the speed of the queen at time t. It is apparent that this function acts as an annealing function, where the probability of mating is high when either the queen is still in the start of her mating–flight and therefore her speed is high, or when the fitness of the drone is as good as the queen's. After each transition in the space, the queen's speed and energy decay using the following equations

$$S(t+1) = \alpha * S(t) \quad (2)$$

$$E(t+1) = E(t) - \gamma \quad (3)$$

where α is a factor $\in]0, 1[$, and γ is the amount of energy reduction after each transition.

In Figure 1, a generic MBO algorithm is shown. The algorithm starts with initializing the queen's genotype at random. After that, the heuristic is used to improve the queen's genotype, therefore preserving the assumption that a queen is usually a good bee. Afterwards, a set of mating–flights is undertaken. In each mating–flight, the queen's energy and speed are initialized with some value at random. The queen then moves between different states (*ie.* solutions) in the space according to her speed and mates with the drone she encounters at each state using the previously discussed function in Equation 1. If a drone is successfully mated with the queen, his sperm is added to the queen's spermatheca (*ie* a list of partial solutions). After the queen finishes her mating–flight, she returns to the nest and starts breeding by selecting a sperm from her spermatheca at random followed by crossover with the queen's genome that complements the chosen sperm. This crossover process results in a brood and we will refer to this type of crossover as *haploid–crossover*. Mutation then acts on the brood; therefore, if the same sperm is used once more to generate a brood, the resultant brood will be different because of mutation. This process is followed by applying the worker to raise (improve) the broods. Afterwards, the queen is replaced with the fittest brood if the latter is better than the former. The remaining broods are then killed and a new mating–flight starts. In reality, the female broods become workers or queens and the diploid males are killed. However, since a worker in our algorithm represents a heuristic without a genome, all remaining broods are assumed to be diploid males for simplification. Also, to avoid inbreeding (since we have a single queen), drones are generated independent of the queen; therefore they are assumed to be unrelated.

```
randomly generate the queen's chromosome
apply local search to get a good queen
for a pre–defined maximum number of mating–flights
    initialize energy and speed
    while queen's energy > 0
        the queen moves between states
            and probabilistically choose drones
        if a drone is selected, then
            add its sperm to the queen's spermatheca
        end if
        update the queen's internal energy and speed
    end while
    generate broods by haploid–crossover and mutation
    use the worker to improve the broods
    if the best brood is fitter than the queen then
        replace the queen's chromosome with
            the best brood's chromosome
    end if
    kill all broods
end for
```

Figure 1: *Optimization by marriage in honey–bees: a single queen single worker pleometrosis monogynous model.*

3 The propositional satisfiability problem

A general constraint satisfaction problem (CSP) is the problem of finding an assignment to a set of variables that satisfies a set of constraints over the domains of those variables. To formally define a CSP, we introduce the following notations. "S_v^V" represents an ordered instantiation of a set of variables V by substituting corresponding values v from their domain $D(V)$, "$\nvdash$" for not derive, and finally "$\perp$" for falsification. We can define a CSP problem formally as follows: A set of constraints C over the set of variables $V \in D(V)$ - that is $C \subset V \times V$ - is satisfiable iff $\exists\ v \in D(V)$, such that $S_v^V \wedge C \nvdash \perp$.

In SAT, the domain of each variable is either true or false, or equivalently 1 or 0 (*ie.* $V \in \{0, 1\}$). Although SAT is a special case of CSP, any CSP can be mapped to SAT [7]. Many problems in planning and scheduling can be represented using SAT; therefore solving SAT is a very attractive research area [6]. However, it is known that SAT is intractable [11].

In the literature, there are two main streams of techniques for solving SAT: complete and incomplete techniques [8]. The former use an exhaustive search approach and guarantee a solution if one exists. However, a complete technique (such as Davis–Putnam) can only handle small problems [16], after which the time needed to solve the problem gets beyond any computer capabilities. The latter, although they do not guarantee convergence to a solution, they are fast and more suitable for large problems. Therefore, incomplete techniques become more attractive, especially with problems in planning which include thousands of variables

[13].

The easiness/hardness of solving SAT depends on a phenomena known as "phase transition" [3]. Problems before the phase transition are easy to solve and those after the phase transition are mostly unsatisfiable. Hard SAT problems exist around the phase transition region. A phase transition is defined by the ratio between the number of clauses (constraints), l, and the number of literals (variables), n. For 3–SAT, the phase transition was experimentally found to be 4.3 [3].

4 MBO for SAT

In this section, the application of the MBO algorithm to the propositional satisfiability problem is introduced in three stages. First, the representation of a colony and the means of calculating the individuals' fitness are presented in Section 4.1. Second, a description of the haploid–crossover operator is summarized in Section 4.2. Third, the MBO algorithm applied to SAT is introduced in Section 4.3.

4.1 Representation

A genotype of an individual is represented using an array of binary values and length equal to the number of literals in the problem, where each cell corresponds to a literal. A value of 1, assigned to a cell, indicates that the corresponding literal is true; otherwise it is false. The fitness of the genotype is the ratio between the number of clauses satisfied by the assignment to the total number of clauses in the problem.

$$Genotype\ Fitness = \frac{Number\ \ of\ \ satisfied\ \ clauses}{Total\ \ number\ \ of\ \ clauses} \tag{4}$$

A drone is represented using a genotype and a genotype marker. Since all drones are haploid, a genotype marker is used to randomly mark half of the genotype's genes and leave the other half unmarked; the unmarked genes are the ones that form a sperm.

Each queen has a genotype, speed, energy, and spermatheca (a repository of drones' sperm). Before each mating–flight, a queen's speed and energy are initialized at random in the range $[0.5, 1]$ to guarantee that the queen will fly for some steps. When a mating takes place between a queen and one of the drones' sperm in her spermatheca, a brood is constructed by copying the unmarked genes in the drones' sperm into the brood and completing the rest from the queen's genome. A brood has only a genotype. The worker represents a heuristic. Two heuristics are used in this paper; these are GSAT and random walk. For the description of these heuristics, the reader may refer to [14, 15].

4.2 Haploid–Crossover

To illustrate the haploid–crossover procedure during a mating, assume that the drone's genotype and genotype marker are as follows:

Genotype	1	1	1	0	0	1	0	0
Genotype–marker	u	m	m	u	u	u	m	m

where, u and m represent an unmarked and a marked gene respectively. Therefore, the drones sperm is

1	*	*	0	0	1	*	*

where * represents a non–existing gene. In reality, genes exist in pairs, but in our algorithm, we assume that a haploid drone lacks half of its genes because of representation. Now, assume that the queen's chromosome is as follows:

0	1	0	0	0	0	1	0

Therefore, the child must have the following chromosome:

1	1	0	0	0	1	1	0

where the genes, which are missing in the drone's sperm, are transmitted from the queen.

4.3 The algorithm

The complete MBO algorithm is presented in Figure 2. The algorithm starts with three user–defined parameters; these are, the queen's spermatheca size representing the maximum number of matings in a single mating–flight, the number of broods that will be born by the queen, and the amount of time devoted to brood care signifying the depth of local search or number of attempts made to improve a solution.

At the start of a run, the queen is initialized at random. The worker is then used to improve the queen's genotype. Afterwards, a number of mating–flights are carried out. In each mating–flight, the queen moves between the different states in the space based on her energy and speed, where both are generated at random before each mating–flight starts. At the start of a mating–flight, a drone is generated at random and the queen is positioned over that drone. The transition made by the queen in the space is based on her speed which represents the probability of flipping each bit in the drone's genome. Therefore, at the start of a mating–flight, the speed is usually high and the queen makes very large steps in the space. While the queen's energy decreases, the speed decreases. Subsequently, the neighborhood covered by the queen decreases. In each step

made by the queen in the space, the queen is mated with the drone encountered at that step using the probabilistic rule in Equation 1. If the mating is successful (*ie.* the drone passes the probabilistic decision rule), the drone's sperm is stored within the queen's spermatheca. Each time a drone is generated, half of his genes are marked at random since each drone is haploid by definition. Therefore, the genes that will be transmitted to the broods are fixed for each drone.

After the queen completes her mating–flight, she starts breeding. For a required number of broods, the queen is mated with a randomly selected sperm from her spermatheca. The worker is then used to improve the resultant brood. After all broods are being generated, they are sorted according to their fitness. The best brood replaces the queen if the brood is fitter than the queen. All broods are then killed and a new mating–flight is undertaken until all mating–flights are completed or an assignment that satisfies all clauses is encountered.

```
Define M to be the spermatheca size
Define E(t), and S(t) to be the queen's energy and speed
    at time t respectively
Initialize the queen's genotype at random
use the worker to improve the queen's genotype
while the stopping criteria are not satisfied
    t = 0
    initialize E(t) and S(t) randomly between [0.5, 1]
    initialize the energy reduction step γ = 0.5×E(t) / M
    generate a drone at random
    while E(t) > 0
        evaluate the drone's genotype
        if the drone passes the probabilistic condition, and
            the queen's spermatheca is not full, then
                add the drone's sperm to
                    the queen's spermatheca
        end if
        t = t + 1; E(t) = E(t) − γ; S(t) = 0.9 * S(t)
        with a probability of S(t)
            flip each bit in the drone's genotype
    end while
    for brood = 1 to total number of broods
        select a sperm from the queen's spermatheca at random
        generate a brood by crossovering the queen's genome
            with the selected sperm
        mutate the generated brood's genotype
        use the worker to improve the drone's genotype
    end for
    if the best brood is better than the queen then
        replace the queen with the best brood
    end if
    kill all broods
end while
```

Figure 2: *MBO for SAT*

To illustrate the reason of using a single worker, imagine that we have two workers, each of them represents a distinctive heuristic. If the algorithm performs better than each of these heuristics alone, the question will be whether this improvement is because of the cooperative behavior between the two heuristics or because of the way the colony evolved. Therefore, we run MBO twice; each with a different heuristic. When GSAT is used as the heuristic, we will refer to the algorithm as MBO–GSAT, and when random walk is used, we will refer to the algorithm as MBO–RWK

5 Experiments

5.1 Experimental setup

The goal of these experiments is to test the behavior of the algorithm with respect to the colony size, signifying the number of trial solutions per generation, and amount of time devoted to brood–care, representing the number of steps used in local search. As mentioned in Section 2, the number of matings per flight usually ranges between seven and twenty [1]. In this paper, we fixed this number to 7. To have a fair comparison among our experiments, we needed to guarantee that the number of broods all over a single run and under any experimental setup is equal. Therefore, we experimented with 20, 40, 60, 80, and 100 broods per flight, where the corresponding number of mating–flights was 120, 60, 40, 30, and 24 respectively. For example, 20 broods per flight times 120 mating–flights will result in 2400 trial solutions, where the worker will work to improve each of these solutions. Three different time–lengths devoted to brood–care are taken to be 100, 200, and 300 representing the number of attempts by the worker to improve a brood. The reduction factor of the queen's speed, α, is taken to be 0.9 and γ, the reduction step in queen's energy contents, is taken to be $\frac{0.5\times E(0)}{M}$, where $E(0)$ is the energy at the start of the mating–flight, which is initialized at random, and M is the spermatheca size which is fixed to 7 as we previously mentioned. The mutation rate is fixed to 1% in all runs.

A hundred different 3–SAT problems were uniformly generated. Since the phase transition of a 3–SAT problem occurs at a ratio of 4.3 between the number of clauses and the number of literals, each of the hundred problems contained a hundred literals and 430 clauses to maintain the ratio of 4.3. Therefore, all problems are hard and there is no guarantee that a solution exists.

In the following two sections, MBO–GSAT and MBO–RWK are compared against GSAT and random walk respectively. In our implementation, both versions of GSAT and random walk are the same as the one implemented within MBO; therefore, implementation efficiencies are fixed to have a fair comparison.

5.2 Results and comparisons: MBO–GSAT

In this section, we first scrutinize the behavior of MBO–GSAT then we compare the results with GSAT itself.

Table 1: *The average number of unsatisfied clauses by the queen's genotype and the total number of solutions found according to each colony size for MBO–GSAT.*

Number of broods	Percentage of unsatisfied clauses	Total number of solutions found
20	1.6 ± 1.0	17
40	1.8 ± 1.0	15
60	2.0 ± 1.1	10
80	2.0 ± 1.2	8
100	2.2 ± 1.1	7

In Table 1, the average number of unsatisfied clauses by the queen's genotype at the end of each run and the total number of solutions found are presented, grouped by the colony size. As we can see from the table, the results are shown for each number of broods without showing the amount of time devoted for brood care. The reason for this is that the results were the same, regardless of the amount of time devoted for brood care. This is very interesting and may suggest that the improvement in the algorithm's performance is independent of the time devoted for brood care. We cannot generalize this as we expect that it is problem dependent and it also depends on the used heuristic. This will be re–visited in the following section.

An important point in SAT is to identify the best assignment which minimizes the number of unsatisfied clauses. Some real–life problems (such as in planning and scheduling) are not satisfiable (sometimes called over–constrained problems), so returning a good assignment that minimizes the number of unsatisfied clauses is indispensable. In Table 1, we present the average number of unsatisfied clauses by the queen's genotype in the last mating–flight. It is not necessary that the last mating–flight is the maximum number of mating–flights because the algorithm terminates also when an assignment that satisfies all clauses is found. The least number of unsatisfied clauses is found with the smallest number of broods.

Once more, the highest number of solutions found corresponds to the smallest number of broods. This may indicate that a smaller number of broods is better in our test cases. This result is very interesting because the behavior of the model on SAT is somehow consistent with the underlying biological model. A monogynous honey–bees' colony usually contains small number of broods, although the definition of "small" may vary between the biological and computational model. Moreover, since we assume that the drones are unrelated to the queen, the amount of inbreeding is small; therefore, a small colony would still maintain genetic diversity.

To find out whether GSAT alone would have achieved the best results overall without the additional overhead of MBO–GSAT, we solved the hundred problems with the same version of GSAT that we used in our implementation. The number of trial solutions is set to 2400 (the number of mating–flights times the number of broods per flight). Three different search sizes or numbers of flips of 100, 200, and 300, are used so that they are consistent with the three times devoted for brood care in our algorithm.

Table 2: *The average number of unsatisfied clauses and the total number of solutions found according to each search size for GSAT.*

Search size	Percentage of unsatisfied clauses	Total number of solutions found
100	7.1 ± 2.1	0
200	4.7 ± 1.8	1
300	3.6 ± 1.8	5

In Table 2, we present the total number of solutions found by GSAT, along with the average number of unsatisfied clauses for the hundred problems. Similar to MBO–GSAT, GSAT terminates when a solution is found or when the maximum number of trials is reached.

Before discussing the results of Table 2, it is worth mentioning that the only real difference between MBO–GSAT and GSAT is in the means trial solutions are generated. In GSAT, trial solutions are generated at random, whereas in MBO–GSAT they are generated after the mating–flight using haploid–crossover, as discussed in Section 4.2, and mutation.

The average number of unsatisfied clauses found by GSAT ranged from 3.6 to 7.1. This value is much higher than the worst value of 2.2 found by MBO in Table 1. Moreover, the number of solutions found by GSAT, as shown in Table 2, is much lower than the number of solutions found by similar search sizes using MBO–GSAT. These results emphasize the importance of the bias introduced by the mating–flight in generating trial solutions. Centering the trial–solutions on the queen (by mating the queen with the broods) proved to be better than re–initializing GSAT at random.

5.3 Results and comparisons: MBO–RWK

Similar to the previous section, we first scrutinize the behavior of MBO–RWK then we compare the results against random walk.

In Table 3, we present the average number of unsatisfied clauses by the queen's genotype at the end of each run

Table 3: *The average number of unsatisfied clauses by the queen's genotype for MBO–RWK.*

Number of broods	Search size		
	100	200	300
20	4.0 ± 2.0	3.2 ± 1.8	2.9 ± 1.8
40	4.4 ± 2.1	3.2 ± 1.8	2.9 ± 1.9
60	4.4 ± 2.2	2.9 ± 1.9	2.7 ± 1.7
70	4.3 ± 2.0	3.1 ± 1.8	2.8 ± 1.8
100	4.4 ± 2.0	3.0 ± 1.9	2.8 ± 1.7

grouped by the colony size and amount of time devoted for brood care. On the contrary to MBO–GSAT, the amount of time devoted for brood care influenced the performance of the algorithm. As it can be seen in the Table, the more time devoted for brood care, the better the performance. This was not the case in MBO–GSAT and therefore it is hard to generalize. The behavior of the algorithm is highly influenced with the used heuristic. The best performance obtained with a colony size of 60 and amount of time devoted for brood care of 300.

Table 4: *The total number of solutions found for MBO–RWK.*

Number of broods	Search size		
	100	200	300
20	5	7	8
40	4	8	10
60	4	15	10
80	3	11	12
100	1	8	11

In Table 4, the total number of solutions found are grouped by the colony size and amount of time devoted for brood. Once more, the highest number of solutions found corresponds to 60 broods. However, this best performance is obtained with search size of 200.

Similar to the comparison between MBO–GSAT and GSAT, we compare the performance of MBO–RWK and random walk. We solved the hundred problems with the same version of random walk that we used in our implementation. Similar to GSAT, the number of trial solutions is set to 2400 (the number of mating–flights times the number of broods per flight) and the search size or numbers of flips are 100, 200, and 300.

The performance of random walk is found consistent, regardless of the search size. The average number of unsatisfied clauses is 2.2 ± 0.9 and the total number of solutions found is 3 for the three search sizes. Comparing this result with the performance of MBO–RWK, we find that random walk is more successful in satisfying the clauses, although not as successful in finding a solution. MBO–RWK, on the other hand, is more successful in finding solutions than random walk alone.

5.4 A summary of the results

A general trend in both versions of MBO is that MBO improved the performance of both heuristics when the latter are used alone. MBO–GSAT and MBO–RWK, both found much more solutions than GSAT and random walk alone. MBO-GSAT also found better assignment than GSAT, although this was not the case for MBO-RWK compared to random walk. In summary, MBO-GSAT performed the best among the other three (GSAT, MBO-RWK, and random walk), in terms of both the number of solutions found and the number of unsatisfied clauses achieved by each algorithm.

To summarize, the annealing stage undertaken by the queen during her mating–flight in conjunction with the use of haploid–crossover and mutation, proved useful in the set of SAT problems solved here. In addition, a small to average number of broods produced better results in our test examples.

6 Conclusion

In this paper, a new heuristic, MBO, based on the marriage process in honey–bees was introduced. The biological motivation and the computational aspects of the algorithm were both discussed. From the analysis of the experimental results, MBO was very successful on a group of one–hundred hard 3–SAT problems. The main advantage from our perspective was that the algorithm preserved many of the underlying biological concepts and achieved the good performance with parameters taken from real biological concepts. Moreover, it was shown that MBO–GSAT performed better than GSAT alone. Also, MBO–RWK found more solutions than random walk. For future work, more analysis on the behavior of MBO is required. Also, applying MBO to a wide range of problems is important to elicit those where MBO could be useful.

References

[1] J. Adams, E.D. Rothman, W.E. Kerr, and Z.L. Paulino. Estimation of the number of sex alleles and queen matings from diploid male frequencies in a population of apis mellifera. *Genetics*, 86:583–596, 1972.

[2] E. Bonabeau, M. Dorigo, and G. Theraulaz. *Swarm intelligence: from natural to artificial systems*. Oxford Press, 1999.

[3] S.A. Cook and D.C. Mitchell. Finding hard instances of the satisfiability problem: A survey. In DIMACS Series in Discrete Mathematics and Theoretical Computer Science, editors, *Satisfiability Problem: Theory and Applications*. American Mathematical Society, 1997.

[4] A. Dietz. Evolution. In T.E. Rinderer, editor, *Bee genetics and breeding*, pages 3–22. Academic Press, Inc, 1986.

[5] M. Dorigo, E. Bonabeau, and G. Theraulaz. Ant algorithms and stigmergy. *Future Generation Computer Systems*, 16:851–871, 2000.

[6] I.P. Gent and T. Walsh. The satisfiability constraint gap. Technical Report 702, University of Edinburgh, 1994.

[7] H.H. Hoos. On the run-time behaviour of stochastic local search algorithms for sat. *Proceedings of AAAI*, pages 661–666, 1999.

[8] H.H. Hoos and T. St*ü*tzle. Local search algorithms for sat: An empirical evaluation. *Journal of Automated Reasoning*, 24:421–481, 2000.

[9] H.H. Laidlaw and R.E. Page. Mating designs. In T.E. Rinderer, editor, *Bee Genetics and Breeding*, pages 323–341. Academic Press, Inc, 1986.

[10] R.E. Page, R.B. Kimsey, and H.H. Laidlaw. Migration and dispersal of spermatozoa in spermathecae of queen honey bees: Apis mellifera. *Experientia*, 40:182–184, 1984.

[11] C.H. Papadimitriou and K. Steiglitz. *Combinatorial optimization: algorithms and complexity*. Prentice-Hall, 1982.

[12] T.E. Rinderer and A.M. Collins. Behavioral genetics. In T.E. Rinderer, editor, *Bee Genetics and Breeding*, pages 155–176. Academic Press, Inc, 1986.

[13] B. Selman and H. Kautz. An empirical study of greedy local search for satisfiability testing. *Proceedings of AAAI*, 1993.

[14] B. Selman, H. Kautz, and B. Cohen. Noise strategies for improving local search. *AAAI94*, pages 337–343, 1994.

[15] B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. *Proceedings of AAAI*, pages 440–446, 1992.

[16] H. Zhang and M. Stickel. Implementing davis-putnam's method by tries. Technical report, The University of Iowa, 1994.

Credit Assignment Method for Learning Effective Stochastic Policies in Uncertain Domains

Sachiyo Arai and Katia Sycara
The Robotics Institute, Carnegie Mellon University
5000 Forbes Avenue, Pittsburgh, PA 15213 USA
E-Mail: {sachiyo, katia}@cs.cmu.edu

Abstract

In this paper, we introduce *FirstVisit Profit-Sharing* (FVPS) as a credit assignment procedure, an important issue in classifier systems and reinforcement learning frameworks. FVPS reinforces effective rules to make an agent acquire stochastic policies that cause it to behave very robustly within uncertain domains, without pre-defined knowledge or subgoals. We use an internal episodic memory, not only to identify perceptual aliasing states but also to discard looping behavior and to acquire effective stochastic policies to escape perceptual deceptive states.

We demonstrate the effectiveness of our method in some typical classes of Partially Observable Markov Decision Processes, comparing with Sarsa(λ) using a replacing eligibility trace. We claim that this approach results in an effective stochastic or deterministic policy which is appropriate for the environment.

1 Introduction

In this paper, we present the learning algorithm to address the *credit assignment procedure*, which is a very important issue in both classifier systems and reinforcement learning frameworks [Holland, 1986][Moriarty et al., 1999]. We focus on the environments that have goals to be attained by autonomous agents that learn with delayed reward. This problem can be called a sequential decision problem, which is defined by a set of an agent's sensory observations and a set of actions that map observations to successor observations. If an agent's sensory observations provide the complete state of its environment, the environment can be formulated as Markov decision processes (MDPs), for which a number of very successful planning[Barto et al., 1995] and reinforcement learning[Watkins & Dayan, 1992] approaches have been developed.

However, in many real environments, such as multi-agent and distributed control mobile robotics' environments, only partial information about the state-spaces can be expected. These environments can be formulated as partially observable Markov decision processes (POMDPs) where agents suffer from hidden states or perceptual aliasing (i.e., the agent takes some different environmental states as the same sensory observation). Therefore, finding an efficient method for solving POMDPs would be a very practical contribution to creating adaptive agents.

Recent approaches in POMDPs can be classified into two types. One is called a memory-based approach[Chrisman, 1992][McCallum, 1995], which attempts to overcome perceptual aliasing by using memory to estimate real-state and finally to find deterministic policies for the environment. The other is called a memory-less approach[Jaakkola et al., 1994] [Loch & Singh, 1998][Singh et al., 1994], which can acquire a stochastic policy to make the agent robust against perceptual aliasing. Recently, there also is an intermediate approach[Lanzi, 2000] which introduces small internal memories, not to construct a model of the environment but to retain effective classifiers by combining these memories with a genetic algorithm.

Sarsa(λ) using a replacing eligibility trace, proposed in [Singh & Sutton, 1996], is highly regarded in [Peshkin et al., 1999][Lanzi, 2000][Loch & Singh, 1998] as a memory-less approach which can work very well in POMDPs. Unfortunately, their results refer to different testbeds, so we cannot see what the resolvable sub-class of POMDPs is by this method.

In this paper, we abstract three interesting subclasses

of POMDPs which bring agents serious confusions and propose a *FirstVisit Profit-Sharing* (FVPS) approach to make agents behave robustly in these confusions. We show the performance of our approach by comparison with Sarsa(λ) in an *episodic task* where there is a goal within a finite number of steps from every initial state. The episodic task is common in the real world, where we can define a desirable situation as a goal but cannot define the subgoals.

FVPS is classified as a memory-less approach. Although we use an internal one-dimensional episodic memory, this memory is not used to construct the state-transition model but only to discard looping behavior and to acquire an effective stochastic policy to escape perceptual deceptive states. Therefore, FVPS does not cost much computation and memory space. It is very similar to the Monte-Carlo approach and Sarsa(1) in that it makes no attempt at satisfying Bellman equations relating the values of successive states. It is different from Monte-Carlo in that the weight[1] of rules acquired by our method has no meaning as the estimation of state-action values, whereas Monte-Carlo attempts to estimate the value of the state (or state-action pair in Sarsa(λ)) as an averaged reward. These properties of FVPS not only make an agent behave robustly against perceptual aliasing but also save memory and computation costs by finding and retaining only rules essential for surviving in the environment.

In Section 2, we describe our domain, notations, and related algorithms. Section 3 introduces our approach, FVPS. An empirical comparison of performance using two learning approaches, FVPS and Sarsa(λ=0, 0.9, 1), is presented via several experiments in Section 4, and we discuss the applicability and effectiveness of our approach for the real world in Section 5. Finally, we conclude and summarize our future work.

2 Problem Domain

2.1 Agent Model

The agent is modeled as a reinforcement learning entity engaged in an episodic task in an unknown environment, where there are no intermediate subgoals for which intermediate rewards can be given. (Note: Because our focus here is on *credit assignment*, the agent does not have any genetic algorithm framework.) An environment is defined by a finite set of state S, the agent has a finite set of actions A, and the agent's sensors provide its observations from a finite set X. Each agent consists of five modules: a *State Recognizer*, a *LookUp Table*, an *Action Selector*, an *Episodic Memory* and the *Learner*, which includes the credit assignment procedure, as shown in Fig.1.

[1]Weight here is similar in meaning to *Value* in the DP-based approaches, and *Strength* in the classifier systems.

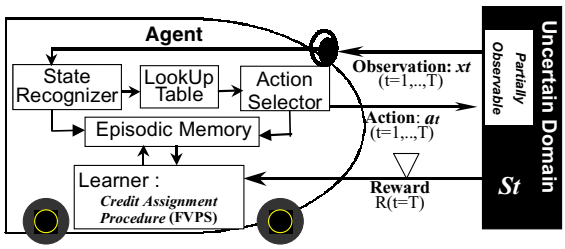


Figure 1: Agent Model

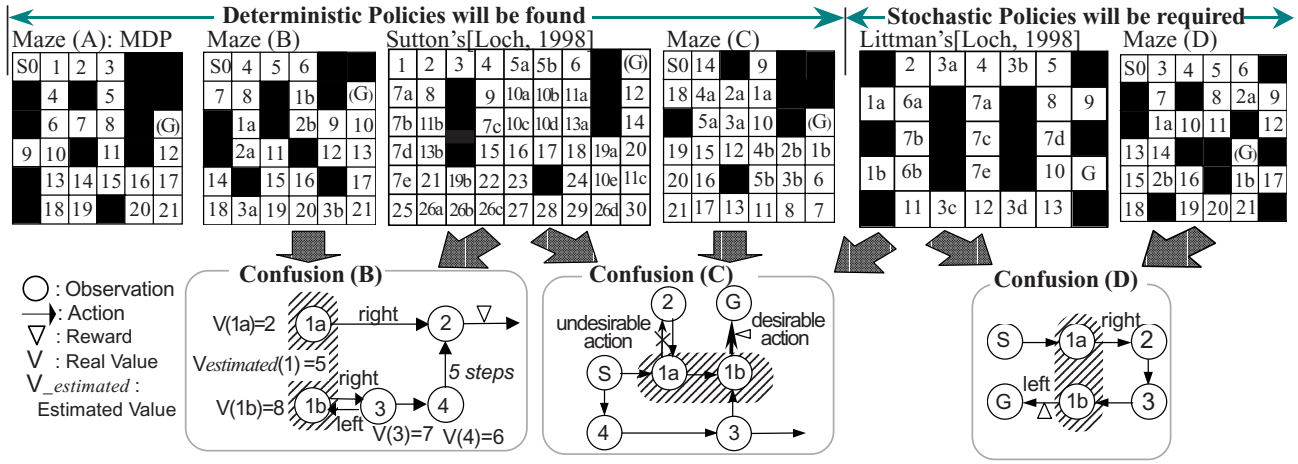
Initially, the agent observes x_t as s_t, the partially available state of its environment at time t. An action is then selected (using a certain exploration method) from the action set A_t, which contains all the available actions at time t. If there is no reward after action a_t, the agent stacks the observation-action pair, (x_t, a_t), into its *Episodic Memory*, and repeats this cycle until a reward is generated. The observation-action pair is called a *rule* in this paper. The process of moving from a start state to the final reward state is called an *episode*.

2.2 Target classes of POMDPs

We take five mazes, including two test problems which are treated in [Loch & Singh, 1998], as shown in Fig.2, to show the typical confusions which cause improper behavior of the agent. Besides the mazes we treat here, the *load-unload problem*[Peshkin et al., 1999] and *Woods101, Maze7, Woods101$\frac{1}{2}$, and Woods102* [Lanzi & Wilson, 2000], which includes confusion type (D), have been used in researching POMDPs.

Except for maze (A), they include some perceptual aliasing areas (such as 1a, 1b,..and 26d), which make the agent fall into the confusion types (B), (C), and (D). For mazes (B) and Sutton's, a common action can be effective among the same observations, and for maze (C), there exists another reliable route which does not include the aliasing area, so a stochastic policy is not required. Therefore, in mazes of (B), (C) and Sutton's, a deterministic policy can be found in each observation, regardless of existing confusions.

However, in maze (D), which includes confusion type (D), the agent cannot achieve the goal with only



For Types A-D and Sutton's, the agent can observe its eight neighboring cells and (G) indicates the goal position of each environment, but the agent cannot observe it as a goal position. In Littman's environment, the agent can observe its four directions: up, right,down and left; if G is in one of these four directions, it can observe G as a goal. Maze (A), (B) and (C) have been reproduced from [Miyazaki and Kobayashi 1999].

Figure 2: Target subclasses of POMDPs and their Confusion Types

a deterministic policy. For example, if the agent is in state $1a$, *down* is a desirable action, but if it is in state $1b$, it needs to move *up* to reach the goal. Also, stochastic policies are necessary in Littman's maze[Loch & Singh, 1998], because the observations are noisy, with the agent getting the correct observation only 70% of the time.

2.3 Related Algorithms

There are two types of credit assignment procedures. One is inspired by dynamic programming (e.g., [Watkins & Dayan, 1992]). The other is inspired by Holland's learning classifier systems (e.g., [Grefenstette, 1988]). The former one basically attempts to satisfy Bellman equations relating the values of successive states (or state-action pairs) to make an agent behave optimally. The latter one does not attempt to estimate the value of all rules that cover the state space, but just accumulates the weight on successful rules based on the agent's experiences. These credit assignment procedures are called the *boot-strapped* method and the *non-boot-strapped* method, respectively. The equivalence between *bucket brigade algorithm*, used in the classifier system[Holland, 1986], and Q-learning is proved in [Dorigo & Bersini, 1994], so we can't say that all classifier systems apply the *non-boot strapped* method.

Q-learning by [Watkins & Dayan, 1992](Eq.1) computes by successive approximations a table of all values $Q(x,a)$. At each time step in the episode n the agent updates $Q_n(x_t,a_t)$ by recursively discounting future utilities and weighting them by positive learning rate α. Here $\gamma(0 < \gamma < 1)$ is a discount parameter. If there is no immediate reward r, the agent uses $r = 0$ to update $Q_n(x_t,a_t)$.

Q-learning often performs poorly in POMDPs due to computing by successive approximations. Sarsa($\lambda = 0$) also updates the value of the state using successive state values, as shown in Eq.2. However, [Loch & Singh, 1998] demonstrated that Sarsa using a replacing eligibility trace (as shown in Eq.3) with a large λ value (such as $\lambda > 0.9$) performs well in some classes of POMDPs.

Q-learning:

$$Q_{n+1}(x_t,a_t) \leftarrow (1-\alpha)\cdot Q_n(x_t,a_t) + \alpha(r + \gamma max_{b \in actions} Q_n(x_{t+1},b)) \quad (1)$$

Sarsa(λ):

$$Q_{n+1}(x_t,a_t) \leftarrow (1-\alpha)\cdot Q_n(x_t,a_t) + \alpha(r + \gamma Q_n(x_{t+1},a_{t+1})) \quad (2)$$

Replacing Eligibility Trace and Sarsa(λ):

$$\begin{aligned} &1. \quad \delta t \leftarrow r_t + \gamma Q_n(x_{t+1},a_{t+1}) - Q_n(x_t,at) \\ &2. \quad \eta_t(x_t,a_t) = 1 \\ &3. \quad \forall(x \neq x_t, a \neq a_t); \eta_t(x,a) = \gamma\lambda\eta_{t-1}(x.a) \quad (3) \\ &4. \quad \forall(x,a); Q_{n+1}(x,a) \leftarrow Q_n(x,a) + \alpha\cdot\delta_t\cdot\eta(x,a) \end{aligned}$$

The eligibility traces are initialized to zero, and in episodic tasks they are reinitialized to zero after every episode. When $\lambda = 1$ in Sarsa, it is the same as the Monte-Carlo method[Singh & Sutton, 1996], which does not attempt to satisfy Bellman equations

relating the values of successive states. It seems that using a successive value for state-value estimation is not effective for POMDPs. Here, we should note that Sarsa($\lambda > 0$) requires computation time to update the whole table of experienced rules, or in the most serious case, must update all state spaces at each step.

Profit-Sharing by [Grefenstette, 1988], used in the classifier system, provides a hopeful *non-boot strapped* credit assignment method. Profit-Sharing is very similar to the Monte-Carlo method in that it does not utilize successive approximations to compute a table of all state values. This is an important property which can make Profit-Sharing attractive when one requires the value of only a subset of the state.

3 Our Approach

3.1 Requirements for Acquiring a Proper Policy

There are three requirements in the algorithm to make an agent behave robustly in uncertain domains. The first is that the algorithm needs to update the weight of the state independently without using successive state values. The second is that the algorithm should not attempt to estimate the value function. This will fail in POMDPs in which the agent's sensory input is limited. The value estimation, mapping one value to one rule, makes no sense when there is no unique value to an observation. The accumulation of the weight on successful rules is enough to make policies proper. The third requirement is that the algorithm must guarantee that the agent will reach the goal within a finite number of steps.

Our credit assignment approach is based on Profit-Sharing[Grefenstette, 1988]. The good properties which FVPS inherits from Profit-Sharing satisfy the first and second requirements mentioned above. However, the Profit-Sharing approach does not take the infinite loops in the agent's episode explicitly into consideration because the Profit-Sharing is assumed to use in combination with a genetic algorithm which will get rid of any bad behavior.

FVPS improves on Profit-Sharing in that it guarantees that loops are discarded without evolutionary pressure. In much reinforcement learning research, the target problems do not contain loops (e.g., board games), although there are some problems which do contain loops[Hansen, 1998]. These loops may result in the agent exhibiting improper behavior with respect to achieving its goal. In general, it is important to pursue *proper* policy rather than *optimal* for POMDPs. A *proper* policy is one that is guaranteed to converge on a solution; i.e., the agent should not become trapped within infinite loops in the state machine. To guarantee convergence on a *proper* policy in POMDPs, we introduce the *FirstVisit* routine and credit assignment function.

To illustrate the advantage of this point, consider the example in Fig. 2 Confusion (B). The state value, V, represents the minimum number of steps to a reward. In this example, the highest value of V is 1. The values of states 1a and 1b, V(1a) and V(1b), are 2 and 8, respectively. Although these two states are different, they are perceived by the agent as being the same state (i.e., state 1). If the agent moves to state 1a and 1b with equal weight, $\mathsf{V}(1) = \frac{2+8}{2} = 5$. Therefore the value of state 1 is equal to the value of state 3, i.e., $\mathsf{V}(3) = 5$. If the agent uses these state values according to a DP-based algorithm, such as Q-learning (Eq. 1), it will move *left* into state 3. Otherwise, the agent moves *right* into state 1. This means that the agent learns the improper policy in which it only transits between states 1b and 3. If the agent only reinforces the successful rules without any propagation to other rules, it can escape this looping behavior caused by confusion type (B).

3.2 FirstVisit Profit-Sharing

Our solution to this problem is very simple. If the current observation is the revisited one and the same action is executed, the agent does not stack this rule into the *Episodic Memory*, since the re-executed rule will cause a cyclic loop in the agent's route. This routine does not require any extra memory other than that used by the current framework of the Profit-Sharing algorithm, where an *Episodic Memory* is used to accumulate rules until the goal is achieved.

Fig. 3 shows our algorithm. The *FirstVisit* routine prohibits the agent from reinforcing the weight of the rules which make up the loop, and can retain essential rules for a stochastic policy. Consider the episode illustrated in Fig. 4(1), which is one example of a generated decision sequence in the initial stage of learning, in the maze (C) shown in Fig. 2(C). In this episode, the sub-sequence $(1a, Up) - (9, Down) .. - (9, Down)$ forms an obviously needless loop; therefore the weight of $(1a, Up)$ should not be reinforced, while $(1b, Up)$ is necessary to reach the goal. However, the agent cannot tell the difference between $1a$ and $1b$, so we need to consider which rule should be retained and which rule should be removed from the episodic memory.

In FVPS, if the agent finds the same rule as the ex-

begin
Initialize $W(x,a)$ arbitrarily and TotalSteps=0.
  Repeat (for each episode n){
    do {
      - get sensory input x_t of the environment ;
      - choose a_t from an available action set A_t ;
      - take action a_t ;
      - *FirstVisit Routine (episodicMemory[], currentStackSize, (x_t, a_t))* ;
      - check reward r_t ;
      - TotalSteps ++ ;
      } while (reward ! = 0) ;
    if $r_t = R(>0)$ at time T steps then, T=TotalSteps;
      $\forall(x,a)$ in the episodic memory

$$W_{n+1}(x,a) \leftarrow W_n(x,a) + R \cdot \beta^T \qquad (4)$$

  } until enough number of episodes.
end

FirstVisitRoutine(episodicMemory[], currentStackSize,(x_t, a_t)):
begin
    initialize pointer pt=0,
      do from the first rule of episodicMemory[]{
        search same rule as (x_t, a_t).
          (compare (x_t, a_t) with stacked rule$(x,a)_{pt}$ of episodicMemory[]; pt++;)
          if (x_t, a_t)== episodicMemory[pt=k], break;
      } while ($pt == currentStackSize$);
      if found the same rule in the episodicMemory
          ($pt < currentStackSize$) {
          retain whole rules of episodic memory,
            do not stack executed rule, (x_t, a_t).
      }
      else {
          stack executed rule, (x_t, a_t) into episodicMemory[].
          currentStackSize ++;
      }
end

Figure 3: FVPS algorithm

ecuted rule (x_t, a_t) in the episodic memory, it does not stack to the episodic memory. Finally, each executed rule appears only once in the episodic memory as shown in Fig. 4(2). Even in this method, assignments will be done on needless rules, such as $(10, Up)$ in maze (C). But FVPS can eliminate these needless rules by our reinforcement function, $R \cdot \beta^{TotalSteps} (0 < \beta < 1)$, in which the value is small when the length of the agent's route is long (i.e., the size of *TotalSteps* is large). Therefore, the assignment quantity on needless rules (e.g., $(10, UP)$ in maze (C)) will be smaller than the one on effective rules (e.g., $(10, Down)$) that make the length of the agent's route short, thereby causing only the effective rules to be retained. On the other hand, because both $(1, Up)$ and $(1, Down)$ in maze (C) are necessary to reach the goal, both rules are retained

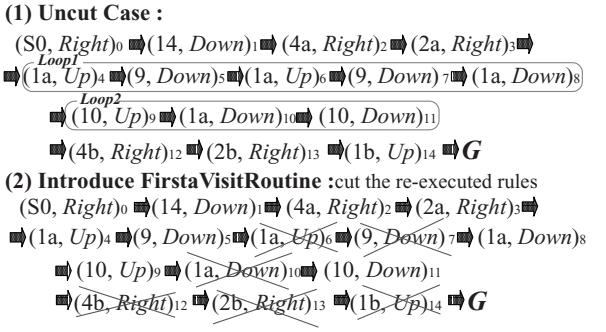


Figure 4: Example: How to Discard Looping Behavior in Maze (C)

as effective rules. Also, if there exist shorter routes to the goal without stochastic policies, the agent exploits these deterministic policies and reinforcement of the rules for the perceptual aliasing area is precluded. (We describe this using a concrete example in Section 4.2.) We claim that this approach brings about the effective stochastic or deterministic policy which is appropriate for the environment.

FVPS is very similar to the *First-visited* Monte-Carlo method[Singh & Sutton, 1996], where the assignment will be given only to the first visited state. However, the values in the Monte-Carlo method are estimated as sample averages of observed reward using the Widrow-Hoff rule (Eq. 5)[2], whereas FVPS just piles weight on successful rules according to trial and error experiences, as shown in Eq.4. It is important that the averages of observed reward among the aliasing states will fluctuate from episode to episode, and will make no sense as the value of the observation. One the other hand, piling up weights on successful rules is simple and makes agents robust in POMDPs as well.

$$\begin{aligned} \text{NewEstimate} &\Leftarrow \text{OldEstimate} \\ &+ \ \textit{StepSize}[\text{Target-OldEstimate}] \end{aligned} \qquad (5)$$

4 Experiments

4.1 Settings

To demonstrate the effectiveness of the FVPS approach, we compared its performance with that of the Sarsa(λ) algorithm on the MDP(maze (A)) and five POMDP problems. (Two of them are taken from [Loch & Singh, 1998].) We describe aspects of the empirical results here. In the cases of maze (A) to maze (D), the agent starts from state S_0, as shown in

[2] *StepSize* is represented by α in Eq.1 and Eq.2.

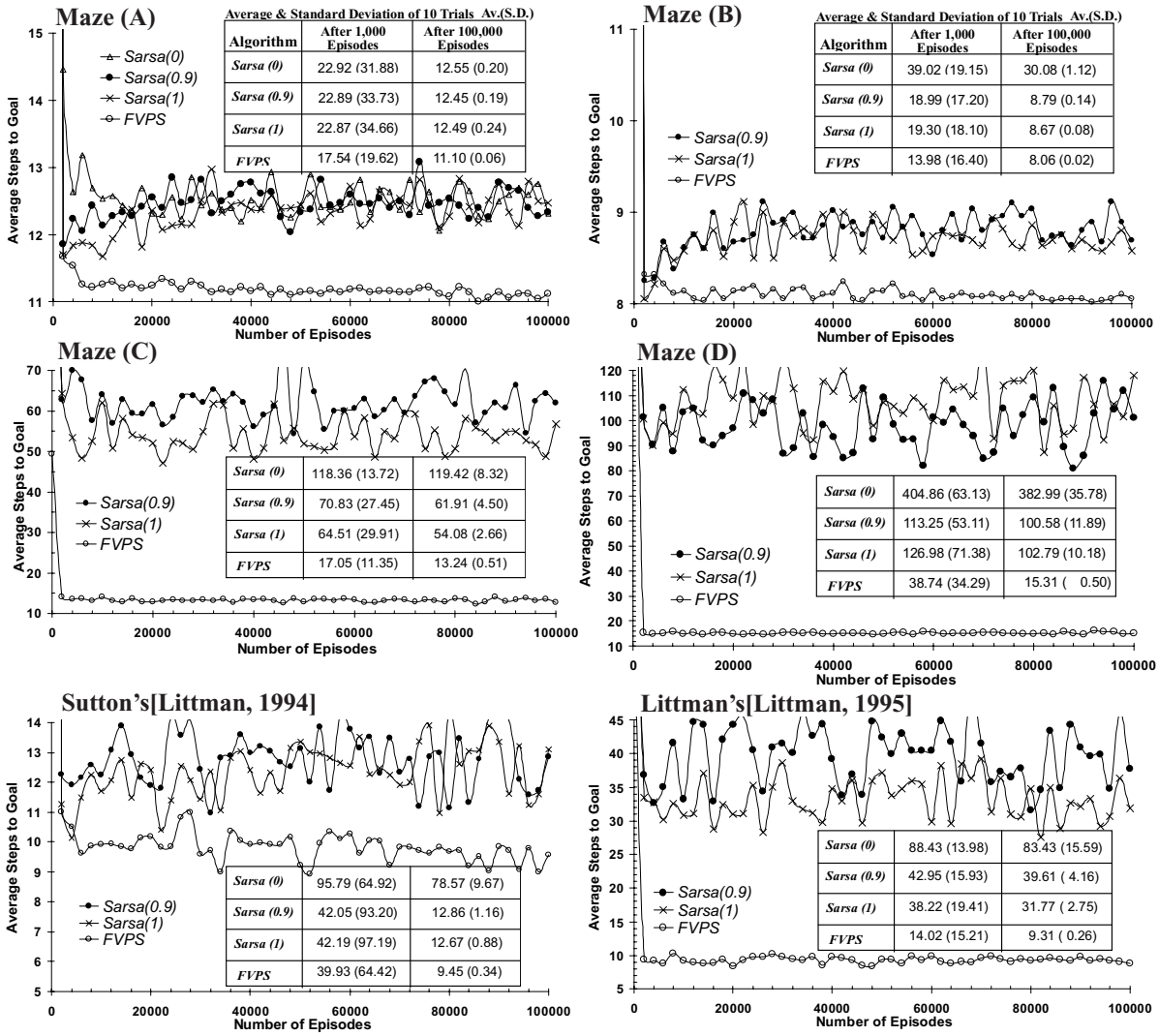


Figure 5: Comparisons among FVPS and Sarsa(λ)

Fig. 2, while in the cases of Sutton's and Littman's, it starts from different locations. At each time step, the FVPS agent selects an action by a roulette wheel method, where the rate of each action is $p(a_i|x) = \frac{W(x,a_i)}{\sum_{a_k \in A_t} W(x,a_k)}$, while the Sarsa agent uses the Boltzmann distribution $p(a_i|x) = \frac{e^{Q(x,a_i)/T}}{\sum_{a_k \in A_t} e^{Q(x,a_k)/T}}$ ($T = 0.2$) to select its action. There are four actions within the action set, $A_t = \{Up, Right, Down, Left\}$, except in Littman's case. Littman's permits an additional action, $\{Stay\}$. The weight (or value) of the rules was initialized to 10.0 in FVPS and 0.0 in Sarsa. The reward 1000.0 in FVPS and 1.0 in Sarsa is given after achieving the goal. In the Sarsa case, 0.0 is used to update at each time step. In all but Littman's case, the agent can observe its eight neighboring cells and it cannot see the goal as an entity, which means that the agent gets the reward when it is in the goal position. In Littman's case, the agent can observe the relative four directions: front, back, left and right, and can see the goal if the goal is in its relative four directions, although the observations are noisy, with the agent getting the correct observation only 70% of the time.

The parameters R, β, and T of FVPS are used to update the weight of each rule in the episodic memory. In our experiments, the rules which are retained after the *FirstVisit* routine will be given the value $R \cdot \beta^T = (1000.0) \cdot (0.8)^{TotalSteps}$. The parameters α, λ, and γ of Sarsa are also used to update the Q-value of each rule. In our experiments, both the step-size α and the λ values are held constant in each experiment. We followed Loch[Loch & Singh, 1998] to select

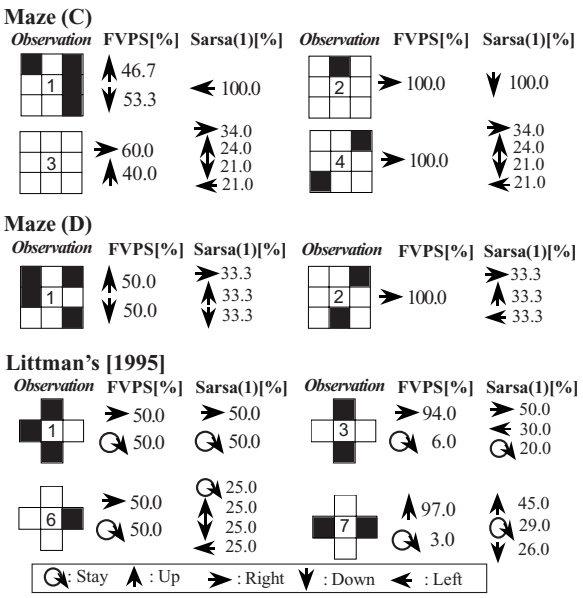


Figure 6: Policies of FVPS and Sarsa(λ) in Uncertain Areas

these values: $\alpha = 0.01$, $\lambda = 0.9$ and 1.0. We searched over γ value for these problems and selected $\gamma = 0.8$, which gave the best performance across all problems. The evaluation metric is determined by averaging the number of steps to reach the goal. Experiments consist of 10 trials, each of which consists of 100,000 episodes. The lookup table is reset for each trial.

4.2 Results

After every episode, the policy (which is selected by a roulette method in FVPS and by the Boltzmann distribution in Sarsa as mentioned in Section 4.1) was evaluated and the learning curves of all types of mazes shown in Fig. 2 plotted as shown in Fig. 5. The x-axis shows the number of episodes and the y-axis shows the average steps to the goal of 10 trials.

The maze (A) is an MDP, where Q-learning and any other DP-based algorithms can reach the optimal policy of 11 steps. Although Sarsa also can reach the optimal theoretically, it seems that step-size parameter α needs to be adjusted to reach it, because Sarsa($\lambda = 0, 0.9, 1$) starts to approach the optimal but then gets farther away from it as episodes are repeated. FVPS acquired the optimal policy only with the $FirstVisit$ routine and our reinforcement function, $R \cdot \beta^{TotalSteps}$.

The maze (B) includes confusion type (B), where Q-learning and Sarsa(0) are no longer useful because of their value estimation method as described in 3.1. FVPS and Sarsa($\lambda = 0.9, 1$) could reach optimal policy here, although parameter adjustment seems to be required for Sarsa.

The maze (C) includes confusion type (C), where there are two routes to reach the goal. One route consists of 9 steps in total, $S0 - 14 - 4a - 2a - 1a - 10 - 4b - 2b - 1b - G$, and the other consists of 11 steps in total, $S0 - 14 - 4a - 5a - 3a - 10 - 4b - 5b - 3b - 6 - 1b - G$. The former route is the optimal one, but to realize it, the agent must select $Down$ in $1a$ and Up in $1b$, whereas in the latter one the agent does not need a stochastic policy. Sarsa($\lambda = 0.9, 1$) dropped down here, because its value estimation failed even though it uses replacing eligibility traces when one observation requires different actions. As shown in Fig. 6, FVPS reached the stochastic policy in observations 1 and 3, and in 2 and 4, the agent acquired the deterministic policy. That is, FVPS acquired the effective policy that the agent needs.

The maze (D) includes confusion type (D), where there is no route to reach the goal with only a deterministic policy and where a stochastic policy is required. The Littman's maze also has this confusion because of the noise with the agent's observation. In these environments, FVPS works best and can reach nearly optimal policy within the agent's perceptual ability.

5 Discussion

The results demonstrate that FVPS found the proper policies even in the POMDPs where the currently evaluated Sarsa(λ) does not show good results. FVPS performs much better than Sarsa in the environment where more than one action will be reinforced due to the aliasing, such as mazes of (C), (D) and Littman's. In these environments, the update method of Sarsa(λ), in which the values are estimated as sample averages of observed reward by Eq.5, seems not to work well, because there is no unique value to be estimated as the rule value. FVPS, however, just piles the weight on successful rules according to the agent's trial and error experiences. This simple method seems to work very effectively in such environments.

We claim that FVPS requires an episodic memory of moderate size. Only rules which are retained in the episodic memory would be updated after each episode. Sarsa($\lambda > 0$), on the other hand, requires memory to keep eligibility traces and computation time to update all rules at each time step. The size of state spaces of environments such as those treated in this paper is very small, so the computation time for updating is not substantial. But the larger the state space, such

as in the multi-agent learning environment, the more serious a problem it might be in practical use.

6 Conclusion

In this paper, we introduce FVPS, a variant of the Profit-Sharing algorithm, and demonstrate its effectiveness within some interesting subclasses of POMDPs and its minimal memory requirements. We believe that our method will be easily introduced into a classifier system, and can solve over many subclasses of POMDPs by combining with a certain genetic algorithm. In future work, we will prove the effectiveness of FVPS theoretically and show the powerful results on more difficult classes.

Acknowledgement
This research has been sponsored in part by ONR grant N-00014-96-16-1-1222 by DARPA grant F-30602-98-2-0138.

References

[Barto et al., 1995] Barto, A.G., Bradtke, S.J., and Singh, S.P. Learn to Act using Real-Time Dynamic Programming. *Artificial Intelligence*, Vol.72, Number 1-2, pages 81-138, 1995.

[Chrisman, 1992] Chrisman, L. Reinforcement learning with perceptual aliasing: The Perceptual Distinctions Approach. *Proceedings of the 10th National Conference on Artificial Intelligence*, pages 183-188, 1992.

[Dorigo & Bersini, 1994] Dorigo, M. and Bersini, H. A Comparison of Q-learning And Classifier Systems. *Proceedings of the 3rd International Conference on Simulation of Adaptive Behavior*, pages 248-255 1994.

[Grefenstette, 1988] Grefenstette J. J. Credit Assignment in Rule Discovery Systems Based on Genetic Algorithms, *Machine Learning*, Vol.3, pages 225-245, 1988.

[Hansen, 1998] Hansen, E.A. Solving POMDPs by searching in Policy Space. *Proceedings of 14th International Conference on Uncertain Artificial Intelligence*, 1998.

[Holland, 1986] Holland, J. H. Escaping Brittleness: The Possibilities of General-Purpose Learning Algorithms Applied to Parallel Rule-Based Sysems. In R.S.Michalsky et al. (eds.), *Machine Learning: An Artificial Intelligence Approach, Vol.2*, pages 593-623, Morgan Kaufman 1986.

[Jaakkola et al., 1994] Jaakkola, T., Singh,S.P. and Jordan, M.I. Reinforcement Learning Algorithm for Partially Observable Markov decision Problems. *Advances in Neural Information Processing Systems 7*, pages 345-352, 1994.

[Lanzi, 2000] Lanzi, P.L. Adaptive Agents with Reinforcement Learning and Internal Memory. *Proceedings of 6th International Conference on Simulation of Adaptive Behavior*, pages 333-342, 2000.

[Lanzi & Wilson, 2000] Lanzi, P.L. and Wilson, S.W. Toward Optimal Classifier System Performance in Non-Markov Environments. *Evolutionary Computation*, Vol.8(4), pages 393-418, 2000.

[Loch & Singh, 1998] Loch, J. and Singh, S.P. Using Eligibility Traces to Find the Best Memoryless Policy in Partially Observable Markov Decision Processes. *Proceedings of 15th International Conference on Machine Learning*, 1998.

[McCallum, 1995] MacCallum, R. A. Instance-Based Utile Distinctions for Reinforcement Learning with Hidden State. *Proceedings of 12th International Conference on Machine Learning*, pages 387-395, 1995.

[Miyazaki & Kobayashi, 1999]
Miyazaki, K. and Kobayashi, S. Proposal for and Algorithm to Improve a Rational Policy in POMDPs. *IEEE International Conference on Systems, Man, and Cybernetics*, pages 285-288, 1999.

[Moriarty et al., 1999] Moriarty, D.E., Schultz A.C. and Grefenstette J.J. Evolutionary Algorithms for Reinforcement Learning. *Journal of Artificial Intelligence Research*, Vol.11, pages 241-276, 1999.

[Peshkin et al., 1999] Peshkin, L., Meuleau N., and Kaelbling L. Learning Policies with External Memory. *Proceedings of 16th International Conference on Machine Learning*, pages 307-314, 1999.

[Singh et al., 1994] Singh, S.P., Jaakkola, T. and Jordan, M.I. Learning Without State-Estimation in Partially Observable Markovian Decision Processes. *Proc. of the 11th International Conference on Machine Learning*, pages 284-292, 1994.

[Singh & Sutton, 1996] Singh, S.P. and Sutton, R.S. Reinforcement Learning with Replacing Eligibility Traces. *Machine Learning*, Vol.22 :1-37, 1996.

[Watkins & Dayan, 1992] Watkins, C. J. H., and Dayan, P. Technical note: Q-learning. *Machine Learning*, Vol.8: 55-68, 1992.

Interactive Evolutionary Computation with Small Population to Generate Gestures in Avatars

A. Berlanga
Depto de Informática
Univ. Carlos III de Madrid
aberlan@ia.uc3m.es

P. Isasi
Depto. de Informática
Univ. Carlos III de Madrid
isasi@ia.uc3m.es

J. Segovia
Facultad de Informática
Univ. Politécnica de Madrid
fsegovia@fi.upm.es

Abstract

In this article a method is presented to use Genetic Algorithms with small populations. The small populations worsen the problem of the premature convergence due to the small existent genetic diversity. The incorporation of a weight that modifies the probability of change of alleles it will be to avoid the premature convergence. This technique has been applied to carry out the design of expressions for avatars.

1 INTRODUCTION

In his book "On the origin of species" Darwin, put forward, as the engine in the evolution of species, the struggle of these to obtain the best resources in their environment. Thus, those living beings that have any characteristic that allows them to gain advantage over their counterparts, in food acquisition, reproductive success, etc. get better chances of surviving and, consequently, of spreading their genetic code along with those capabilities which made them superior. The group of learning techniques that utilize computational models based on the biological evolution as a key element in their design and implementation is referred to as Evolutionary Computation. Genetic Algorithms (Goldberg 1989) was the main technique of Evolutionary Computation. In this technique a whole population of structures is maintained, which evolve according to some specific selection rules and, by means of the application of certain operators, referred as genetic operators –such as the recombination or the mutation operators, for instance— each individual is given a measurement for his or her fitting to the environment, named adequacy value, obtained after an evaluation process. The reproduction focuses on individuals whose environmental adaptation is higher. Although they may seem quite simplistic from a biological evolutionary point of view, this algorithm come out complex enough to provide a powerful, robust mechanism for an adaptive search, (Holland 1995).

The key point in the EC paradigm is usually the evaluation of the individuals. Some times a single evaluation requires a complex and time expensive process. Some other times the evaluation is comparative or subjective. There is not an absolute value of an individual evaluation, but the final result depends on a comparison with other individual in the population. It also could depend on the final result to be produces. In these last cases and interaction with the user is needed to make the evaluation. Anyway, the populations have to be restricted to some few individuals and the canonical genetic algorithm can not be applied.

In this paper, news operators are proposed to overcome the problem related to the small population size. The operators must include devices for the maintenance of genetic diversity.

2 GENETIC ALGORITHMS

The history of Evolutionary Computation starts with Genetic Algorithms in the decade of the seventies. Holland (Holland 1975) was one of the pioneers in its research. Since then the research on Genetic Algorithms and their applications have experienced a great boost and new genetic techniques have developed from Holland's original ideas (Mitchell 1996), (Chambers 1995).

Genetic Algorithms do not work directly on the solutions of a problem, but deal with a codification of such solutions – a chromosome. The chromosomes make up a population of coded solutions. The search is based on the adequacy function as well as on a selection operator, proportional to the estimate, known as the adequacy value, which provides such function. The adequacy function allocates each chromosome an adequacy value. This value indicates how good or deficient is the solution represented by that individual. Genetic Algorithms are responsible for the proliferation of the best adapted individuals and the disappearing of the worst adapted ones, (Goldberg 1989), (Mitchell 1996), by discriminatory applying the genetic operators according to the adequacy value.

Due to the very little information on the domain that Genetic Algorithms employ to lead the search, they are usually applicable to all sorts of problems, specially in domains with scarce information about the pursued objective, or in domains which can change over time; therefore, the genetic search could be sought with certain periodicity. Some of the applications that were carried out using Genetic Algorithms required the development of specific operators that included information from the domain, which invalidates them as generic solution methods. However, some genetic operators, specifically developed for a problem, have proved useful in other kinds of problems.

Evolution can be seen in a simplified way as a genetic information transfer process between individuals, (Wilson 1985). This transfer can be summarized in three rules:

- Selection of individuals to be reproduced. The selection of an individual can occur according to its adaptation to the environment or to the competition with other individuals.
- To recombine the genetic information by means of the crossover operator. This operator preserves the information of old individuals in the new ones.
- Generation of new genetic information. The mutation operator carries out this process. Its purpose is to keep a high degree of genetic diversity within the population; while the crossover operator merely recombines genes, mutation creates them.

The power of Genetic Algorithms resides in the global search that they carry out at the space of solutions. The major concept that supports the theory of convergence is that of the scheme, (Stephens et al. 1999). A scheme describes a set of chains, i.e. solutions, which share common features. The scheme is defined on the alphabet of the chromosome chains, plus the symbol "*" which stands for a "wild card" or substitutive character. A scheme represents a hyperplane of the search space.

One of the main characteristics of Genetic Algorithms, implicit parallelism, arises from the concept of scheme. Every time that a population chain becomes evaluated in order to estimate its adequacy degree, implicitly the hyperplanes that represent it are being evaluated as well. Implicit parallelism implies that the competition among the different hyperplanes is solved in a parallel manner. The theory suggests that, using the reproduction and recombination processes, schemes increase or decrease their presence in the population, through the population chains that represent them.

2.1 GENETIC ALGORITHM IN SMALL POPULATIONS

There are two main fields in which a small population size is required. One of them are problems in which the fitness values is measured by a human operator, this fields is the application of evolutionary computing techniques is artistic as well as functional design of bidimensional and tridimensional objects. Their parallel solution exploration capacity permits genetic techniques to evaluate many designs, surpassing in this aspect the creativity of many artists and engineers, (Sims 1991).Biomorphs are the classical example of the application of evolution to simple shapes (Dawkins 1986), in this case is the user who interactively decides which shapes look closer to the figure of a real insect. Practical applications have been carried out, such as in the creation of tables, (Bentley 1999). The purpose is to find a table design that follows the various guidelines: it must have a large and stable surface at the top, it must contain no floating elements, and so on. This problem is an example of application to validate a general-purpose tool in the 3D genetic design known as GADES (Genetic Algorithm DESigner). Artistic design, not only applied to graphics but to music as well, where it uses AECS (Artistic Evolutionary Computer System) evolutionary techniques, is a flourishing and promising field (Santos et al. 2000). Static design has also been applied to many types of procedural codings (Sims 1993), with purposes of industrial design (Rowland 2000) or purely artistic (Soddu 2000).

The other kind of problems that require a small population is those with high computational cost.

In this paper we have tested the method proposed in problems that interactively the user input a fitness value. A field of recent application of 3D animated design is motivated by Internet success in the communication among people through virtual worlds where each person can adopt an anthropomorphic figure with which express feelings and emotions. Thus, it arises the need to bring forward some form of expressiveness to the gestures, (Segovia et al. 1999), in order to make communication easier among fellows or to apply the sign language of the deaf to avatars (Losson et al. 2000).

All systems applying evolutionary processes require the incorporation of an evaluation function. Evolutionary processes can fall into two categories according to the fitness or adequacy function they use. The adequacy function can be a predetermined one and thus the evaluating process is automatically performed for all the individuals in the population. This is the case of applications where selection criteria can become clearly expressed, for instance with functionality criteria. The problem arises for artistic design, where it is quite difficult to determine what can be felt as pleasant or

expressive; there have been formalizing attempts that used complexity and self-resemblance criteria. For this kind of problems the so-called 'interactive evolution' is used (Lim et al. 1999), (Unemi 2000); the user provides the adequacy value, evaluates according to his artistic criteria which individuals are the best.

This is the position where this paper stands, the generation of gesture sequences in order to express an emotional status cannot be carried out without the intervention of a human being criterion. A generic answer cannot be given, since, after all, the user is showing his or her personality.

In this paper a design scheme is introduced, which is based on genetic algorithms. It permits the carrying out of an optimal search, minimizing the number of alternatives to be tried (Holland 1975). In this scheme, each expression constitutes an individual from a genetic algorithm; that individual is made up of some genes, which are each of the gestures, which make up the expression. Thus, if an individual –an expression– is made up of 5 gestures, it will have 5 genes; each of one will contain a value –or allele– that will become the corresponding gesture.

3 GENERAL SCHEME

When dealing with small populations with genetic algorithm, two main considerations have to be taken into account:

1. There is not too much genetic diversity and the search is restricted to a very small region of the state space. This makes the process to converge to a bad result in few generations.
2. Very often a subjective evaluation is needed and a numerical value is then not useful. In those cases the canonical scheme has to be change.

To overcome the first restriction a modification of the mutation operator has been developed. The individual chromosome incorporates a 'weight' for each gene allele called importance. The significance of a gene allele is a measure of the changing likelihood; the more important an allele is, the less likelihood for its changing. When the user selects an individual its performance will have to pass on to the next generation, that is to say, its gene alleles will have to reduce their probability of being changed. The new allele that substitutes the mutated one is selected from allele pool of genes, proportional to a value that measures the primacy of each allele. This artifice is necessary due to the requirement of a quick convergence and to the small size of the population.

Another important restriction has mentioned in second restriction, is imposed by the user's way of selecting individuals, with the performance, which satisfy him. This restriction is the manner in which the user assigns the fitness value of the individuals in the population. In a genetic algorithm all the individuals from a population are applied a fitness function, which allows obtaining the fitness value. When it is the user who must provide this value, the process has to be different. The user cannot be forced to always provide a value to all individuals. Besides, providing a numerical value can be difficult if the range is huge. It must always be born in mind that the user to whom the tool is addressed has no knowledge of evolutionary computation and the requirement of an adequacy value assignment must be translated into terms which he is able to understand. Thus, in the genetic design tool, the user is asked for a relative selection of the individual performance, something easier to accomplish. The general working scheme, when a new population generation is generated, is shown in Figure 1.

If no individual has been selected, then all the new genotypes are generated through random mutation; the process is the same as in the first generation. This process takes into account the situation in which a user does not like any individual performance at all, not even slightly and he wants to initialize the design process.

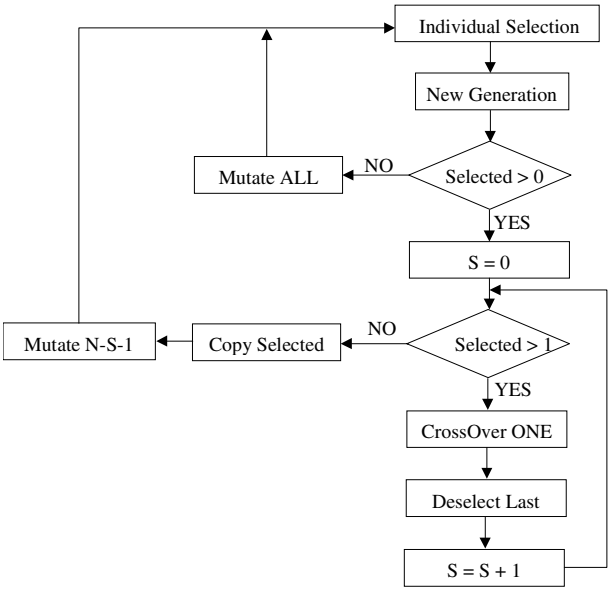


Figure 1. Scheme of process.

When the user triggers the process of generating a new set of individuals with selected ones, two cases are distinguished if there is one or more than one individuals selected to mate. If there is only one selected individual, then this is copied on to the next generation and the rest are generated by means of the mutation of the selected one, until the population has been completed.

The mutation process alters a gene with a probability inversely proportional to its significance. The more important a gene is, the less likelihood of its changing. This device is necessary in order to ease the convergence of the algorithm. Due to the small size of the population, if a classical genetic algorithm were applied, in very few generations a convergence of the population would occur, quickly losing many alleles. Importance is associated to each allele and through genes. It is modified according to the following equation 1:

$$I(t+1) = \begin{cases} I(t)/\alpha, & \text{if not muted} \\ I(t)\,\alpha, & \text{if muted} \end{cases} \qquad 0<<\alpha<1 \qquad (1)$$

If the gene does not mute, then its importance is increased in a factor of $1/\alpha$; if it is altered, then its importance decreases in an α factor. At this paper the value 0.75 has been chosen for α.

If it have been selected k individuals, then k-1 chromosomes are generated by crossover and the remainder by the mutation of the first selected one. The uniform crossover operator is applied with the probability for a gene, p_g, to pass on to the new individual by means of the following formula, equation 2:

$$p_g = \frac{n-o+1}{\sum_{i=1}^{n} i} \qquad (2)$$

Where n is the number of selected individuals to mate and o is the fitness value of the individual.

The selection of an allele for a new individual's gene uses the method of the roulette applied to the genes of selected individuals. With this modification of the crossover operator more importance is given to genes than to the whole chromosome. This imposition arises once again from the limitation of the population's size.

4 RESULTS

The problem that this paper tries to clarify has the originality of generating complex gestures from another simple ones. The target in animations has usually been the life-like movement (Lim et al. 1999). In this sense, realistic movement is already present in simple gestures aiming at a more complex gesture through expressiveness, the designed animations which could be used in non-verbal communication.The application of an evolutionary tool in order to generate lively gestures in avatars has already been carried out by various authors (Segovia et al. 1999), but they were static gestures. A design tool was developed as front-end in order to allow to human evolve the desired complex gesture of his avatar.

Avatars –virtual beings– try to simulate some specific behavior within a virtual environment. Each avatar is, therefore, carrier of a personality. This personality will be strongly determined by the user who is manipulating the avatar, but it will also have an essential component closely related to the avatar expressiveness itself. So, for instance, a user will be capable of telling the avatar to react in an aggressive way at a given situation, but he may think that it is not quite correct the way in which the avatar expresses that aggressiveness. It seems clear enough that the avatar's own expressiveness will determine its own behavior and that the suitability of the expressiveness will depend on the user who is handling it. Therefore, the programming of certain 'expressiveness' will be inappropriate in most cases; it should be the user himself the one in charge of the design of each and every 'expressiveness' of his avatar. This personalization plays a very important part in order for a user to decide on using an avatar and to really identify with it.

The problem with avatars' personalization is which device should be used to develop it. All gesture configuration mechanisms are extremely complicated and the user is neither a computer expert, nor a programmer, nor has he got to carry out a complex design process. Therefore, a simple, self-contained model of avatar is needed as well as an applicable gesture design model; a device that makes the mechanics of gesture generation transparent.

This is what has led to the development of a design of gestures at a complete level, but these are simple gestures, which, linked up can create a great deal of expressions. In this model an expression is merely a sequence, of no prefixed length, of simple gestures.

The system is made up of 10 different simple gestures as a whole; if we consider expressions of an average length of 5 gestures, then we avail about hundred thousand different expressions. This scheme has the advantage of being able to generate a very large number of possible expressions; but this makes the expression design scheme very sophisticated: the plain election of gestures is not possible, due to the enormous number of existing possibilities.

In order to solve this problem, a design scheme has been developed, which is based on genetic algorithms. It permits the carrying out of an optimal search, minimizing the number of alternatives to be tried (Holland 1975). In this scheme, each expression constitutes an individual from a genetic algorithm; that individual is made up of some genes, which are each of the gestures, which make up the expression. Thus, if an individual –an expression– is made up of 5 gestures, it will have 5 genes, each of one

will contain a value –or allele– that will become the corresponding gesture.

Genetic Algorithm scheme is referred to as exogenous Genetic Algorithm. In the case of the design of avatars, there is no such a function to determine which designs are best or worst; therefore, we lack of a fitness function to lead the search. In this case we must opt for an endogen genetic algorithm scheme, in which the system is producing new solutions only in terms of the surviving ones. The survival of solutions is determined by the user, who is the one that indicates which solutions seem better to him, from among the suggested ones.

Upon entering the system of genetic gesture generation, the user encounters four avatars placed over four pedestals, plus a set of different buttons, Figure 4. We have used the ActiveWorld© avatars to generate complex movements.

The user can perform the following actions:

- Selecting avatars.

Figure ??. Desing tool

- The selection of a particular avatar is done by clicking with the mouse pointer over it, Figure ??. The pedestal will display the order in which the avatar has been selected. It must be remembered that the selection order is very important, since the amount of genetic code that avatar pass on to the next generation goes in terms of their selection order. If a selected avatar is clicked over, this one loses its selection and the order of the rest is reestimated.
- Watching an avatar's gestures. When clicking over a pedestal, the avatar standing on it shows a sequence of gestures.
- By clicking over the button "Move", all avatars make their gestures, starting from the avatar most to the right; when this one has finished its movements, it starts the one to its left, until completion of all gestures.
- Generating a new population of avatars. When clicking over the button "New", a new generation of gestures is produced. The avatars having new gestures replace the old ones and no one appears selected. The avatars' new gestures are then displayed.
- Saving the desired gesture and exiting. When the user clicks over the button "Exit", then the program understands that the user has finished the task of defining a complex gesture. The simple gestures, which make up the complex sequence of the avatar selected with number one, are saved inside a file. All pedestals buttons and avatars associated to the gesture generation tool disappear. If the user has no selected avatars, then no file is saved.

All buttons, pedestals and avatars are inserted in ActiveWorld© such that they behave in the same manner as any other objet would do; the user can move around, getting closer, going away and turning around all these components.

It must be emphasized the simplicity of the design; just by selecting and invoking a new generation the creation of complex gestures is attained.

The check of the convergence it was verified with the number of generations that a user needed to obtain the desired expression. In average eight generations was needed to obtain the goal movements. This result shows that the dynamic of the genetic process trades off between the convergence and the maintaining of the genetic diversity in order to explore the solutions space.

5 CONCLUSIONS

The restriction in the small number of individuals along with the peculiarity of being the user who supplies the adequacy value have made it necessary to add some variations to the classical genetic operators. The use of a weight to modify the probability of change an allele solves the problem of convergence premature in genetic algorithm with small populations.

This method has proven to be useful in the artistic design. Other important application could be problems with high computational cost.

References

Goldberg D.E. (1989) *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison Wesley, Reading Massachusetts.

Holland J.H. (1995) *Hidden order: how adaptation builds complexity*. Reading Massachusetts, Addison-Wesley.

Holland J.H. (1975*) Adaptation in Natural and Artificial Systems*. University of Michigan Press.

Mitchell M. (1996) *An Introduction to Genetic Algorithms*. MIT Press, Massachusetts.

Chambers L. (1995) *Practical handbook of genetic algorithms*. Vols. 1, 2 edited by Lance Chambers, CRC Press.

Wilson S. (1985*) Knowledge growth in an Artificial Animal*. Proceedings of the 1st International Conference on Genetic Algorithms and their Applications, (pp. 16-23).

Stephens C., Waelbroeck H. (1999*) Shemata Evolution and Building Blocks*. Evolutionay Computation 7(2). MIT Press. (pp. 109-124).

Sims K., (1991) *Artificial Evolution for Computer Graphics*, Computer Graphics, Vol. 25, (4), pp. 319-328.

Dawkins R. (1986) *The blind watchmaker*, Longman Scientific and Technical, Harlow.

Bentley P. (1999) *From Coffee Tables to Hospitals: Generic Evolutionary Design*, Evolutionary design by computers, Morgan-Kauffman, pp. 405-423.

Santos A, Dorado J., Romero J., Arcay B., Rodriguez J. (2000) *Artistic Evolutionary Computer Systems*, Proceedings of the Genetic and Evolutionary Computation Conference Workshop Program, Las Vegas.

Sims K. (1993) *Interactive Evolution of equations for procedural models*, The visual computer 9. pp.466-476. Springer-Verlag.

Rowland D. (2000) *Evolutionary Co-operative Desing Methodology: The genetic sculpture park*. Proceedings of the Genetic and Evolutionary Computation Conference Workshop Program, Las Vegas.

Soddu C. (2000) *Argenia, Art's Idea as Generative Code*, Proceedings of the Genetic and Evolutionary Computation Conference Workshop Program, Las Vegas.

Segovia J., Antonio A., Imbert R., Herrero P., Antonini R. (1999) *Evolución de gestos en mundos virtuales*, Proceedings of CAEPIA 99.

Losson O., Cantegrit B. (2000) *Generation of Sentences in Sign Language by a 3-D Expressive Avatar*, Proceedings on Systemics, Cybernetics and Informatics, SCI 2000, vol 3, Orlando.

Lim I.S., Thalmann D. (1999) *Pro-actively Interactive Evolution for Computer Animation*, Proceedings of Eurographics Workshop on Animation and Simulation, CAS99, Milan.

Unemi T. (2000) *SBART 2.4: an IEC Tool for Creating 2D images, movies and collage*, Proceedings of the Genetic and Evolutionary Computation Conference Workshop Program, Las Vegas.

Repeated Structure and Dissociation of Genotypic and Phenotypic Complexity in Artificial Ontogeny

Josh C. Bongard **Rolf Pfeifer**
Artificial Intelligence Laboratory
University of Zürich
CH-8057 Zürich, Switzerland
[bongard|pfeifer]@ifi.unizh.ch

Abstract

In this paper, a minimal model of ontogenetic development, combined with differential gene expression and a genetic algorithm, is used to evolve both the morphology and neural control of agents that perform a block-pushing task in a physically-realistic, virtual environment. We refer to this methodology as artificial ontogeny (AO). It is demonstrated that evolved genetic regulatory networks in AO give rise to hierarchical, repeated phenotypic structures. Moreover, it is shown that the indirect genotype to phenotype mapping results in a dissociation between the information content in the genome, and the complexity of the evolved agent. It is argued that these findings support the claim that artificial ontogeny is a useful design tool for the evolutionary design of virtual agents and real-world robots.

1 Introduction

In the field of evolutionary robotics and artificial life, emphasis is increasingly coming to bear on the question of evolvability: that is, how well the artificial evolutionary system continually discovers agents or robots better adapted to the task at hand (Wagner & Altenberg 1996; Kirschner & Gerhart 1998). It is becoming apparent that modularity, at either the genetic or phenotypic level, or both, is a necessary characteristic of highly evolvable systems (Wagner 1995; Rotaru-Varga 1999; Calabretta *et al* 2000).

Developmental geneticists have made clear that evolved genetic regulatory networks in biological DNA contain master control switch genes, known as *Hox* genes, which orchestrate the transcription of other genes to grow high-level repeated structure, such as the segments in *D. melanogaster* (refer to Gehring & Ruddle (1998) for an overview). It has been shown in a dramatic set of experiments (Lewis 1978) that mutations of *Hox* genes can lead to large-scale but localized changes in phenotype. It has been argued (Raff 1996) that in some cases, differentiation and/or duplication of a feature may allow evolution to co-opt one copy of the feature to perform a different functional role. This process is known as exaptation (Gould & Vrba 1982). A similar mechanism has been shown to have occurred at the gene level (Ohno 1970).

Riedl (1978) demonstrated that the information content of a complex organism is many orders of magnitude higher than that contained in the genome, and has argued that the increased complexity arises from the hierarchical organization of organic units. Raff (1996) has pointed out the same principle holds for the complex processes that take place during ontogeny. Others have argued (Delleart & Beer 1994) that an indirect, developmental genotype to phenotype mapping allows for artificial evolution to discover more complex phenotypes than is possible with direct mappings.

In this paper we introduce an augmented genetic algorithm, in which the genomes are treated as genetic regulatory networks. The changing expression patterns of these networks over time leads to the growth of both the morphology and neural control of a multi-unit, articulated agent, starting from a single unit. We refer to this system as artificial ontogeny (AO), and as is shown in Bongard & Pfeifer (2001), such a system can be used to evolve agents that perform non-trivial behaviours in a physically-realistic, virtual environment, such as directed locomotion in a noisy environment. It is reported here that in agents evolved for a block-pushing task, the morphologies exhibit hierarchical, repeated structure. Evolved agents from previous studies contain repeated structure, however these studies relied on more direct, parametric encoding schemes (Sims 1994; Ventrella 1996; Komosinski & Ulatowski 1999; Lipson & Pollack 2000). Conversely, in studies conducted using developmental encoding schemes, the agents are relatively simple, and do not exhibit any higher-order, repeated structure (Delleart & Beer 1994; Jakobi 1995).

In the next section the morphologies of the evolved agents are explained, the differential gene expression model used to grow them, as well as the method by which neural networks are grown along with the developing morphology of the agent. The following section reports the results of a set of evolutionary runs in which agents are evolved for a block-pushing task, and provides some analysis of the resulting phenotypes and gene expression patterns. The penultimate section discusses the adaptive potential of the AO system, and promising areas of future research. The final section provides some concluding remarks.

2 The Model

In this system, there is a translation from a linear genotype into a three-dimensional agent complete with sensors, actuatable limbs and internal neural architecture, such as in Sims (1994), Ventrella (1996), Komosinski & Ulatowski (1999), Bongard & Paul (2000), and Lipson & Pollack (2000). However unlike these other methods, the genotype to phenotype translation described here takes place via ontogenetic processes, in which differential gene expression, coupled with the diffusion of gene products, transforms a single structural unit in a continuous manner into an articulated agent, composed of several units, some or all of which contain sensors, actuators and internal neural structure.

2.1 Agent Morphology

Each agent evaluated in the physically-realistic simulation is composed of one or more units. For the experiments reported here, spheres are used to represent these units. By scaling up the number of units used to construct an agent, increasingly arbitrary morphologies can be evolved. Each agent begins its ontogenetic development as a single unit. Depending on the changing concentrations of the gene products within this unit, the unit may grow in size, until the radius grows to twice that of the unit's original radius. At this point the unit splits into two units; the radii of both the parent and child units are then reset to the default radius.[1]

Each unit contains: zero to six joints attaching it other units via rigid connectors; a copy of the genome directing development of the given agent; and six diffusion sites. Each of the six diffusion sites are located midway along the six line segments originating at the centre of the sphere, terminating at the surface, and pointing north, south, west, east, up and down. Each diffusion site contains zero or more diffusing gene products and zero or more sensor, motor and internal neurons. The neurons at a diffusion site may be connected to other neurons at the same diffusion site, another diffusion site within the same unit, or to neurons in other units. Each of the components of a unit are described in more detail in the following sub-sections.

A newly-created unit is attached to its parent unit in one of six possible directions using a rigid connector that maintains a constant distance between the units, even though one or both of the attached units may continue to grow in size. The new unit is placed opposite to the diffusion site in the parent unit with the maximum concentration of growth-enhancing gene product. After a unit splits from its parent unit, the two units are attached with a rigid connector, the ends of which are located in the centres of the two units. The parent unit is fixed to the rigid connector. The new unit is attached to the rigid connector by a one degree of freedom rotational joint. The fulcrum of the joint is placed in the centre of the new unit. Joints can rotate between $-\frac{\pi}{2}$ and $\frac{\pi}{2}$ radians of their starting orientation. The axis about which a unit's joint rotates is set perpendicular to the plane described by the parent unit, the child unit, and the first unit to split from the child unit. If no units split from a unit, that unit's rotational joint is removed, and the unit is fixed to the rigid connector it shares with its parent unit. This precludes the evolution of wheels, in which units rotate about their own centre of mass. Fig. 1 illustrates the creation and actuation of an agent's joints in more detail.

The agent's behaviour is dependent on the real-time propagation of sensory information through its neural network to motor neurons, which actuate the agent's joints.

There are three types of sensors that artificial evolution may embed within the units of the agent: touch sensors, proprioceptive sensors, and light sensors. Touch sensor neurons return a maximal positive signal if the unit in which they are embedded is in contact with either the target object or the ground, or a maximal negative signal otherwise. Proprioceptive sensors return a signal commensurate with the angle described by the two rigid connectors forming the rotational joint within that unit (refer to Fig. 1). Light sensor neurons return a signal that is linearly correlated to the distance between the unit in which the sensor is embedded and the target object in the environment. The light sensors are not physically simulated, but calculated geometrically.

The agent achieves motion by actuating its joints. This is accomplished by averaging the activations of all the motor neurons within each unit, and scaling the value between $-\frac{\pi}{2}$ and $\frac{\pi}{2}$. Torque is then applied to the rota-

[1] Although the agent grows through repeated division of units, and each unit retains a copy of the genome that directs the agent's growth, the units used in this model are not to be equated with the biological concept of a cell, such as in the AES system (Eggenberger 1997), nor are they equivalent to the units employed in the parametric models mentioned above. Rather, repeated division is a useful abstraction that allows for a relatively continuous transition from a single unit into a fully developed agent composed of many such units.

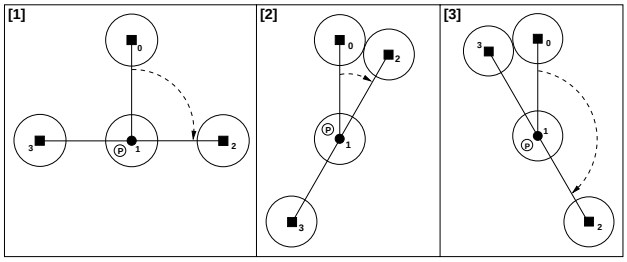


Figure 1: **Architecture of articulated joints** Panels [1] through [3] depict part of an agent's morphology. In this hypothetical scenario, unit 1 split from unit 0, and units 2 and 3 split from unit 1. The black squares represent fused joints; the black circles represent rotational joints. The fused joints connecting units 2 and 3 to unit 1 are not shown for clarity. Rotation occurs through the plane described by the angle between units 0, 1 and 2. Panel [1] shows the configuration of the agent immediately after growth, before activation of the neural network. Unit 1 contains a proprioceptive sensor neuron, which emits a zero signal. In panel [2], unit 1 has rotated counterclockwise, either due to internal actuation or external forces. The proprioceptive sensor in unit 1 emits a nearly maximal negative value. In panel [3], the hinge in unit 1 reaches has rotated clockwise: the proprioceptive sensor now emits a nearly maximal positive signal. Note that the architecture of the agent's morphology precludes the hinge from reaching its rotational limits, and the proprioceptive sensor from generating either a maximally negative or positive signal.

tional joints such that the angle between the two rigid connectors forming the joint matches this value. The desired angle may not be achieved if: there is an external obstruction; the units attached to the rigid connectors experience opposing internal or external forces; or the values emitted by the motor neurons change over time. Note that failure to achieve the desired angle may be exploited by evolution, and may be a necessary dynamic of the agent's actions. If a unit contains no motor neurons, the rotational joint in that unit is passive.

Internal neurons can also be incorporated by evolution into an agent's neural network, in order to propagate signals from sensor to motor neurons. Two additional neuron types are available to evolution. Bias neurons emit a constant, maximum positive value. Oscillatory neurons emit a sinusoidal output signal. The summed input to an oscillatory neuron modulates the frequency of the output signal, with large input signals producing an output signal with a high frequency, and low input signals producing a low frequency output signal.

2.2 Differential Gene Expression

Unlike the recursive parametric encoding schemes mentioned above, each genome in the AO system is treated as a genetic regulatory network (Kauffman 1993, Jakobi 1995, Eggenberger 1997 and Reil 1999), in which genes produce gene products that either have a direct phenotypic effect or regulate the expression of other genes.

For each genome to be evaluated in the population, it is first copied into the single unit from which the eventual fully-formed agent develops. The genome is then scanned by a parser, which marks the site of promotor sites. Promotor sites indicate the starting position of a gene along the genome. A value in the genome is treated as a promotor site if the value is below $\frac{n}{l}$, where n is the average number of genes that should appear within each initial random genome, and l is the length of genomes in the initial, random genetic algorithm population. This is done so that, given a starting population of random genomes, each genome will contain, on average, the desired number of genes. In the results reported in the next section, $l = 100$ and $n = 10$, causing values between 0.00 and 0.10 to serve as promotor site indicators.

Fig. 2 provides a pictorial representation of a genome directing the growth of an agent. The seven floating-point values following a gene's promotor site supply the parameter values for the gene. If the first value ($P1$ in Fig. 2) is less than 0.5, gene expression is repressed by presence of the gene product which regulates its expression; otherwise gene expression is enhanced by presence of its regulating gene product. The second value ($P2$ in Fig. 2) indicates which of the 24 possible gene products regulates the gene's expression. The third value ($P3$ in Fig. 2) indicates which of the 24 possible gene products is produced if this gene is expressed. The fourth value ($P4$ in Fig. 2) indicates which of the 6 gene product diffusion sites the gene product is diffused from if this gene is expressed. The fifth value ($P5$ in Fig. 2) indicates the concentration of the gene product that should be injected into the diffusion site if the gene is expressed. The sixth and seventh values ($P6$ and $P7$ in Fig. 2) denote the concentration range of the regulating gene product to which the gene responds. If the concentration of the regulating gene product to which the gene responds is within this range, and the gene is enhanced by presence of its regulating gene product, the gene is expressed; otherwise, gene expression is repressed. Genes that are repressed by their regulating gene product are expressed if the gene product's concentration is outside the denoted range, and repressed otherwise.

After the genes in the genome have been located, the originating unit of the agent to be grown is injected with a small amount of gene product at diffusion site 1. Due to gene product diffusion, a gradient is rapidly established in this first unit, among the 6 diffusion sites. This is analogous to the establishment of a gradient of maternal gene product in fruit flies, which leads to the determination of the primary body axis (Anderson 1984), and breaking of symmetry in early embryogenesis. It can be seen from Fig. 3 that the degree of symmetry in evolved agents varies, and is under evolutionary control.

As the injected gene product diffuses throughout the

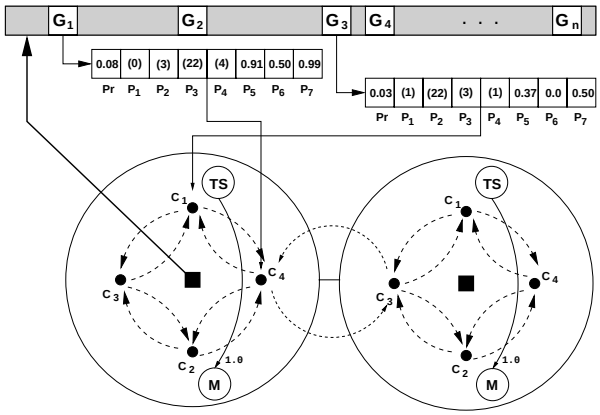


Figure 2: **Ontogenetic interactions in a developing agent** Two structural units of an agent are shown above, but only displayed in two dimensions for clarity. For this reason, only four of the six gene product diffusion sites are shown; the other two lie at the top and bottom of the spherical units. The genome of the agent is displayed, along with parameter values for two genes. The values in parentheses indicate that these values are rounded to integer values. Gene G_1 indicates that it is repressed (parameter P_1) by concentrations of gene product 3 (P_2) between 0.5 and 0.99 (P_6, P_7). Otherwise, it diffuses gene product 22 (P_3) from gene product diffusion location 4 (P_4), indicated in the diagram by C_4. Note that genes G_1 and G_3 emit gene products which regulate the other's expression. The thick dotted lines indicate gene product diffusion between diffusion sites within a unit; the thin dotted lines indicate gene product diffusion between units. Both units contain a touch sensor neuron (TS) and a motor neuron (M) connected by excitatory synapses.

unit, it may enhance or repress the expression of genes along the genome, which in turn may diffuse other gene products. There are 24 different types of gene products. Two affect the growth of the unit in which they diffuse. At each time step of the development phase, the difference between the concentration of these two chemicals is computed. If the difference is positive, the radius of the unit is increased a small increment; if the difference is negative, the unit does not grow in size. Thus these two chemicals function as growth enhancer and growth repressor, respectively. If the radius of a unit reaches twice that of its original radius, a split event is initiated. The radius of the parent unit is halved, the gene product diffusion site with the maximum concentration of growth enhancer is located, and a new unit is attached to the parent unit at this position. Half of the amounts of all gene products at this diffusion site are moved to the neighbouring diffusion site in the new unit. A copy of the genome is assigned to the new unit. The gene expression patterns of the parent and child units are now independent, except for indirect influence through inter-unit diffusion of gene products.

There are then 17 other chemicals which affect the growth of the agent's neural network, and are explained in the next section. Finally, five gene products have no direct phenotypic effect, but rather may only affect the expression of other genes. That is, concentrations of these gene products at diffusion sites can enhance or repress gene expression in that unit (like the other 19 gene products), but cannot modify neural structure, or stimulate or repress the growth of that unit.

All 24 gene products share the same fixed, constant diffusion coefficients. For each time step that a gene emits gene product, the concentration of that gene product, at the diffusion site encoded in the gene, is increased by the amount encoded in the gene (which ranges between 0.0 and 1.0), divided by 100. All gene product concentrations, at all diffusion sites, decay by 0.005 at each time step. Gene products diffuse between neighbouring diffusion sites within a unit at one-half this rate. Gene products diffuse between neighbouring units at one-eighth the rate of intra-unit diffusion.

2.3 Neural Growth

Cellular encoding (Gruau 1996) has been incorporated into our model to achieve the correlated growth of morphology and neural structure in a developing agent. Cellular encoding is a developmental method for evolving both the architecture and synaptic weights of a neural network. The process involves starting with a simple neural network of only one or a few neurons, and iteratively or recursively applying rewrite rules that modify the architecture or synaptic weights of the growing network.

In our model, for each new unit that is created, including the first unit, a small neural network is created as follows: A touch sensor neuron (TS) is placed at diffusion site 1, a motor neuron (M) is placed at diffusion site 2, and a synapse with a weight of 1.0 is connected from the sensor neuron to the motor neuron (refer to Fig. 2). When a unit undergoes a split event, any neurons at the diffusion site where the split event was initiated are moved to the neighbouring diffusion site of the new unit. For example, if a unit splits, and the new unit is attached near its northern face, all the neurons in the northern diffusion site of the parent unit are moved to the southern diffusion site in the new unit. Neurons may also move from one diffusion site to another within a unit, depending on the concentrations of gene products at those sites. The combination of these dynamics may lead to the directed migration of neurons across the units as they divide. As they migrate, synapses connecting these neurons are maintained: although this process is different from the neural growth cone model (in which biological neurons innervate distant cells using exploratory synaptic outgrowths (Kater 1990)) and instantiations of this model (Delleart & Beer 1994; Jakobi 1995), it does allow for neurons in distant units to remain connected.

Each of the 17 gene products responsible for neural development correspond to one rewrite operation that

modifies local neural structure. At each diffusion site, two pointers are maintained: the first pointer indicates which synapse will undergo any synaptic modification operations; the second pointer indicates which neuron will undergo any neuronal modification operations. The 17 rewrite rules correspond to serial and parallel duplication of neurons; deletion of neurons and synapses; increase and decrease of synaptic weight; duplication of synapses; neuron migration within a unit; changing of the afferent and efferent target of synapses; and changing of neuron type. If the concentration of one of these 17 gene products at a diffusion site exceeds 0.8, and there is neural structure at that site, the corresponding operation is applied to the neural structure there. Once development is complete, the neural network that has grown within the agent is activated. At each time step of the evaluation period, the input to each neuron is summed, and thresholded using the activation function $\frac{2}{1+e^{-s}} - 1$, where x is the neuron's summed input. Neuron values can range between 1 and -1. Using this neural development scheme, the AO system is able to evolve dynamic, recurrent neural networks that propagate neural signals from sensor neurons to motor neurons distributed throughout an agent's body.

3 Results and Analysis

The evolutionary runs reported in this section were conducted using a variable length genetic algorithm; the genomes were strings of floating-point values ranging between 0.00 and 1.00, rounded to a precision of two decimal places. A population size of 200 was used, and each run lasted for 200 generations. All genomes in the initial random population have a starting length of 100 values. The mutation rate was set to produce, on average, random replacement of a single value for each new genome. Unequal crossover was employed, which allowed for gene duplication and deletion. Tournament selection, with a tournament size of 2, was used to select genomes to participate in crossover.

As in Bongard & Pfeifer (2001), agents are evaluated in a physically-realistic virtual environment using a commercially available physics-based simulation package[2]. Each genome in the population is evaluated as follows: The genome is copied into a single unit, which is then placed in a virtual, three-dimensional environment. A target cube is placed 20 units[3] to the north of the unit; the sides of the cube are 70 units long. Morphological and neural development is allowed to proceed, as described in the previous section, for 500 time steps. After the development phase, the neural network is activated, and the agent is allowed to operate in its virtual environment for 1000 time steps. The fitness of an agent is given as $\sum_{i=2}^{1000} n(t(i-1)) - n(t(i))$, where $n(t(i))$ is the northern distance of the centre of the cube from the origin at time t. Thus the agent is rewarded for reaching the cube as fast as possible, and pushing it as far as possible. By making the cube much larger than the units comprising an agent, we can exert indirect selection towards large agents: agents must have a large mass in order to exert a large force against the cube. Agents a) and b) in Fig. 3 depict the morphologies of the most fit agents from two independent runs. Agents c) and d) were the most fit agents at generation 110 and 130 of the run shown in Fig. 4.

In order to detect the presence of hierarchical, repeated structure in evolved agents, the local neural structure within units was used as a signature to dis-

[2]MathEngine PLC, Oxford, UK, `www.mathengine.com`

[3]Spatial distance in the physics-based simulator is relative; we treat a 'unit' as equal to the default radius of a newly-created unit.

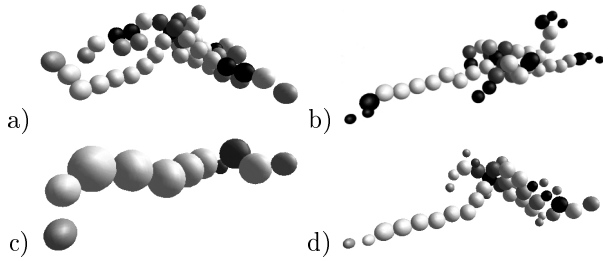


Figure 3: **Four agent morphologies** The block is not shown in the figure for the sake of clarity, but lies just to the left of the agents. The rigid connectors are also not shown. The white units indicate the presence of both sensor and motor neurons within that unit. The light gray units indicate the presence of both sensor and motor neurons in that unit, but the one or more motor neurons do not actuate the rotational joint in that unit either because there are no input connections to the motor neuron, or because there is no joint within this unit. The dark gray units indicate the presence of sensor neurons, but no motor neurons. The black units indicate the unit contains neither sensor nor motor neurons.

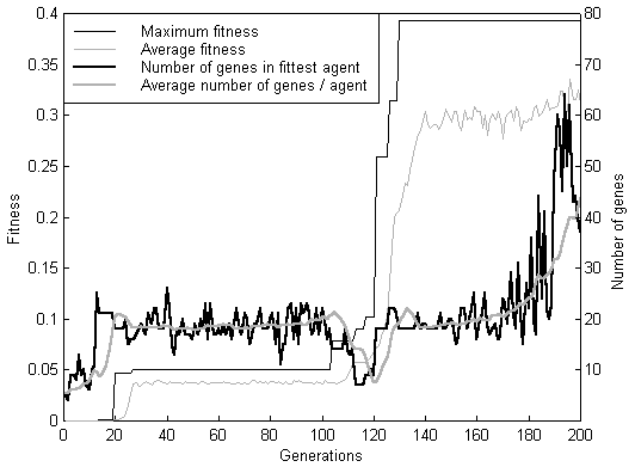


Figure 4: **Results from a typical run.** Genome length was found to be roughly proportional to the number of genes, and is not plotted.

tinguish between units. For instance in agent a) in Fig. 3, the two neighbouring units that have lost their motor and sensory capabilities are repeated twice. In the right-hand agent, the three most distal units in the three main appendages have also lost their motor and sensory capabilities.

The most fit agent from each of the nine evolutionary runs was extracted, and the number of motor and sensor neurons in each unit of each appendage was counted. The tallies for each unit are reported in Fig. 5. Because the units comprising an agent are organized as directed trees, appendages can be determined as follows: for each terminal unit in the agent, traverse up the tree until a unit is found with more than one child unit. The units that were traversed, minus the last one counted, comprise an appendage.

Finally, the gene expression patterns of four units are reported in Fig. 6. Units a) and c) give rise to appendages with similar patterns of local neural structure, and themselves have similar internal neural structure. Units b) and d) do not give rise to further structure, and have similar neural structure. This structure is different from units a) and c). The four units are indicated in bold in Fig. 5. Units a) through d) all split from the same parent unit during ontogeny, but appear at increasingly later times during the agent's development.

4 Discussion and Future Work

Fig. 4 indicates that no agent is able to push the block until generation 20; this event is accompanied by a doubling in the number of genes carried by these more fit agents. However, the gene complement of agents does not increase considerably during the rapid fitness increase which occurs around generation 120. Agents c) and d) in Fig. 3 indicate that this fitness increase was accomplished by a radical increase and reorganization of the agent's morphology and neural control. This suggests that the AO system is exhibiting that predicted property of indirect encoding schemes, that is, large increases in phenotypic complexity[4] without corresponding large increases in genome size.

Fig. 5 indicates that invariably, evolution converges on agents that exhibit hierarchical repeated structure. This can be seen most clearly in the first agent in Fig. 5, in which the first agent contains three similar appendages with three distal units each containing neither motor nor sensor neurons. Moreover, Fig. 5 indicates that genetic changes to local neural structure can be repeated both within an appendage—as seen by the deletion of function in the three distal units—and across appendages—as seen by the triple deletion of function repeated in three different appendages.[5] In other words, agents tend to have appendages in which local neural structure is repeated along the length of the appendage, and appendages themselves are repeated. It is important to note that this structure—which we, as observers, consider hierarchical, repeated structure—is the result of the complex, dynamical interplay between the evolved genetic regulatory networks, the developmental process, and the selection pressure exerted on the evolving population. This suggests that the study of genetic regulatory networks should not be conducted in isolation, but rather in the context of embodied agents evolved for a specific task. This would then give us a clearer picture of how both natural and artificial evolution shape such regulatory networks over time.

Finally, Fig. 6 indicates that the units that give rise to similar appendages have similar gene expression patterns, even though they appear at different times during ontogeny. Similarly, the gene expression patterns of two other units, which appear at roughly the same time as the other two units, correspond. However, the gene expression patterns are different between these two pairs of units. This is shown by the expression of the first marked gene in units a) and c), but not in b) and d); a short expression band for the other three marked genes appears during late ontogeny in units b) and d), but not in a) and c). This indicates that, even though all four of these units originated from the same parent unit, and at roughly the same time during ontogeny, the units which gave rise to appendages have a shared pattern of expression that differs from the pair that does not give rise to appendages. This result suggests that future studies might uncover one or a small set of genes that lead to the growth of higher-order structure when active, but repress such growth when inactive. These genes would serve as analogues of *Hox* genes in biological organisms, and would indicate that such genes are the natural result of evolution when coupled with ontogeny and differential gene expression. Our future studies will also include more detailed analysis of the evolved genetic networks.

5 Conclusions

To conclude, this paper has demonstrated that a minimal model of biological development, coupled with a genetic algorithm that allows for gene duplication and deletion, is sufficient to evolve agents that perform a non-trivial task in a physics-based virtual environment. Moreover, this system—referred to as artificial ontogeny—is sufficient to produce hierarchical, repeated phenotypic structure. In addition, it has been shown that the inclusion of differential gene expression in artificial ontogeny dissociates the information con-

[4] In this context, complexity is simply taken as the number and organization of units, and variation in local neural structure within those units.

[5] From visual inspection of these agent's behaviours, it seems as if these appendages use a whiplike motion, requiring strong actuation at the proximal end and little or no actuation at the distal end.

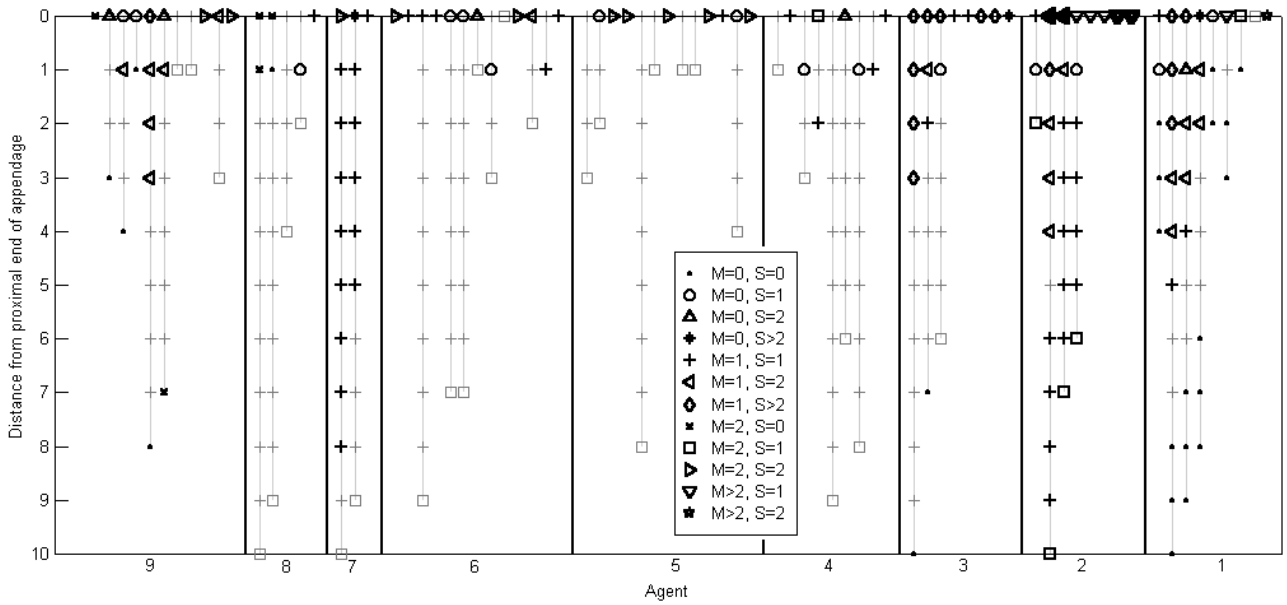


Figure 5: **Neural composition of nine evolved agents** Each symbol indicates the number of motor and sensor neurons in a structural unit. Neural structure is only reported for units that are part of an appendage. Units comprising an appendage are linked by gray lines. Gray symbols indicate no rewrite rules have been applied to the neural structure in that unit; black symbols indicate units in which genetic manipulation of local structure has occurred. The gene expression patterns of the four units indicated by bold symbols is shown in Fig. 6. Agent 1 corresponds to agent b) in Fig. 3.

tent of the genome from the complexity of the evolved phenotype.

Both of these properties point to the high evolvability of the AO system: both the production of hierarchical, repeated organization and the dissociation of genotypic and phenotypic complexity are necessary if artificial evolution is to prove useful for the design of robots that solve increasingly complex tasks, the ultimate goal of evolutionary robotics research.

References

J. Bongard and C. Paul (2000). Investigating morphological symmetry and locomotive efficiency using virtual embodied evolution. In *Proceedings of the Sixth International Conference on Simulation of Adaptive Behaviour*, pp. 420–429. MIT Press.

J. Bongard and R. Pfeifer (2001). Evolving complete agents using artificial ontogeny. To appear in *Proceedings of The First International Workshop on Morpho-functional Machines*, Springer-Verlag, Berlin.

K. V. Anderson and C. Nüsslein-Volhard (1984). Information for the dorso-ventral pattern of the *Drosophila* embryo is stored in maternal mRNA. In *Nature* **311**:223–227.

R. Calabretta, S. Nolfi, D. Parisi and G. P. Wagner (2000). Duplication of modules facilitates the evolution of functional specialization. In *Artificial Life* **6**(1):69–84. Cambridge, Mass: MIT Press.

F. Delleart, and R. D. Beer (1994). Toward an evolvable model of development for autonomous agent synthesis. In *Artificial Life IV*, 246–257. MIT Press.

P. Eggenberger (1997). Evolving morphologies of simulated 3D organisms based on differential gene expression. In *Proceedings of the Fourth European Conference on Artificial Life*, 205–213. Berlin: Springer-Verlag.

W. J. Gehring and F. Ruddle (1998). *Master Control Genes in Development and Evolution: The Homeobox Story (Terry Lectures)*, New Haven: Yale University Press.

S. J. Gould and E. S. Vrba (1982). Exaptation—a missing term in the science of form. In *Paleobiology* **8**:4–15.

F. Gruau, D. Whitley, and L. Pyeatt (1996). A comparison between cellular encoding and direct encoding for genetic neural networks. In Proceedings of the First Genetic Programming Conference, 81–89. MIT Press.

N. Jakobi (1995). Harnessing morphogenesis. Presented at *The International Conference on Information Processing in Cells and Tissues*, Liverpool, UK.

S. B. Kater and P. B. Guthrie (1990). Neuronal growth cone as an integrator of complex environmental information. In *Cold Spring Harbor Symposia on Quantitative Biology, Volume LV*, 359–370. Cold Spring Harbor Laboratory Press.

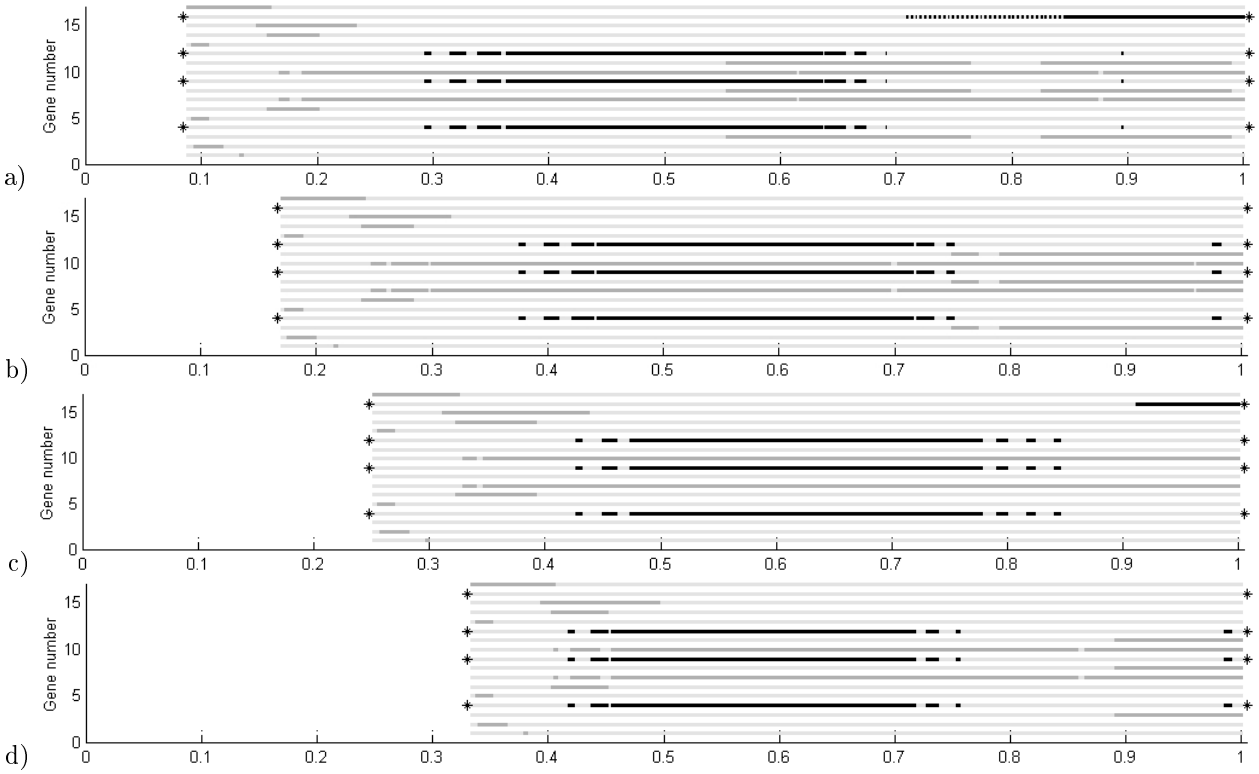


Figure 6: **Gene expression patterns for four units.** Dark gray and light gray bands correspond to periods of gene activity and inactivity, respectively. Four genes are marked by asterisks; the expression pattern of these genes is similar in units a) and c), but different in units b) and d). The expression times of these genes are darkened for clarity. Genes that are always on or always off during ontogeny are not shown. Note the evolved gene families, which have similar expression patterns.

S. A. Kauffman (1993). *The Origins of Order*, Oxford, UK: Oxford University Press.

M. Kirschner and J. Gerhart (1998). Evolvability. In *Proc. Nat. Acad. Sci* **95**:8420-8427.

M. Komosinski, and S. Ulatowski (1999). Framsticks: Towards a simulation of a nature-like world, creatures and evolution. In: *Proceedings of 5th European Conference on Articial Life*, 261–265. Springer-Verlag.

E. B. Lewis (1978). A gene complex controlling segmentation in Drosophila. In *Nature* **276**:565-570.

H. Lipson and J. B. Pollack (2000). Automatic design and manufacture of artificial lifeforms. In *Nature* **406**:974–978.

S. Ohno (1970). *Evolution by Gene Duplication*. New York: Springer Verlag.

R. A. Raff (1996). *The Shape of Life*. Chicago: The University of Chicago Press.

T. Reil (1999). Dynamics of gene expression in an artificial genome–implications for biological and artificial ontogeny. In *Proceedings of the Fifth European Conference on Artificial Life*, 457–466. Springer-Verlag.

R. Riedl (1978). *Order in Living Organisms: A Systems Analysis of Evolution*. Chichester: John Wiley & Sons.

A. Rotaru-Varga (1999). Modularity in evolved artificial neural networks. In *Proceedings of the Fifth European Conference on Artificial Life*, 256–260. Spring-Verlag.

K. Sims (1994). Evolving 3D morphology and behaviour by competition. In *Artificial Life IV*, 28–39. MIT Press.

D. Terzopoulos, T. Rabie and R. Grzeszczuk (1996). Perception and learning in artificial animals. In *Artificial Life V*, 313–320. MIT Press.

J. Ventrella, (1994). Explorations of morphology and locomotion behaviour in animated characters. In *Artificial Life IV*, pp. 436–441. MIT Press.

G. P. Wagner (1995). Adaptation and the modular design of organisms. In: *Advances in Artificial Life*, 317–328. Springer Verlag.

G. Wagner and L. Altenberg (1996). Perspective: Complex adaptations and the evolution of evolvability. In *Evolution* **50**(3):967–976.

On the influence of learning time on evolutionary online learning of cooperative behavior

Jörg Denzinger
Computer Science Department
University of Calgary, Canada

Michael Kordt
Fachbereich Informatik
Universität Kaiserslautern, Germany

Abstract

We present an online learning approach for learning cooperative behavior in multi-agent systems based on invoking an offline learning method as a special action "learn". We apply this approach to evolutionary offline learning using situation-action-pairs and the nearest-neighbor rule as agent architecture. For the application Pursuit Games we show that the online approach using evolutionary offline learning allows for good success rates for rather different game variants. Particularly, we perform experiments highlighting the influence of the time needed for learning and of the parameters of the evolutionary offline method.

Our results show that even a duration of "learn" which is several times longer than the usual duration of an agent's actions still achieves good success rates. The same applies to rather small values for the key parameters of the offline method. Together, this suggests that this evolutionary online learning approach is a very good alternative to the well-known online approaches based on reinforcement learning.

1 Introduction

Achieving cooperative behavior of a group of agents either requires a careful planning, design and implementation of the agents by human developers or can be tackled by using learning techniques. Learning can either be used in an offline manner, which means that the learning phase is separated from (i.e. before) the real application phase of the agents, or it can be online, in which case the agents try to adapt their behavior during their work on the real application. Evolutionary approaches have already proven to be quite successful for offline learning of cooperative behavior of agents, as has been demonstrated in Manela and Campbell (1993), Hayes et al. (1995) or Denzinger and Fuchs (1996). In these approaches the behavioral patterns of all cooperating agents were evolved together and the resulting agents were not capable of learning for themselves, they just consisted of the application specific strategies that then later were applied to solve the problem. The cited papers show that even a certain flexibility of these strategies, to deal with random effects, is achievable.

For online learning of agents, not many concepts involving evolutionary methods have been proposed so far. The dominating approach has been to employ reinforcement learning techniques, i.e. associating with each possible action in each possible situation a certain weight describing the usefulness of the action in the situation (see Watkins (1989), Weiß (1995) or Tan (1993)). As in case of evolutionary algorithms, some random decisions are needed in order to realize the necessary exploration of alternatives during solving the given task. From the point of view of online learning, at first glance the use of evolutionary methods with the need to evaluate a lot of different individuals that are the result of the evolutionary process seems to promise performance problems, because many individuals will not represent solutions and trying to evaluate their performance might gravely endanger the solving of the given task (that has to be interleaved with the learning process in order to be an online approach). We will see that this does not have to be the case.

In this paper we will use the offline learning approach presented in Denzinger and Fuchs (1996) together with the idea of an action "learn" and modeling of other agents to get an online learning approach for cooperative behavior of agents. An online learning agent will monitor and remember all occurring situations and the

actions the other agents performed in these situations in order to construct models of these other agents. It will also periodically perform the action "learn" that results in the agent invoking the offline learning approach. This offline learning approach evaluates the fitness of agent strategies based on a simulation of the real application and the success of the particular agent strategy in this simulation. For online learning, the simulation starts from the situation the learning agent anticipates to be in after learning and uses the models of the other agents to predict their behavior.

We applied this online learning approach to variants of the well-known Pursuit Game and showed that the approach is capable to solve instances that are not solvable by offline learning (see Denzinger and Kordt, 2000). But before we can apply the approach to more complex application areas, a more precise study of the influence of the time needed for performing "learn" relative to the execution time of "standard" actions is needed. And then some experimental evaluation is necessary to determine if the parameter values of the offline learning approach – like number of individuals and generations, or length of the simulation – that are needed to achieve sensible execution times for "learn" still result in a sufficient learning quality.

The experiments we present in this paper show that indeed the relation between the time needed for performing "learn" and the execution time of other actions plays an important role for success, and too much time needed for "learn" will lead to too much uncertainty so that learning is not sufficiently successful. On the other side, we can show that already relatively small values for the parameters mentioned above achieve fine learning results, so that the presented approach to evolutionary online learning is a good, even superior alternative to the non-evolutionary online learning approaches.

2 Online Learning by Offline Learning

Offline learning of cooperative behavior of a group of agents is intended to produce *strategies* for all or some of the agents involved in performing the task to achieve a certain common goal. Offline learning takes place before the agents actually perform this task and usually involves the exploration of many possible strategies, their evaluation and then their optimization. During the exploitation of these learned strategies, i.e. the actual tackling of the task, no more learning takes place. So, we can see offline learning as a kind of selection process to find the strategies for the individual agents that, if applied together, achieve the intended common goal. In this sense, the resulting agents themselves do not have any learning capabilities.

In contrast, *online learning* agents are in the process of performing tasks towards achieving a certain goal. Therefore learning has to be integrated into these tasks and combined with the actions that are taken. As a consequence, timing of actions and learning becomes a very crucial issue, because there can be actions that are not reversible. And if later learning leads to the knowledge that such an action should not be performed in a certain situation, the goal might already be not reachable anymore. So, the balance between exploration and exploitation of the learned knowledge is very crucial and online learning of cooperative behavior of agents is a very difficult task (if the problem to be solved is not very simple).

2.1 Our General Method

In general, an agent can be described as a function f_{Ag} that maps situations and the internal state of the agent to an action (or action sequence) that the agent is capable to perform. The internal state of an agent contains all its knowledge about the environment, other agents and itself. This can include goals, plans or observations. Note that from outside the internal state usually cannot be observed, so that an observer sees an agent only as a function $f_{Ag,Obs}$ mapping situations to actions. Also actions might include components that manipulate the internal state of an agent, and these components also cannot be observed from outside.

The result of offline learning is such a function f_{Ag_i} for each agent Ag_i that was included into the learning process. In contrast, the function f_{Ag} of an online learning agent contains a component that is responsible for learning (and such a component cannot be identified in the f_{Ag_i} resulting from offline learning). Our idea is to realize this component by invoking the offline learning approach and to trigger the use of the component by introducing a new action "learn". The offline learning approach is based on a simulation of the real environment the agent is acting in, its most important part being the simulations (or *models*) of the other agents. These models are either derived from the agent's observations of their behavior or from information communicated by the other agents to the online learning agent.

More precisely, an online learning agent performs the following cycle:

1. acting in the environment (*real world*) according to its current strategy for a given amount of time or until its success is obviously not good enough
2. performing the action "learn":

(a) generate/update the models of the other agents
(b) determine the situation s_{after} after performing "learn"
(c) run offline learning approach with s_{after} as start situation and the models of the other agents
(d) combine best strategy found by offline learning with current strategy to get a new one

There are several ways to realize each of these steps, even for a given agent architecture and a given offline learning approach. Problems to be solved are, for example, how to determine success during the application of the current strategy in the real world, or how to generate the models for the other agents. There are also many possibilities for generating a new strategy, from simply replacing the old strategy with the learned one over somehow putting old and learned strategy together to just keeping at the old strategy (if the learned one does not seem good enough).

In addition to the possibilities and parameters of this general online learning method, there are also the parameters of the offline learning approach that influence the online learning. Obviously, the time needed for offline learning will also influence online learning, because this time must be known before performing offline learning in order to determine s_{after}. Therefore the offline learning method should be realized as any-time algorithm (see Boddy and Dean, 1988). And among the parameters of the offline learning method should be the length (in time units) of the simulations used by it.

2.2 Evolutionary Learning with SAPairs

If we look at the requirements for the offline learning method in 2.1, then evolutionary algorithms in general already provide the any-time property. The other requirements are met by the approach presented in Denzinger and Fuchs (1996) that is based on prototypical situation-action-pairs (SAPs) and the nearest-neighbor rule (NNR) as agent architecture and a genetic algorithm with sets as individuals and evaluation of simulation runs as fitness measure.

More precisely, an agent consists of a function f_{Ag} that is based on a set of SAPs. A situation is represented by a vector of numerical values that describe the environment the agent is acting in (including information about the other agents) and also the necessary information about the internal state of the agent. This means that a situation in a SAP may contain more information than the situations mentioned in 2.1. The actions in a SAP are the actions that the agent is able to perform (without the action "learn") or sequences thereof.

The agent determines its next action by measuring the distance between the actual situation and the situations of all its SAPs. A possible distance measure is the Euclidean distance, but also other measures can be used. Then it performs the action of the SAP with the smallest distance. This is the well-known nearest-neighbor rule.

This agent architecture is very well suited for learning, because the strategy of an agent can be easily changed by adding or deleting SAPs or just changing components of an SAP. From the point of view of evolutionary algorithms, the agent architecture also has many advantages, as we will see. In Denzinger and Fuchs (1996), a genetic algorithm was used to evolve sets of agent strategies, i.e. a strategy (which is a set of SAPs) for each agent that needed to learn one. These strategies, when applied together, should achieve a good cooperative behavior of the agents. The used genetic operators were picking random SAPs out of the strategy for one agent of two individuals as crossover and generating a random SAP or deleting a SAP out of a set as mutation.

For determining the fitness of an individual (which describes a team of agents), the agents are put to solving their task (in a limited simulation). If the agents are able to succeed, the fitness is the number of time units needed. If the task is not accomplished then for each time unit t of the simulation a measure of task fulfillment $m_{task}(t)$ is computed and these measures are summed up. If there are random factors involved in the simulations then the mean fitness value over several different simulation runs is taken as the fitness of an individual. Important parameters of this offline learning method are the maximal number sap_{max} of SAPs of an agent, the maximal number T_{max} of time units in a simulation, the number b of simulation runs that go into determining the fitness of an individual and, as usual for a genetic algorithm, the number of individuals in a population and the maximal number of generations.

Using this offline learning approach for online learning, we have to learn a strategy for one agent only. Also, the length of a simulation run can be much shorter than in the offline case (as we will see in our experiments). We might also use a lower sap_{max} value. The measures of task fulfillment $m_{task}(t)$ can be used to determine the success of a strategy in the real world, by assuming that the strategy is successful as long as $m_{task}(t_i)$ gets better from some t_i to some t_{i+k}. Com-

bining strategies realized by SAPs is rather easy by just combining the sets. Finally, SAPs can also be easily generated out of an agent's observations of the other agents. Therefore SAPs (together with NNR) provide a very good way to model the other agents.

3 The OLEMAS System

The OLEMAS system (**O**n**L**ine **E**volution of **M**ulti-**A**gent **S**ystems) instantiates the methods described in Section 2 for the application area Pursuit Games. In the following, we will first take a closer look on this application area and then we will describe OLEMAS in more detail.

3.1 Pursuit Games

The original Pursuit Game was described in Benda et al. (1985) as four dot-shaped hunter agents trying to surround and immobilize a dot-shaped prey agent on an infinite grid world without any obstacles. All agents could only perform movements in the horizontal and vertical directions at the same speed (one square per time unit). Every agent could see all the other agents and this was the only kind of "communication" between the agents. The goal of the game was to develop a set of hunter strategies to win the game against a prey choosing its moves randomly.

While this original game is considered a toy problem these days (although a difficult one), many variants have been developed that vary numerous features of the game. A large collection of these features and possible instantiations can be found in Denzinger and Fuchs (1996). We implemented the possibility to vary all the features mentioned there in OLEMAS and added some additional possible instances. For example, we do not only have fixed obstacles, but also agents that play the role of innocent bystanders (or moving obstacles).

Among the features that raise Pursuit Games from simple toy problems to abstractions of real world high-level robot control problems are the shape, speed, and possible actions of an agent. By letting agents occupy several squares of the grid representing the world we introduce additional actions, namely turns, and the difficulty of not only being in a certain position but also needing a certain orientation. Assigning to each action of each agent an execution time (as multiples of a basic time unit) also accounts for more reality and often more difficulties in achieving the common goal. The same applies to having the bystanders that accidentally can help or hinder both hunters and prey (and that allow for game variants in which predicting their behavior is essential for winning the game). Naturally, bystanders and prey have their own strategies and the different possibilities here already produce an infinite number of variants. Other features include the number of agents of all types, the start situation and the definition of what is a winning situation.

3.2 The System

OLEMAS is implemented in C++ and uses Tcl/Tk and Tix for visualizing the pursuit games. The system architecture is depicted in Figure 1. Basis for the agent architecture are the situation vectors and the distance measure. A situation vector of an agent contains each other agent's coordinates relative to the agent's position (in a fixed order) and its orientation. If a situation s_j contains the coordinates x_{ij} and y_{ij} of agent i, $1 \leq i \leq$ n, and its orientation o_{ij} (0 = north, 1 = east, 2 = south, 3 = west), then the distance d to situation s_k is computed as

$$d(s_j, s_k) = \sum_{i=1}^{n}((x_{ij} - x_{ik})^2 + (y_{ij} - y_{ik})^2 + ((o_{ij} - o_{ik})^2 \text{mod } 8)).$$

The contribution of the orientation of the agent has to be taken mod 8 in order to treat different orientations clockwise and anticlockwise the same. Without this modification the difference $o_{ij} - o_{ik}$ between o_{ij} = north and o_{ik} = east would not be the same as the difference between north and west, which is not our intention. Since we use the square of the distance, the results must be modified by mod 8.

For the Evolutionary Learning Component, we have to define the fitness of an individual, more precisely, we have to define the function m_{task}:

$$m_{task}(t) = \sum_{i=1}^{n} \delta(i, t),$$

where $\delta(i, t)$ is the Manhattan distance that separates hunter i from the prey at time unit t (or the sum of the distances to all prey agents if there are more than one).

OLEMAS is intended as experimental system to evaluate all the possible influences on online-learning by offline learning. Therefore we tried to allow for changing as many parameters and components as possible. In addition to the learning time aspect that will be the subject of the experiments in the next section and the other parameters already mentioned, we allowed for the easy addition of other agent architectures and strategies, for filtering of all the information accumulated during a run (influencing the modeling of

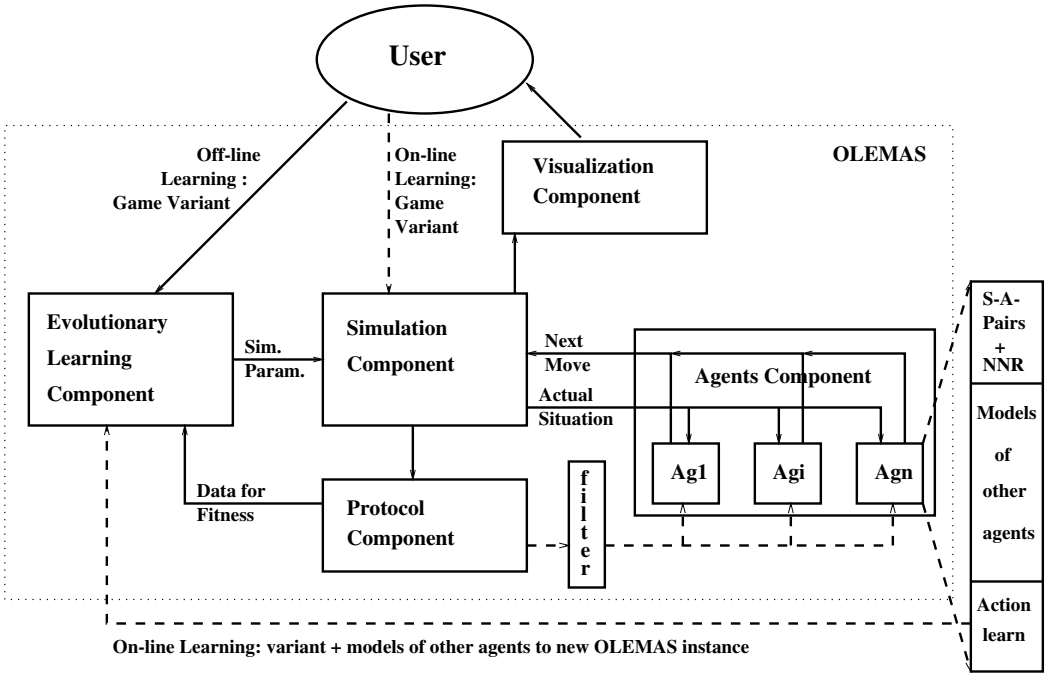


Figure 1: OLEMAS: system architecture

other agents) and even for changes in the Evolutionary Learning Component.

When having online learning agents, the user defines a game variant for the Simulation Component (the "real world"). With this variant, also the online learning agents are identified and an initial strategy for each of them is given (either predetermined, totally at random, or the result of performing "learn" as first action). Then the agents act in the real world until the first online learning agent has to "learn". This is either after a fixed number of time units or in between a minimum and maximum number of time units, determined by the performance (see Denzinger and Kordt, 2000, for details). The learning agent then generates SAP-based models for all other agents for which it does not have more accurate information (i.e. for all agents that do not communicate their real strategy to it). To do so, it generates a SAP for each situation observed so far from the perspective of the other agent (using the protocol generated by the Protocol Component; filtering will be used in future experiments). Then the models, the current situation and the other parts of the variant being the real world are passed as a game variant to a new instance of OLEMAS, more precisely to the Evolutionary Learning Component of it. There, first the simulation is started to determine the situation at the end of executing "learn" and then offline learning takes place. The best strategy found will then be the new strategy for the online learning agent.

4 Experiments

While the experiments described in Denzinger and Kordt (2000) concentrated on the suitability of the general method for online learning of cooperative behavior and established the advantages of this method, we want to concentrate in this paper on the suitability of evolutionary algorithms as basic underlying learning, resp. search, method. Since "learn" is an action, a learning agent performing it obviously cannot (and should not) do anything else, which kind of takes this agent out of the game during learning. So, a first important question is how much time an agent can spend on learning while still being able to cooperate with the other agents in achieving the goal of the game. Obviously, here the relation of (game) time units spent on learning and units necessary for the other actions is of interest. OLEMAS allows us to define the length of learning with respect to the real world game freely and independent from the computing time needed. We examine this relation in our first experimental series.

Although OLEMAS allows us to have two times for learning (game time and processing time needed), for later applications it is very important to know if the processing time can be limited sufficiently, so that, for example, a robot can have an action "learn" while acting in the real world. This leads to investigating the parameter values that mainly influence how much processing time is needed to perform "learn", namely the maximal number sap_{max} of SAPs in an individual, the number of individuals in a generation, the number of

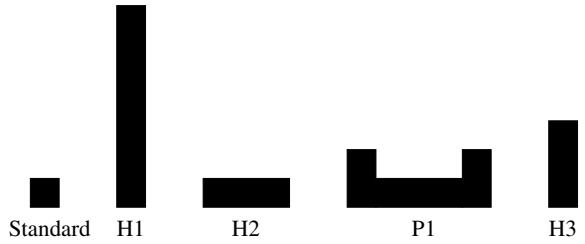


Figure 2: Shapes of the used agents

generations, and the maximal number T_{max} of time units to be simulated for determining the fitness of an individual. We examine the influence of these parameters in the experiments of the second series.

4.1 The game variants

The three game variants we selected for our experiments require rather different strategies from the hunters to be successful, and in all variants random factors (in the form of random start positions for the agents) are involved, so that two runs of OLEMAS with the same variant usually differ very much. In all variants we have only one online learning hunter, because we are interested in the learning time in general (for multiple learning agents, see Denzinger and Kordt, 2000). One single experiment for a variant always consisted of 100 real world runs and as success we will report the percentage of successful runs within the given time limit. In all variants, the time needed for any action except "learn" was one time unit and the possible moves of the agents are up, down, left, right, stay put, and, if the shape makes turns useful, turn left and turn right (for 90 degrees around the agent's predefined center point). The grid size was 30 $\times$ 30 squares for all variants and there are no obstacle agents.

In variant 1 we have two hunters and one prey. The prey tries to maximize the distance to the nearest hunter and it occupies one square only. The first hunter has the shape of H1 in Figure 2 and uses a fixed strategy, namely trying to come as close to the prey as possible. The learning hunter has the form of H2 in Figure 2. The goal situation of this variant is to immobilize the prey.

In variant 2 we have 4 hunters and 4 preys, all of them occupying one square only. The prey agents use the same strategy as the prey in variant 1. While three of the hunters use the strategy of the non-learning hunter in variant 1 (forming a pride), the fourth hunter is a learning one. The goal situation is that each hunter ocuppies the same square as a prey (no two prey agents are allowed to occupy the same square), and whenever

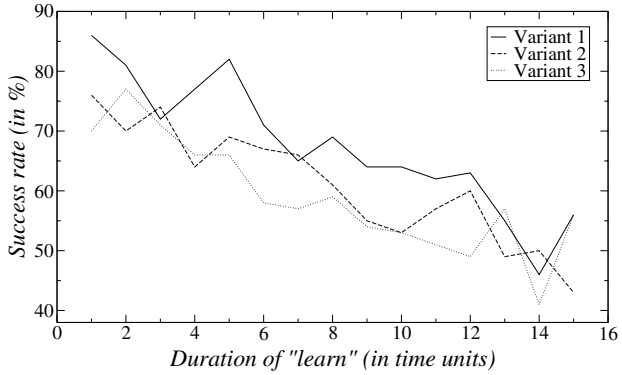


Figure 3: Influence of duration of "learn" on success

a prey is caught in this sense both prey and hunter do not move any longer.

In variant 3 we have 2 hunters and 1 prey, again. The prey has the shape of P1 in Figure 2. The non-learning hunter has standard form and uses the strategy of the non-learning hunter of variant 1. The learning hunter has the form of H3 of Figure 2. As in case of variant 1, the goal situation is to immobilize the prey which tries to stay away from both hunters and boundaries, thus complicating the hunters' task.

4.2 Series 1

The number of time units spent on learning in contrast to the execution time of the other actions obviously should have some influence on the outcome of a game run. This becomes immediately clear if we are considering game variants in which agents move randomly or in which the learning agent relies on models of other agents based on observations, thus representing unsure knowledge. Since the simulation runs start with the situation that the learning agent expects to face after having executed "learn", unsure knowledge or random factors result in an also unsure prediction of this situation. And the longer the learning takes the more unsure is the prediction.

But also if we have game variants for which we can reliably predict the start situation for the simulations, we expected the execution time for "learn" to have an impact on the success of hunter teams including an online learning agent. As Figure 3 shows, we were right to expect this. All three variants we described above allow a precise prediction of the situation the agent will face after learning. Our experiments show for all variants some decline in the success rate. But this decline is rather gracious – using 4 to 5 times the time for learning than for other actions achieves definitely still acceptable results. Even when needing 10 to 12 times the time we are still above 50 percent success

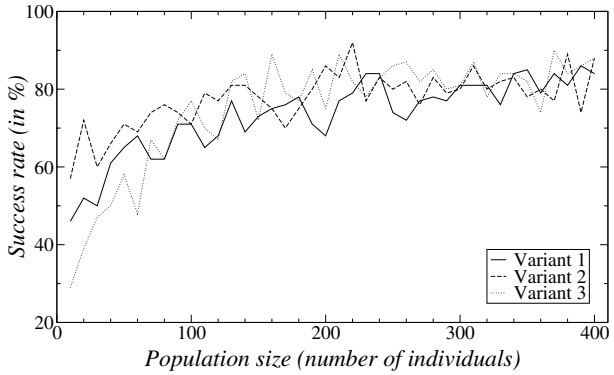


Figure 4: Influence of population size on success

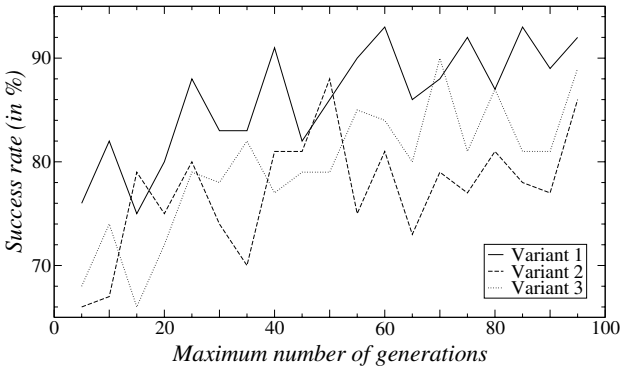


Figure 5: Influence of number of generations allowed on success

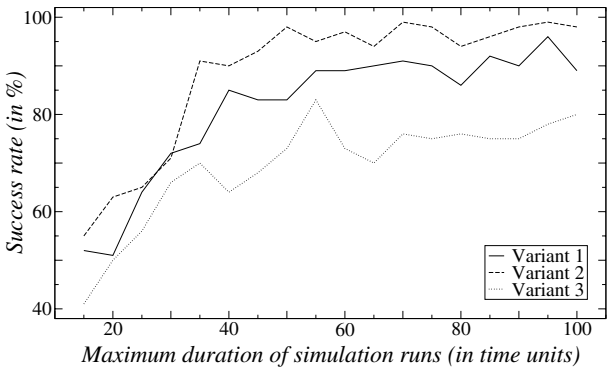


Figure 6: Influence of simulation length on success

rate in all variants. If we consider that allotting more time units for the real world game usually results in higher success rates (see Denzinger and Kordt, 2000), then these results indicate that our evolutionary online learning approach is feasible.

4.3 Series 2

The experiments of series 1 were performed with the following settings for the parameters that influence the processing time needed to execute "learn": The population comprised 300 individuals in variant 1 and 100 individuals in all other cases. We defined $sap_{max} = 40$ for variant 1, $sap_{max} = 20$ otherwise. The number of generations was limited to 15 in variants 1 and 2 and restricted to 10 in variant 3. Furthermore, we defined $T_{max} = 35$ in variant 1, $T_{max} = 30$ in variant 2 and $T_{max} = 50$ in variant 3. Although our experiments so far show that learning can take some time without reducing the success rate very much, they also show that different variants can differ quite a lot in this regard (which we expected and which led to the different initial parameter values for the different variants). Therefore we examined the influence of the parameters mentioned above on the success rate in order to see how processing time can be reduced. Obviously, reducing the value of each of these parameters reduces the amount of computation necessary for "learn" and therefore the needed processing time.

As Figure 4 shows, relatively small populations already achieve rather good success rates. And an increase in the number of individuals (beyond 80 to 100 individuals) does not increase the success very much. Since the number of individuals that are generated during "learn" has a very large influence on the needed processing time, this is already a very promising result with respect to using our learning approach in more realistic applications. The other parameter influencing the number of individuals generated during a simulation run is the maximum number of generations. The influence of this parameter is depicted in Figure 5. Already with 25 generations all variants show a success rate above 70 percent. Then we have rather large variations for each variant, so that more than 40 generations seem to be redundant in either case.

The more time units are allowed in the simulation runs the more processing time is needed to evaluate the fitness of an individual, i.e. an agent strategy. Also, the further these simulations look into the future the more online learning changes towards offline learning. As Figure 6 shows, the longer we plan into the future the better is our chance for success (although we do not have a steady increase). But 30 time units already achieve a success rate of over 60 percent for all variants. And for variants 1 and 2 going above 40 time units does not increase the success much more. The relatively bad results for variant 3 are due to the fact that this variant allows for many situations in which both hunters are directly near the prey without having it immobilized, so that the m_{task}-values for these situations are rather good. While this affects all experiments with this variant, especially longer simulation runs suffer from the not very selective fitness values that are generated for this variant.

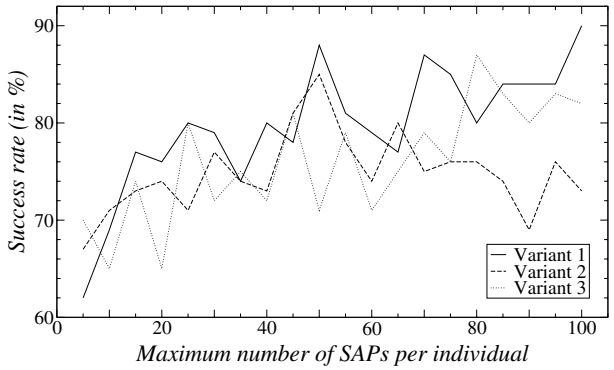


Figure 7: Influence of possible complexity of strategy on success

The last parameter we want to look at influences the processing time needed for executing "learn", because the more SAPs are in a strategy, the longer it takes to decide what action to perform in a situation, since the similarity to each pair from the actual situation has to be computed. But the number of SAPs per strategy also determines how complex the strategies can be. If there are too few SAPs, then each possible strategy might be too simple. But if there are too many SAPs then the possibility is much higher to have unnecessary or even disturbing pairs. As Figure 7 shows, the influence of the complexity of the possible strategies on the success rate is not easy to describe and the different variants show rather different results. 25 to 35 SAPs achieve for all variants high success rates and already with only 5 SAPs we are above 60 percent success rate. But all variants show large variations in their results, with variant 2 definitely having decreasing success as a trend for larger numbers of SAPs. But this is not exactly surprising. In variant 2 the goal is to catch several preys and with a high number of SAPs our evolutionary approach leads to learning SAPs for catching each of these preys. In the simulations this is a good trait, because, due to the random factor involved, the fitness is evaluated by several runs. But in the real world the additional pairs make the hunter less efficient.

5 Conclusion

We presented an online learning approach for achieving cooperative behavior of agents. It is based on introducing an action "learn" that executes an offline learning method. In our case this method is the evolutionary approach presented in Denzinger and Fuchs (1996) based on SAPs and NNR. In this paper, our experimental evaluation concentrated on the relation of the execution times needed for "learn" and the other actions. In the area Pursuit Games we showed for rather different game variants that learning times that are several times longer than the times needed for performing other actions still achieve good success rates (for fixed game lengths). We also investigated different values for the key parameters of the evolutionary offline learning method and found that already rather small values of these parameters achieve good success rates, again. Since small values for these parameters result in less processing time needed for "learn", our experiments suggest that our evolutionary online learning approach can be successfully used in more realistic applications, like cooperating robot teams.

Before tackling such new application areas, in future work we want to investigate several other problems and phenomena with our Pursuit Games application. Among them are improving the evolutionary learning by getting more information out of the simulation runs, co-evolution of hunters and preys, and testing the adaptation capability of online learning agents when becoming member of already good cooperating teams. Other future work will be to improve on the modeling of other agents and to investigate how to better adapt the intervals between performing "learn". The latter also focuses on making more use of seeing on- and offline learning as extremes of a spectrum of possibilities instead of different kinds of learning tasks, which is suggested by our results.

6 References

M. Boddy and T. Dean (1988). An Analysis of Time-Dependent Planning, Proc. 7th AAAI, pp. 49–54.

M. Benda, V. Jagannathan and R. Dodhiawalla (1985). An Optimal Cooperation of Knowledge Sources, Technical Report BCS-G201e-28, Boeing AI Center.

J. Denzinger and M. Fuchs (1996). Experiments in Learning Prototypical Situations for Variants of the Pursuit Game, Proc. ICMAS'96, pp. 48–55.

J. Denzinger and M. Kordt (2000). Evolutionary On-line Learning of Cooperative Behavior with Situation-Action-Pairs, Proc. ICMAS-2000, IEEE Press, pp. 103–110.

T. Haynes, R. Wainwright, S. Sen and D. Schoenefeld (1995). Strongly typed genetic programming in evolving cooperation strategies, Proc. 6th Intern. Conf. on Genetic Algorithms, Morgan Kaufmann, pp. 271–278.

M. Manela and J.A. Campbell (1993). Designing good pursuit problems as testbeds for distributed AI: a novel application of genetic algorithms, Proc. 5th MAAMAW, pp. 231–252.

M. Tan (1993). Multi-agent reinforcement learning: Independent vs cooperative agents, Proc. 10th Machine Learning, Morgan Kaufmann, pp. 330–337.

C.J.C.H. Watkins (1989). Learning from Delayed Rewards, PhD thesis, University of Cambridge.

G. Weiß (1995). Distributed Machine Learning, Infix-Verlag, Sankt Augustin.

Modular Designer Chemistries for Artificial Life

Keith L. Downing
Department of Computer Science
The Norwegian University of Science and Technology (NTNU)
7020 Trondheim, Norway
Tele: (+47) 73 59 18 40
Email: keithd@idi.ntnu.no

Abstract

The vast majority of artificial life (alife) systems lack underlying chemical models that exhibit both a) realistic relationships between biomass production/consumption and energy, and b) strong restrictions on the legal combinations of atomic units. In many cases, biomass is simply accumulated resource, occasionally paid for by an energy tax. This suffices for many purposes, but when the alife system serves as a testbed for microbiological investigations, more realism is desirable. However, real biochemistry is so complex that one naturally turns to the alife discipline of abstract/artificial chemistries for rich, yet manageable chemical backdrops; but with little success.

This paper introduces MD-CHEM, a simple algorithm for generating random chemistries that a) meet user specifications, b) capture important biomass-energy relations, and c) easily plug into alife simulators. Runs of these chemistries in our METAMIC simulator illustrate how a) organisms evolve to exploit the energetic potential of the chemistry and environment, and b) chemistries restrict population density and diversity.

1 Introduction

There exists a sizeable gap between research into artificial chemistries (AC) [7, 16, 3, 2] and artificial-life simulations of higher-level systems such as cells [8] organisms [15] and ecosystems [9, 5]. In most cases, the artificial chemistry *is* the artificial-life system, exhibiting fascinating emergent structures such as autocatalytic sets [11, 10], metabolisms[1] and other interesting molecules and reaction dynamics [7, 4]. Unfortunately, simulations at the chemical level are often time-consuming, making the inclusion of full-scale AC processes in populations of hundreds or thousands of cells rather infeasible.

Higher-level alife systems typically avoid chemistry completely by relying on a simple mapping between environmental food sources and the agents' internal energy reserves, with the biochemical details of that conversion being largely irrelevant.

However, when alife moves to the microcosmic level, biochemistry becomes an essential issue for exploring many interesting phenomena. Unfortunately, biochemical interactions are highly complex, making the simulation of essential metabolic chemistry into a large-scale research project of its own. Therefore, there is an acute need for computationally-efficient artificial chemistries at this level: modular systems that embody the essence of biochemistry and enable the emergence of metabolically diverse individuals, but without the overhead of full-scale emergent AC simulations running in thousands of cells.

One possible solution is to run a conventional AC until it reaches equilibrium or stable oscillatory behavior and then extract the above-threshold compounds and frequently-occurring reactions, filling in gaps to insure closure. While theoretically possible, the practical feasibility of this approach is questionable, since ACs run from scratch often produce either a) only a few stable surviving compounds, or b) a very large set of compounds and reactions, any subset of which will not form a closed reaction topology. In short, we cannot guarantee that the emergent AC will generate anything of practical use.

Furthermore, most AC systems, with a few notable exceptions [13], ignore energy constraints. Hence, they abstract away two essential aspects of biochemistry: a) energy yields of reactions, and b) relationships be-

tween energy and structure/biomass formation. Energetic arguments are crucial to understanding the emergence of *biochemical guilds*, i.e., groups of microorganisms with diverse (often complementary) metabolisms, which are the main focus of our alife research [6]

We now introduce an alternative artificial chemistry, the Modular Designer Chemistry (MD-CHEM), which includes a very simple energy scheme and ties it to biomass production, thereby supporting emergent biochemical phenomena at the microbial guild level while avoiding complex simulations at the molecular level.

In a nutshell, MD-CHEM generates random closed reaction sets of user-specified size and uses statistical entropy changes to determine energy yields. The complete set (or any subset) can then be easily incorporated as a metabolism module into a simulated biochemical agent; populations of these agents, often with diverse metabolisms, can then interact chemically via diffusive exchanges with a common environment.

This supports various *what-if* or *play-the-tape-again* alife experiments in which the evolutionary processes are rerun with different underlying chemistries. If similar patterns continue to emerge, then general explanations may exist that subsume those involving the vagaries of *biochemistry as we know it.*

This paper describes the basics of chemistry generation in MD-CHEM, followed by examples of chemical-module usage in our Metabolizing Abstract Microorganism (METAMIC) system.

2 MD-CHEM Basic Mechanisms

MD-CHEM generates artificial chemistries by ignoring atomic-level dynamics and choosing random regroupings of random reactant sets, constrained only by user-specified parameters such as the desired numbers of compounds and reactions. Hence, the system is only weakly constructive, since new compounds arise stochastically, but not on the basis of any first-principle physical relationships between the atoms, such as their potentials for bond formation. So chemistry construction in MD-CHEM is a purely algorithmic (albeit non-deterministic) process, guided not by an interaction dynamic, but by simple trial-and-error search for a constraint-satisfying set of compounds and reactions. The interesting artificial life can occur at the next level, when an MD-CHEM module forms the basis for interactions between metabolizing cells.

The *designer* aspect of MD-CHEM is quite simple: a user specifies several hard constraints and biases, and the system then generates a chemistry that meets the specification and easily plugs into a higher-level alife simulator.

The key hard constraints are:

1. N_a - the number of legal atoms,
2. N_c - the total number of compounds to be generated,
3. N_r - the total number of reactions to be generated,
4. N_{ic} - the number of initial randomly-generated compounds,
5. R_{cs} - the range of legal compound sizes
6. R_{rs} - the range of reaction sizes, in terms of the minimum and maximum number of reactants/products.

The essential biases are:

1. B_{ics} - the bias of initial compound sizes. This is a normalized list of probabilities, one for each legal size in R_{cs}, which governs stochastic size choice of the initial N_{ic} randomly-generated compounds.
2. B_{rs} - the bias of reaction sizes, a probability distribution for the sizes in R_{rs}, which affects the stochastic choice of both a) the number of reactants chosen for each random reaction, and b) the number of product compounds that the reactant atoms are partitioned into.

The basic algorithm for chemistry generation appears in Figure 1.

For example, if the atomic set is {a b c}, $R_{cs} = [3, 6]$, $R_{rs} = [2, 4]$, and $N_{ic} = 4$, then MD-CHEM begins by generating 4 initial compounds, such as a_2b, abc, ac_2, c_3, where subscripts denote the number of each atom and no subscript implies a single atom.

The reaction generator would then take between 2 and 4 of these compounds, with possible duplicates, such as {a_2b, ac_2, ac_2}, to form the reactant group. All atoms are then thrown into a set: {a a b a c c a c c} which is sent to REACT-COMP, where it is randomly permuted: {c b a c a c a a c}, and partitioned into 2 to 4 subsets: {c b a ! c a ! c a a c}.

Since no inter-atomic relationships are modelled, MD-CHEM is insensitive to the order of atoms within a compound and converts it to a canonical form. Hence, the product group for the above example becomes {abc, ac, a_2c_2}. Since the latter 2 compounds are

```
Program GENCHEM
C = { }; The set of compounds
R = { }; The set of reactions
n_c = 0; The size of C
n_r = 0; The size of R
RG, PG: reactant  product groups.
Initialize C to N_ic randomly-generated compounds.
num-attempts = 0
Repeat
 RG = random group of compounds from C
 Collect all atoms from RG into one set, S
 p = e^((-k_e(N_c - n_c))/N_c)
 known-compound = TRUE
 PG = { };
 n_p = 0; Number of products
 N_p = random-integer-in(R_cs)
 While known-compound and random-number ≤ p
          and n_p < N_p
  known-compound = find-known-compound(S).
  if known-compound
      Add known-compound to PG.
      n_p = n_p + 1
  Remove atoms in known-compound from S.
 End while
 If REACT-COMP(S,C,R,RG,PG,N_p - n_p)
  Then Add [ RG → PG] to R, and update n_r
  Else num-attempts = num-attempts + 1
Until n_r = N_r or num-attempts ≥ max-attempts
End Program


Procedure REACT-COMP(S, C, R, RG, PG, NP)
 ; This procedure randomly completes reactions
 Randomly permute S.
 PG' = PG
 Randomly break S into NP groups
      and add each group to PG'
 If
  All groups in PG' either:
      Form a pre-existing compound, or
      Form a new compound that can be
           added without n_c exceeding N_c
  And PG' ≠ RG And [PG' → RG] ∉ R
  Then
      PG = PG'
      Add newly-generated compounds in PG' to C
      Update n_c
      Return TRUE.
  Else Return FALSE.
End Procedure
```

Figure 1: Algorithmic overview of MD-CHEM

new, MD-CHEM will only add the new reaction if $N_c - n_c \geq 2$.

The new reaction is then:

$$a_2b + 2ac_2 \Longrightarrow abc + ac + a_2c_2 \qquad (1)$$

The use of the num-attempts and max-attempts variables in Figure 1 indicates that GENCHEM makes a finite number of attempts to generate N_r reactions. Note that once $n_c = N_c$, any reaction that generates additional new product compounds will be rejected in REACT-COMP . In cases where $N_r > 2N_c$, or even $N_c \leq N_r \leq 2N_c$, the algorithm will occasionally time out before generating N_r reactions.

To remedy this situation, GENCHEM attempts to extract known compounds from the list of product atoms prior to calling REACT-COMP, thus decreasing the number of new compounds introduced by each reaction. It uses the probability p to determine whether to try to extract one or more known compounds from S before randomly permuting and partitioning the remaining product atoms. In the calculation of p, k_e is the extraction factor, with a typical value between 1 and 2, with lower values implying a higher extraction probability. A high value of k_e is preferable when $N_c \approx N_r$, but it should decrease as $N_r - N_c$ rises.

Regardless of k_e settings, the initial size bias, B_{ics}, is a critical success factor. If the initial compound set contains too many large compounds, then generating N_r legal reactions becomes quite difficult, particularly when R_{cs} and R_{rs} are small (i.e., tight ranges of legal reaction and compound sizes). In short, if n large molecules are chosen as reactants from which m ($m \approx n$) random products of similar size are generated, then the odds of generating pre-existing compounds as products diminish rapidly as the size distribution of those compounds becomes top-heavy. Hence, most products are new, and REACT-COMP can only succeed a few times before $n_c = N_c$, after which it will predominately fail. With a bottom-heavy size distribution, REACT-COMP has a better chance of generating pre-existing compounds, thus prolonging the completion of the compound set and maintaining the flexibility to generate new legal reactions.

2.1 Reaction Energies

One critical aspect of MD-CHEM is the association of energy production and consumption with reactions in a manner that supports the basic biological fact that biomass construction demands energy, while biomass breakdown generally releases energy. At the reaction level, this translates into a simple qualitative MD-

CHEM energy principle: building larger compounds from smaller ones (i.e. anabolism) requires energy input, while the reverse process (i.e., catabolism) releases energy.

Statistical entropy is the basis of size comparisons between the reactant and product sets of a reaction. Entropy essentially measures the evenness and granularity of the size distribution of molecules: many small molecules having higher entropy than a few large molecules; and m molecules of similar size have higher entropy than m molecules of diverse sizes. The fractional sizes of each compound, f_i, relative to the total number of atoms in the reaction, determine the entropy according to:

$$\sum_{i=1}^{k} -f_i \log f_i \tag{2}$$

Thus, in reaction 1, which involves a total of 9 atoms, the fractional sizes are {1/3, 1/3, 1/3} for the reactants, and {1/3, 2/9, 4/9} for the products, yielding a reactant entropy of 1.585 and a product entropy of 1.530. Since lower entropy reflects higher order, the reaction exhibits a small degree of structure formation and is considered endothermic (i.e., energy consuming). MD-CHEM maps entropy differences directly into reaction energies, so this reaction requires 1.585 - 1.530 = 0.055 energy units in order to run. The reverse reaction is modelled as exothermic, yielding 0.055 energy units.

When deployed in an artificial-life simulator, MD-CHEM uses the law of mass action [14] to compute the rates of reactions. To wit, the product of the concentrations of the reactants along with a basal reaction constant determine the rate at which reactants are converted into products and energy is produced or consumed.

2.2 Catalysts

Catalytic relationships between compounds and reactions arise randomly during the reaction-generation process. Any compound that appears as both a reactant and product in the same reaction is considered a catalyst (i.e., enzyme). When running a reaction, MD-CHEM assumes that unless the catalyst is completely absent (concentration of 0.0), the reaction is substrate-limited and can proceed as if each catalyst molecule could instantaneously derive product from reactant molecules. Hence, the law of mass action is only applied to the concentrations of the non-enzymatic compounds in deriving the reaction rate. Of course, if the enzyme is completely absent, then the reaction cannot occur.

To model reaction catalysis, MD-CHEM uses the enzyme's molecular weight (i.e., number of atoms) as a rough estimate of catalytic enhancement, e_c. For exothermic reactions, e_c appears as an extra product in the mass-action derivation of the reaction rate, while in endothermic reactions, the energy consumption of the uphill reaction is reduced by a factor of $1/e_c$.

$$a_2b + ab_2c + c_3 \Longleftrightarrow a_2b + ab_2c_4 \tag{3}$$

For example, if $e_c(c_i) = 1 + length(c_i)$ for any catalytic compound c_i, then the catalytic enhancement of reaction 3 is $1 + length(a_2b) = 4$. Since the left-to-right reaction is uphill/endothermic, with an entropy change from 1.571 to 0.881, or 0.69, the catalytic effect essentially reduces the energy requirement of the uphill reaction to 0.69/4, or 0.1725. Similarly, the rate of the right-to-left downhill reaction will be enhanced by a factor of 4 in any context in which the reaction runs.

3 Deployment in Artificial Life Simulators

A central motivation behind MD-CHEM is an investigation into the abstract relationships between chemistries and the metabolisms that they support, and hence the biochemical guilds that evolve. A typical scenario is to generate hundreds or thousands of different chemistries and test each one in an artificial life simulation of metabolizing organisms. Those chemistries on which microorganism communities can survive are separated from the less supportive variants to get a general understanding of the life-sustaining aspects of a chemical system. Furthermore, a comparison of the pairings of life-giving chemistries and their induced microcosms could yield interesting insights into the chemical foundations of different forms of biochemical guilds.

Consequently, MD-CHEM has been integrated into our METAMIC (Metabolizing Abstract Microorganism) system, where it generates a multitude of chemistries, all of which are tested in simulated populations of metabolizing microscopic agents. This section provides a brief introduction to METAMIC and illustrates its usage with MD-CHEM chemistries.

3.1 METAMIC

Each organism in METAMIC is modelled as a cell with a genetically-determined metabolism, a local chemical

buffer, and a semi-permeable membrane that separates the cell from a global chemical environment. On each timestep, the agent a) performs its metabolic activities, with all chemical reactions driven by intracellular chemical concentrations, and b) exchanges chemicals via diffusion with the global compartment.

METAMIC employs MD-CHEM chemistries as bases for all intra- and extra-cellular chemical activity. The chemical basis for a METAMIC run is defined by the pair (C,R), where C denotes the set of legal compounds, and R the legal reactions.

Each organism's genetic-algorithm (GA) genome encodes r_T, a subset of R, plus base reaction constants (within a user-specified range) for each reaction. The exothermic, r_x, and endothermic, r_n, reactions are separated, where $r_T = r_x \cup r_n$; together, they compose the organism's metabolism.

Those compounds that constitute biomass can vary between organisms, with key restrictions. Any compound declared as biomass cannot diffuse into or out of that organism's cell: all biomass molecules, aside from those present at birth, result from internal production. Two user-defined parameters, N_{bio} and N^*_{bio}, specify the maximum number of compounds that an organism can treat as biomass, and the number of large compounds in C that are legal biomass constituents. To choose the biomass compounds, c_{bio}, for an agent, METAMIC gathers the N_{bio} largest compounds that the organism is a net producer of in r_T. It then intersects that set with the N^*_{bio} largest compounds in C to form c_{bio}. An organism's biomass is then the total internal mass of all c_{bio} compounds.

In METAMIC's GA, fitness is implicit: if an organism doubles its birth biomass, then reproduction occurs by asexual splitting, with possible mutation of both child genomes. Organisms also undergo a form of double bacterial conjugation by occasionally swapping GA chromosome segments with one another. This is essentially standard GA crossover, with each individual continuing its life, but with a new genome and metabolism. The same mortality rate pertains to all living organisms, regardless of age or biomass.

The abstract metabolic process consists of two phases: catabolism and anabolism. On each timestep, organisms receive an energy request and begin catabolism, wherein the exothermic reactions, r_x, run for the maximum of two durations: a) the actual timestep b) the estimated time needed to generate the required energy (based on previous energy-production rates of the r_x). If the former exceeds the latter, then an energy surplus results, thus triggering the anabolic processes (i.e., the endothermic reactions, r_n), which run long enough to consume the available energy and build structure by reducing internal entropy.

The current version of METAMIC employs a simple box model, with all agents residing within a single global environment, E. For each compound in C, the user specifies separate inflow and outflow rates for E. All reactions in R are assumed to occur in E, and since no energy is demanded of E, the energetic fruits of r_x go directly into r_n. The user can also specify a constant energy input to the system, which is distributed among all agents.

3.2 Sample METAMIC Runs

To illustrate the coupling between MD-CHEM and METAMIC, we ask the former to generate a chemistry of 15 compounds and 30 reactions from four atoms: C, H, O, and N. Also, $N_{ic} = 5$, $R_{cs} = [2, 12]$, $R_{rs} = [1, 3]$, $B_{rs} = \{.33, .33, .33\}$, and $B_{ics} = \{.33, .33, .33, 0, \ldots, 0\}$ (biased toward the 3 smallest sizes). We also specify that for each new reaction, its inverse is automatically included. Figure 2 displays the resulting chemistry, CHEM*.

CHEM* is then used as the chemical foundation for a METAMIC run with a start population of 50 agents, a maximum population size of 1000, a chromosome length of 10 reactions (duplicates are ignored), a mutation rate of 0.1 (per gene), and a conjugation fraction of 0.3, where conjugation occurs every 20 timesteps and each such episode denotes a generation, and a mortality rate of .005 (per timestep). For biomass calculations, $N^*_{bio} = 4$, and $N_{bio} = 2$.

Initial concentrations of all compounds in both E and the agent cells are 0.01, and the inflows to E are 1 mass unit per timestep of the 4 smallest compounds (CHO, CNO, N_2O, C_4) and 0 units for all others. However, no external energy inputs occur, nor is energy demanded of the agents for activities other than the running of their endothermic reactions. E has a volume of 100,000 cubic units, while each cell has a volume of 10 cubic units.

As Figure 3 illustrates, the population rapidly grows to the upper bound of 1000, and metabolic diversity increases through all 100 generations. The diversity measures are based on the statistical entropy across the population of 5 different collections: a) biomass compounds, b) compounds that the cell produces (net), c) compounds that the cell consumes, d) exothermic reactions, and e) endothermic reactions.

Detailed analysis of the 1000 metabolisms at generation 100 reveals the four most common exother-

Compounds:
CHO, CNO, N_2O, C_4, CH_2N, HN_2O, HO_3, CN_3O_2, $CH_2N_2O_2$, HN_2O_4, C_5H_2N, $CH_3N_2O_2$, H_4NO_4, $C_5N_3O_2$, HN_4O_5

Reactions:	dG	#
0: CNO + $N_2O \Rightarrow CN_3O_2$	1.00	123
1: $CN_3O_2 \Rightarrow$ CNO + N_2O	-1.00	565
2: $N_2O + HO_3 \Rightarrow HN_2O_4$	0.99	330
3: $HN_2O_4 \Rightarrow N_2O + HO_3$	-.99	140
4: $C_4 + CN_3O_2 \Rightarrow C_5N_3O_2$	0.97	955
5: $C_5N_3O_2 \Rightarrow C_4 + CN_3O_2$	-.97	40
6: CNO + $CN_3O_2 + HN_2O_4 \Rightarrow$ CNO + CNO + HN_4O_5	0.18	174
7: CNO + CNO + $HN_4O_5 \Rightarrow$ CNO + $CN_3O_2 + HN_2O_4$	-.18	50
8: $CH_2N + CH_2N + HN_2O_4 \Rightarrow CH_2N_2O_2 + CH_3N_2O_2$	0.53	126
9: $CH_2N_2O_2 + CH_3N_2O_2 \Rightarrow CH_2N + CH_2N + HN_2O_4$	-.53	291
10: $HO_3 + CH_3N_2O_2 \Rightarrow$ CNO + H_4NO_4	0.11	674
11: CNO + $H_4NO_4 \Rightarrow HO_3 + CH_3N_2O_2$	-.11	68
12: $C_4 + CH_2N \Rightarrow C_5H_2N$	1.00	48
13: $C_5H_2N \Rightarrow C_4 + CH_2N$	-1.00	749
14: $CH_2N_2O_2 \Rightarrow$ CHO + HN_2O	-.99	247
15: CHO + $HN_2O \Rightarrow CH_2N_2O_2$	0.99	45
16: CHO + $N_2O + HO_3 \Rightarrow$ CHO + HN_2O_4	0.69	82
17: CHO + $HN_2O_4 \Rightarrow$ CHO + $N_2O + HO_3$	-.69	116
18: $HO_3 + CN_3O_2 \Rightarrow$ CNO + HN_2O_4	0.09	135
19: CNO + $HN_2O_4 \Rightarrow HO_3 + CN_3O_2$	-.09	152
20: CHO + $CH_2N_2O_2 \Rightarrow$ CHO + CHO + HN_2O	-.69	96
21: CHO + CHO + $HN_2O \Rightarrow$ CHO + $CH_2N_2O_2$	0.69	592
22: $N_2O + CH_3N_2O_2 \Rightarrow$ CHO + $HN_2O + HN_2O$	-.73	201
23: CHO + $HN_2O + HN_2O \Rightarrow N_2O + CH_3N_2O_2$	0.73	364
24: CNO + $CN_3O_2 \Rightarrow$ CNO + CNO + N_2O	-.67	169
25: CNO + CNO + $N_2O \Rightarrow$ CNO + CN_3O_2	0.67	833
26: $CH_2N_2O_2 + HN_2O_4 \Rightarrow$ CHO + $HN_2O + HN_2O_4$	-.49	209
27: CHO $HN_2O\ HN_2O_4 \Rightarrow CH_2N_2O_2 + HN_2O_4$	0.49	85
28: $N_2O + HN_2O_4 + HN_2O_4 \Rightarrow HN_2O_4 + HN_4O_5$	0.52	240
29: $HN_2O_4 + HN_4O_5 \Rightarrow N_2O + HN_2O_4 + HN_2O_4$	-.52	587

Figure 2: An MD-CHEM random chemistry, CHEM*, of 15 compounds and 30 reactions, including inverses. Energy (dG) values are positive (negative) for endothermic (exothermic) reactions. Frequencies (#) denote the number of organism' metabolisms in which the reaction occurs during the 100th (final) generation of a METAMIC run, where the final population size is 1000.

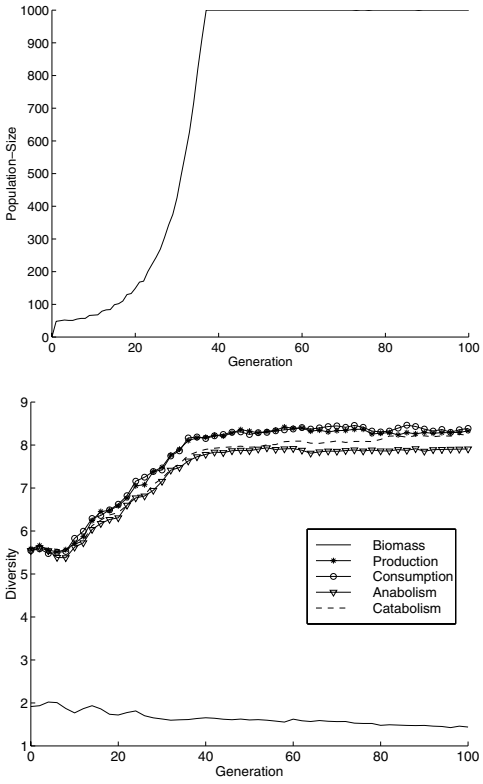


Figure 3: Evolutionary progression of population size and metabolic diversity for a METAMIC population using CHEM* as their underlying chemistry.

mic(catabolic) gene sequences, listed below with reaction indices in parentheses and frequencies in brackets:

1: {1, 13, 29 }[46]
2: {1, 9, 13, 29 }[23]
3: {1, 13 }[20]
4: {13, 29 }[17]

The four most common endothermic (anabolic) gene sequences are:

1: {4, 10, 21, 25 }[52]
2: {2, 4, 10, 21, 25 }[38]
3: {4, 10, 21, 23, 25 }[33]
4: {4, 10, 25 }[21]

Two of the six possible choices of biomass pairs dominate the population:

1: {$C_5N_3O_2$, H_4NO_4 } [494]
2: {$C_5N_3O_2$, HN_4O_5 } [390]

All 5 exothermic groups share reaction 13, R_{13}, which breaks down C_5H_2N, the largest non-biomass compound. As an interesting asymmetry, note that R_{13} has a very high frequency(749), whereas its inverse, R_{12} has a low count(48). Since no other reactions involve C_5H_2N, it clearly functions as a primary energy source with approximately 95% of the population consuming it and only 4% producing it.

Similarly, R_4, R_{10}, and R_{25} appear in all 5 popular anabolic sets, and each forms a very asymmetric pair with its inverse reaction. Not surprisingly, R_4 and R_{10} produce the two most popular biomass compounds, $C_5N_3O_2$ and H_4NO_4, respectively, while R_{25} produces CN_3O_2, which is a key precursor to $C_5N_3O_2$ via R_4, and to HN_4O_5 (the third most common biomass compound) via R_6. The other key reactant in R_6 is HN_2O_4, which is autocatalytic via R_{29}, a very high frequency (587) reaction.

So clearly, the evolved choices of reactions and reaction groups tap the energy- and biomass-producing potential in CHEM* and the environment, E (where E contributes with a large but finite supply of C_5H_2N and

a replenished supply of the four smallest compounds). In this case, the result is not a single dominant phenotype, but a diverse collection of metabolisms that may share a few core reactions, but which also perform complementary biochemical activities, such as producing HN_2O_4 from HN_4O_5 via R_{29}, which enables other cells to produce HN_4O_5 from HN_2O_4 via R_{28} or R_6.

3.2.1 50 Chemistries

The CHEM* example shows how a population evolves to exploit the underlying chemistry, or, conversely how chemistry constraints biotic emergence. To further illustrate this connection, METAMIC was run using 50 different randomly-generated MD-CHEM chemistries, all composed of 15 compounds and 30 reactions. Each METAMIC run for a given chemistry was repeated 10 times, using different random seeds, with final population sizes and metabolic diversities averaged and plotted. METAMIC used the same parameter settings as before, except that the number of generations was reduced to 40 and the maximum population size to 500.

The upper graph in Figure 4 shows the 50 chemistries on the x-axis, sorted by the average population sizes that they induced (y-axis). About 90% of the chemistries supported growth, while about 10% consistently yielded population explosions to 500.

In the lower graph of Figure 4, diversity roughly tracks population size, but note the wide diversity differences between chemistries that induce similar population sizes. Clearly, the distributions of metabolic guilds can vary considerably among chemistries that support the same basic amount of life. So chemistry is a key determinant of population size and diversity, but the causal relationships are non-trivial.

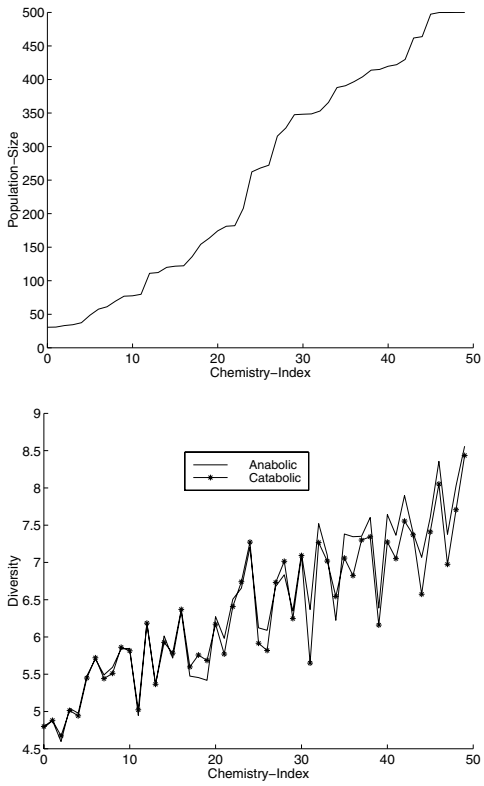


Figure 4: Average results of running 50 different MD-CHEM chemistries 10 times each in METAMIC. Chemistries are sorted and indexed based on the average population size that METAMIC achieved in the 10 runs.

4 Discussion

This research is an initial step in using artificial life techniques to better understand the connections between chemistries and the biological and ecological structures that they engender. By remaining at the level of abstract chemistries, thousands of which can be generated, we avoid heavy biases toward Earth's biochemistry and can focus on general principles such as the relationships between energy and structural entropy. The results above are merely illustrations of how a chemistry generator and an alife simulator can interact in pursuit of this goal.

Simulations of this sort could enable investigations into the "inevitability" of certain biological phenomena across a wide spectra of base chemistries. We have delved deeply into Gaia Theory [12], the premise that the biota can exhibit control over large-scale geochemical processes, in trying to understand how these homeostatic mechanisms might emerge from a population of evolving organisms. Since contemporary Gaia models [17, 6] are extremely abstract, lacking good chemical and/or genetic modules, the conclusions drawn from them are highly questionable.

These shortcomings motivated the design MD-CHEM and METAMIC and their use in Gaian simulations, wherein chemical compounds can affect physical variables (such as temperature), which then affect metabolic rates. MD-CHEM generated 100 life-supporting chemistries, and 21 of these permitted emergent environmental control, a key Gaian fingerprint. Many of these 21 chemistries provide alternate biomass-producing pathways with different energy demands such that organisms can trade off metabolic efficiency for environmental control by evolving metabolisms that yield less biomass per unit energy but indirectly regulate ambient physical fac-

tors.

Three potential MD-CHEM improvements are apparent. First, a model of atomic bonding would provide a principled basis for reaction generation. Second, as in most artificial chemistries, catalyzation in MD-CHEM needs a firmer mechanistic grounding.

Third, a key mechanism in all biochemical systems is inhibition, the basis for regulation of enzymatic activity. It may emerge in abstract chemistries via competition for binding sites, but it rarely does. The simple mass-action dynamics of MD-CHEM only vaguely support competition in the sense that if compound A is a key reactant in reactions R_1 and R_2, where it binds to X and Y, respectively, then an increase in X would accelerate R_1, thus draining more A and slowing R_2. But MD-CHEM needs stronger constraints, for example, that X has a stronger affinity for A than does Y. Then, important physiological mechanisms, such as O_2 poisoning of anaerobes (due to oxygen's electron-grabbing tenacity), may arise naturally.

However, all three improvements entail greater chemical detail, for better or worse. Managing these tradeoffs between abstraction and reduction, while capturing the vital essence of biochemistry, is the primary challenge to the continued improvement of abstract chemistries for artificial life.

References

[1] Richard J. Bagley, J. Doyne Farmer, and Walter Fontana. Evolution of a metabolism. In Christopher G. Langton, Charles Taylor, J. Doyne Farmer, and Steen Rasmussen, editors, *Artificial Life II*, pages 141–158. Addison-Wesley, Redwood City, CA, 1992.

[2] Pietro Speroni di Fenizio. A less abstract artificial chemistry. In M. Bedau, J. McCaskill, N. Packard, and S. Rasmussen, editors, *Artificial Life VII*, pages 49–53, Cambridge, MA, 2000. MIT Press.

[3] Peter Dittrich. Artificial chemistries. Tutorial held at ECAL'99 (European Conference on Artificial Life), 1999.

[4] Peter Dittrich and Wolfgang Banzhaf. Self-evolution in a constructive binary string system. *Artificial Life*, 4(2):203–220, 1998.

[5] Keith Downing. Euzone: Simulating the emergence of aquatic ecosystems. *Artificial Life*, 3(4):307–333, 1998.

[6] Keith Downing and Peter Zvirinsky. The simulated evolution of biochemical guilds: Reconciling gaia theory and natural selection. *Artificial Life*, 5(4):291–318, 1999.

[7] Walter Fontana. Algorithmic chemistry. In Christopher G. Langton, Charles Taylor, J. Doyne Farmer, and Steen Rasmussen, editors, *Artificial Life II*, pages 159–209. Addison-Wesley, Redwood City, CA, 1992.

[8] Chikara Furusawa and Kunihiko Kaneko. Emergence of multicellular organisms with dynamic differentiation and spatial pattern. *Artificial Life*, 4(1):79–93, 1998.

[9] John H. Holland. *Hidden Order: How Adaptation Builds Complexity*. Addison-Wesley, Reading, Massachusetts, 1 edition, 1995.

[10] Harald Huning. A search for multiple autocatalytic sets in artificial chemistries based on boolean networks. In M. Bedau, J. McCaskill, N. Packard, and S. Rasmussen, editors, *Artificial Life VII*, pages 64–72, Cambridge, MA, 2000. MIT Press.

[11] Stuart A. Kauffman. *The Origins of Order*. Oxford University Press, Oxford, 1993.

[12] James Lovelock. *The Ages of Gaia: A Biography of Our Living Earth*. Oxford University Press, Oxford, England, 1995.

[13] Bernd Mayer and Steen Rasmussen. Self-reproduction of dynamical hierarchies in chemical systems. In C. Adami, R. Belew, H. Kitano, and C. Taylor, editors, *Artificial Life VI*, pages 123–129, Cambridge, MA, 1998. MIT Press.

[14] Douglas S. Riggs. *The Mathematical Approach to Physiological Problems*. MIT Press, Cambridge, MA, 1963.

[15] Karl Sims. Evolving 3D morphology and behavior by competition. In R. Brooks and P. Maes, editors, *Artificial Life IV*, pages 28–39, Cambridge, MA, 1994. MIT Press.

[16] Yasuhiro Suzuki and Hiroshi Tanaka. Order parameter for a symbolic chemical system. In C. Adami, R. Belew, H. Kitano, and C. Taylor, editors, *Artificial Life VI*, pages 130–139, Cambridge, MA, 1998. MIT Press.

[17] Andrew Watson and James Lovelock. Biological homeostasis of the global environment: The parable of daisyworld. *Tellus*, 35B:284–289, 1983.

Evolution of strategies in repeated stochastic games with full information of the payoff matrix

Anders Eriksson and Kristian Lindgren
Department of Physical Resource Theory, Complex Systems Group
Chalmers University of Technology and Göteborg University
SE-41296 Göteborg, Sweden
Corresponding author: K. Lindgren (frtkl@fy.chalmers.se, phone: +46(0)31 - 772 3131)

Abstract

A framework for studying the evolution of cooperative behaviour, using evolution of finite state strategies, is presented. The interaction between agents is modelled by a repeated game with random observable payoffs. The agents are thus faced with a more complex (and general) situation, compared to the Prisoner's Dilemma that has been widely used for investigating the conditions for cooperation in evolving populations. Still, there is a robust cooperating strategy that usually evolves in a population of agents. In the cooperative mode, this strategy selects an action that allows for maximizing the payoff sum of both players in each round, regardless of the own payoff. Two such strategies maximize the expected total payoff. If the opponent deviates from this scheme, the strategy invokes a punishment action, which for example could be to aim for the single round Nash equilibrium for the rest of the (possibly infinitely) repeated game.

1 INTRODUCTION

The conditions for cooperative behaviour to evolve have been the focus in a large number of studies using genetic programming, evolutionary algorithms, and evolutionary modelling (Matsuo 1985, Axelrod 1987, Miller 1989, Lindgren 1992, 1997, Nowak and May 1993, Stanley *et al* 1993, Lindgren and Nordahl 1994, Ikegami 1994, Nowak *et al* 1995, Wu and Axelrod 1995). Most of these studies have used the iterated Prisoner's Dilemma (PD) game as a model for the interactions between pairs of individuals (Rapoport and Chammah 1965, Axelrod 1984, Sugden 1986), and in the PD game cooperation is the action that lets two players share the highest total payoff. The problem in the single round PD is that one is tempted to defect to get a higher score, but when the game is repeated one usually finds that various types of cooperative behaviour evolves, for example in the form of the Tit-for-tat strategy (Axelrod 1984).

The basis for cooperation to emerge in these models is the repeated encounters between players. In "real" situations one usually meets the other player again, but seldom in an identical situation. Therefore, we suggest and analyse a more open game-theoretic problem as a basis for the interactions between players. In each round a completely random payoff matrix is generated, and the players observe the payoff matrix before choosing their actions. In order to compare this model with the iterated PD, we introduce a parameter that can continuously change the game from the iterated PD to the repeated random payoff game.

In this model cooperation is not associated with a certain action, but rather with how one chooses to act depending on the structure of the present payoff matrix in combination with how the opponent has acted in previous rounds. There is a very strong cooperative strategy in this type of game, namely the strategy that always plays the action that aims for the highest payoff *sum* for the two players, regardless of the own score. If two such players meet they will, in the long run, get equal scores at the maximum possible level. The circumstances for such cooperative strategies to evolve may be critical, though, and this paper is a starting point for investigating what strategy types evolve in this type of open game.

The model presented here is a population dynamics model with strategies interacting according to their abundances in the population. The equations of motion for the different variables (the number of individuals) are thus coupled, resulting in a *coevolutionary* system. The success or failure of a certain type of in-

dividual (species) depends on which other individuals are present, and there is not a fixed fitness landscape that is explored by the evolutionary dynamics.

The paper is organised as follows. First we give a brief presentation of the repeated stochastic game and some of its characteristics. Then we describe the agents including the strategy representation, and how the population dynamics with mutations lead to a coevolutionary model. In the final sections we briefly discuss some preliminary results and directions for future research within this model framework.

2 RANDOM PAYOFF GAME

The repeated random payoff game used in this study is a two-person game in which each round is characterized by two possible actions per player and a randomly generated but observable payoff matrix. The payoff matrix elements are random, independent, and uniformly distributed real numbers in the range $[0, 1]$. New payoffs are drawn every round of the game, as in Table 1.

The agents playing the game have complete knowledge of the payoffs $u^i_{jk}(t)$ for both players $i \in \{A, B\}$ and possible actions j and k for the present round t. Here all $u^i_{jk}(t)$ are independent random variables with uniform distribution.

The single round game can be characterised by its Nash equilibria (NE), i.e., a pair of actions such that if only one of the players switches action that player will reduce her payoff.[1] If

$$(u^A_{00} - u^A_{10})(u^A_{01} - u^A_{11}) > 0$$
$$\text{or}$$
$$(u^B_{00} - u^B_{01})(u^B_{10} - u^B_{11}) > 0, \quad (1)$$

there is exactly one (pure strategy) Nash equilibrium in the single round game. This occurs in 3/4 of the rounds. If this does not hold, and if

$$(u^A_{00} - u^A_{10})(u^B_{00} - u^B_{01}) > 0, \quad (2)$$

there are two (pure strategy) NEs, which occurs in 1/8 of the rounds. Otherwise, there are no (pure strategy) NEs.

There are a number of simple single round (elementary) strategies that are of interest in characterising the game. Assume first that there is exactly one NE, and that rational (single round) players play the corresponding actions. The payoff in this case is $\max(x, h)$, where x and h are independent uniformly distributed stochastic variables in $[0, 1]$, and this results in an expectation value of $2/3 \approx 0.667$.

Let us define a strategy "NashSeek" as follows. If there is only one NE in the current payoff matrix, one chooses the corresponding action. If there are two NE, one aim for the one that has the highest sum of the two players payoffs, while if there is no NE, one optimistically chooses the action that could possibly lead to the highest own payoff.

A second strategy, "MaxCoop", denoted by C, aims for the highest sum of both players' payoffs. If two such players meet, they score $\max(x_1 + h_1, x_2 + h_2, x_3 + h_3, x_4 + h_4)$ together, where x_i and h_i are independent uniformly distributed stochastic variables in $[0, 1]$, and this results in an expectation value of $s_C = 3589/5040 \approx 0.712$.

If a player using MaxCoop wants to avoid exploitation, some punishment mechanism must be added. A new elementary strategy could be invoked for this, so that if the opponent deviates from MaxCoop, then the defector will at the most get the minmax score for the rest of the game. We call such an elementary strategy "Punish", denoted by P, and the strategy being punished will at the most get $\min(\max(x_1, x_2), \max(x_3, x_4))$, with an expectation value of $s_P = 8/15 \approx 0.533$. Other forms of punishment, that are less costly for both players, are also possible. In Table 2 we show the expected payoffs between the simple strategies in this paper. All expected payoffs for the simple strategies were computed analytically.

This illustrates that both players choosing MaxCoop with punishment correspond to a Nash equilibrium in the infinitely (undiscounted) iterated game. In a discounted iterated game, with a probability δ for the game to end in each round, the condition for avoiding exploitation is $1 + (1 - \delta)s_P/\delta < s_C/\delta$. This means that if $\delta < (s_C - s_P)/(1 - s_P) = 901/2352 \approx 0.383$, then both choosing MaxCoop with punishment is a NE.

In order to be able to illustrate how this model is an extension of the iterated Prisoner's Dilemma game, we introduce a stochasticity parameter r $(0 = r = 1)$ that changes the payoff characteristics from the simple PD game to the fully stochastic payoff. The r-dependent payoff matrix is shown in Table 1.

3 EVOLUTIONARY MODEL

We consider a population of N agents, competing for the same resources. The population is at the limit of

[1] We assume that no payoff values are identical, so that Nash equilibria are strict. Rounds for which this does not hold are of measure zero.

Table 1: Payoff matrix for the stochastic game depending on the parameter r that brings the game from an ordinary PD to the fully stochastic game. The symbols η_i denote independent uniformly distributed random variables in the interval $[0, 1]$, and note that for $r = 1$ these form the payoffs of the game.

	Action 0	Action 1
Action 0	$(1-r) + r\eta_1$ $(1-r) + r\eta_5$	$r\eta_2$ $\frac{5}{3}(1-r) + r\eta_6$
Action 1	$\frac{5}{3}(1-r) + r\eta_3$ $r\eta_7$	$\frac{1}{3}(1-r) + r\eta_4$ $\frac{1}{3}(1-r) + r\eta_8$

Table 2: Expected payoffs for the simple strategies, for the row player. The strategies are defined in section 3.1.

	MaxMax	Punish	MaxCoop	NashSeek
MaxMax	0.600	0.400	0.710	0.619
Punish	0.500	0.400	0.549	0.519
MaxCoop	0.592	0.431	0.712	0.610
NashSeek	0.643	0.443	0.750	0.662

the carrying capacity level of the environment, so the number of agents N is fixed.

In each generation, the agents play the repeated stochastic payoff game with the other agents, and reproduce to form the next generation. The score s_i for strategy i is compared to the average score of the population, and those above average will get more offspring, and thus a larger share in the next generation. Let g_{ij} be the score of strategy i against strategy j, and let x_i be the fraction of the population occupied by strategy i. The score s_i is then

$$s_i = \sum_j g_{ij} x_j, \tag{3}$$

and the average score s is

$$s = \sum_i s_i x_i. \tag{4}$$

We define the fitness w_i of an individual as the difference between the its own score and the average score:

$$w_i = s_i - s \tag{5}$$

Note that this ensures that the average fitness is zero, and thus the number of agents is conserved. The fraction x_i of the population of the strategy i changes as

$$x_i(t+1) = x_i(t) + d\, w_i\, x_i(t) = x_i(t) + d\,(s_i - s)\, x_i(t), \tag{6}$$

where d is a growth parameter, and all terms to the right of the first equality sign are evaluated at time t.

When an individual reproduces, a mutation may occur with low probability. The mutations are described in detail in section 3.2. In this way, diversity is introduced in the population. In the population dynamics, this corresponds to a number of mutations per strategy proportional to the strategies share of the population. The share of the mutant strategy is increased by $1/N$ (corresponding to one individual), and the mutated strategy loses the same share.

3.1 PLAYERS AND STRATEGIES

In many approaches to evolutionary game-theoretic models, strategies are represented as deterministic finite state automata (FSA) (Miller 1988, Nowak *et al* 1995, Lindgren 1997), where each transition conditions on the action of the opponent. Each state is associated with an action, and a certain state is the starting state. In order to calculate the payoffs for a pair of players, an FSA describing the joint states of the players is constructed. The expected undiscounted payoffs are then computed from this FSA.

However, since our games are stochastic, the result of each action varies between rounds. In this case, a natural extension is to act according to a simple behaviour rather than a pure action. In this article, we choose the behaviours to be deterministic maps from a payoff matrix to an action, and we call these elementary strategies. In our first experiments, the behaviours available to the agents are MaxMax (M), Punish (P), MaxCoop (C), and NashSeek (N), described as follows:

- MaxMax: Select the action so that it is possible to get the maximum payoff for the agent in the payoff matrix. Greedy and optimistic.
- Punish: Select the action that minimizes the opponent's maximum payoff. Note that this may be even more costly for the punishing strategy.
- MaxCoop: Assume that the other player also plays MaxCoop, and select the action that maximizes the sum of the players' payoffs.
- NashSeek: If there is only one Nash equilibrium, select the corresponding action. If there are two Nash equilibria, select the one that gives the highest sum of the players' payoffs. If there are no equilibria, play according to MaxMax (see above).

Agents thus have four possible elementary strategies, each determining what specific action should be chosen when a certain payoff matrix is observed. We choose to have only four simple strategies, since we want to keep our system as simple as possible. The MaxCoop is a natural member, since it is the optimal behaviour at self-play. In our choices of the other behaviours, we have tried to find the most interesting aspects of the game: Nash equilibria, punishment against defection, and greediness.

Since the agents cannot access the internal state of the opponent, but only observes the pure actions, we build an action profile by comparing the opponent's action to the actions that the opponent would have taken, given that it followed each of the elementary strategies. In this way, a player may determine whether the opponent's action is consistent with a certain elementary strategy.

We now construct an FSA strategy representation that allows for more complex (composite) strategies by associating different internal states with (possibly) different elementary strategies. In each state, the agent has a list of transitions to states (including the current state). Each transition is characterized by a condition on the opponent's action profile. For each elementary strategy, the agent may require a match or a non-match, or it may be indifferent to that behaviour. The transitions are matched in the order they occur in the list, and the first matching transition is followed. If no transition matches, the state is retained to the next step.

When solving for the players' payoffs, we construct the joint state FSA of the whole iterated game.[2] The expected payoffs from all pairs of behaviours, and the probability distributions over the set of action profiles are precomputed. Since the payoff matrices are independent stochastic variables, we can compute the probability distributions of the transitions from each node in the game FSA by summing over the action profile distribution. It is then straightforward to compute the resulting stationary distribution, and the corresponding payoffs.

3.2 MUTATIONS

There are several different ways to change a strategy by mutations. Since we represent our strategies by graphs, there are some basic graph changes that are suitable as mutations.

A graph may be mutated by changing the number of nodes, changing the connectivity of the graph, or by changing the labels of the nodes. See Fig. 2 for an example mutation. Removing a node constitutes deleting the node, and removing all edges pointing to that node. Adding a node is made in such a way that the strategy is unchanged, by splitting an existing node as follows. A new node is created, and all the edges leaving the original node are copied to this node. All self-referential edges of the original node are redirected to the new node. Finally, a match-all edge to the new node is added last in the rule list of the original node.

We think that it is important that a strategy may change in size without changing the behaviour. If a strategy is at a fitness peak, many mutations that change the fitness may be needed to reach another peak, using only the mutations that change the strategy. But if a strategy may grow more complex without endanger the good fitness, these mutation might bring it close to other peaks (in terms of using the other mutations). That genetic diversification among agents with identical fitness is important has been observed for example in hill climbing algorithms for satisfiability (SAT) problems (Selman *et al* 1992).

Since large random changes in a strategy often are deleterious, this allows the strategy to grow more complex in a more controlled way. For example, it might lead to a specialization of the old strategy.

The connectivity of the graph is mutated by adding a

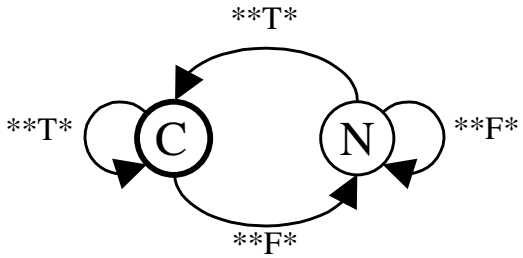


Figure 1: Example of a graph playing strategy. The letters in the nodes stands for the elementary strategies MaxCoop (C) and NashSeek (N), and the black ring indicates the initial state. The transition denoted **T* applies only when the opponent's action is consistent with MaxCoop (the third elementary strategy), but does not care about the other elementary strategies. The transition rule **F* is used when the opponent does not follow the MaxCoop strategy. This composed strategy is similar to the Tit-for-tat strategy in that it cooperates by playing MaxCoop if the opponent played according to MaxCoop in the last round, otherwise it defects by playing NashSeek.

[2] A more detailed description is in preparation (Eriksson & Lindgren 2001).

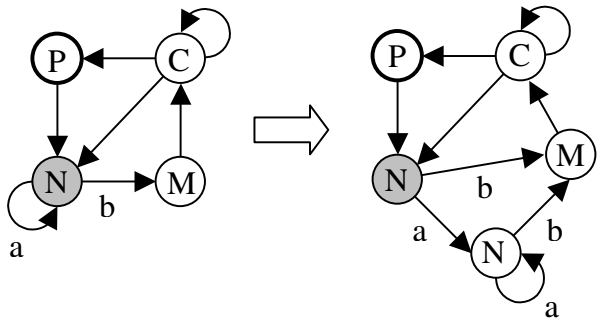


Figure 2: Example of a neutral mutation of the grey node, adding a node. Labels a and b are transition rules. The new node is created with the same links as the grey node, and all self-links of the grey node are directed to the new node.

random rule, removing a rule, or by mutating a rule. Rules are mutated by changing the matching criterion at one location, or by changing the rule order (since the rules are matched in a specific order, this changes the probability of a match by that rule). Finally, a node may be mutated by randomly selecting a different strategy as a label. After each mutation, we perform a graph simplification where we remove all nodes that are unreachable from the start node, in order to keep the representation as compact as possible.

In our experiments, the most common mutation is the node strategy mutation. We keep the node addition and removal equiprobable, so that there is no inherent preference for strategy growth in the simulation. The same holds for rule addition and removal. All growth of the strategies that is observed in the population is thus due to the competition between strategies, and the need for complexity that arises as a consequence of that competition.

4 RESULTS

The model described in the previous sections has been simulated and analysed in a number of ways. First, we have made a simple analysis of how the scores between pairs of elementary strategies depend on the degree of stochasticity in the payoff matrix, in the range from the pure PD game to the fully stochastic payoff game. In Table 2, the results for a number of strategies are shown. The table shows clearly that there is a possibility for a cooperative behaviour scoring higher than the short term Nash equilibrium level.

The strategy NashSeekOpt (play optimally against NashSeek) is identical to the basic Nash-seeking strategy (NashSeek) when there is only one Nash equilibrium in the single round game, but plays optimal against NashSeek in case of no or two Nash equilibria (and thus exploiting NashSeek).

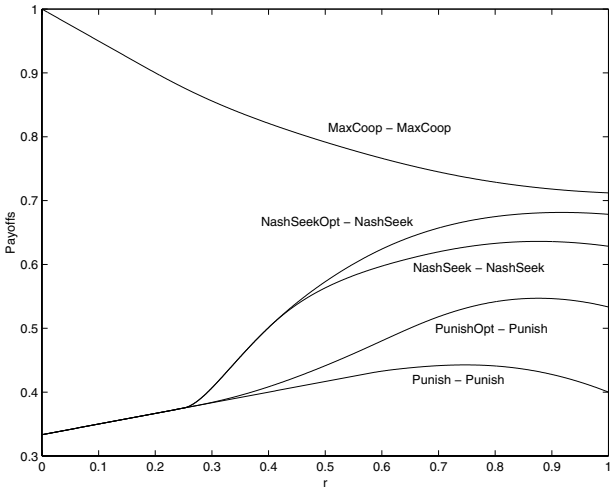


Figure 3: By the stochasticity parameter r, we can make a continuous transition from a pure Prisoner's Dilemma game to the fully stochastic payoff matrix. Here we show the first player's score for different pairs of strategies as functions of r. The strategy PunishOpt plays optimally against Punish.

A player that wishes punish an opponent, may try to minimize the score for the opponent by playing a Min-Max strategy, and the opponent's score in that case is the second lowest curve in Fig. 3. From the figure we see that when the game becomes more stochastic there may be several elementary strategies that can work as a punishment, and that this allows for punishments of different magnitudes. In the PD game, of course, the only way to punish the opponent (in a single round) is to defect (which is equivalent to NashSeek and Punish).

In Fig. 4, we show the evolution of the average score in the population in the case of a fully stochastic payoff matrix. The score quickly (in a few thousand generations) increases to the level of the single round Nash equilibrium, but with sharp drops when mutant strategies manages to take a more substantial part of the population. After about 20,000 generations, the population is close to establishing a MaxCoop behaviour, but fails, probably because of the lack of a strong enough punishment mechanism. A new attempt is made after 60,000 generations, and here it seems that a composite strategy using MaxCoop against other cooperative strategies, but punishing exploiting opponents, is established.

The noisy behaviour observed in Fig. 4 is an effect of the sub-population of mutants created by the rel-

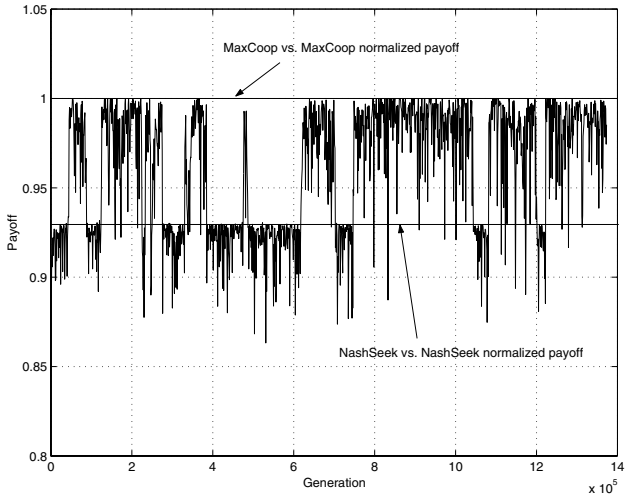


Figure 4: The average score in the population is shown for the first 1.4 million generations in the case of a fully stochastic payoff matrix. Population size is 100 strategies, the growth parameter d is 0.1, and the mutation rate m is 0.5. The score is normalized to the MaxCoop level. The thick lines indicate the normalized levels of MaxCoop vs. MaxCoop (upper) and NashSeek vs. NashSeek (lower). It is clear that the population finds a strategy with MaxCoop, that is resistant against greedy strategies for many generations.

atively high mutation rate (5%). There may be two mechanisms involved here. The immediate effect is that many mutants deviate from the MaxCoop strategy which induces all playing against them to switch to punishment, e.g., NashSeek. A secondary effect is that mutations that remove the punishment mechanism may survive for a while, growing in size by genetic drift, until a mutant that exploits its lack of punishment enters. Such an exploiting mutant may increase its fraction of the population at the cost of reducing both average population score and the size of the exploited strategy, eventually leading to the extinction of both of the mutants.

When the population reaches the score level of the MaxCoop strategy, there are periods when the score drops drastically down to the Nash level or even lower. A reasonable explanation is that the punishment mechanism that is necessary in order to establish the MaxCoop strategies when Nash-seeking strategies are present, is not needed when the Nash-seeking strategies have gone extinct. By genetic drift, MaxCoop strategies without a punishment mechanism may increase their fraction of the population, and then when a mutant NashSeek enters there is plenty of strategies for it to exploit, leading to a drop in average score. But then the MaxCoop with punishment can start to grow again, resulting in the noisy average score curve of Fig. 4. Occasionally, if the simulation is continued, there may be transitions back to the score level of the single round Nash equilibrium.

5 DISCUSSION

It is clear from the payoff matrix describing the elementary strategies, that the iterated game (with a sufficiently low discount rate) has Nash equilibria in the form of both players using a certain form of MaxCoop-punishment, the most common one being MaxCoop with NashSeek as punishment.

But it is also clear that there are several strategies that are playing on equal terms with such a MaxCoop-punishment strategy. For example, strategies that by mutation loose their punishment mechanism may enter and increase their fraction of the population by genetic drift. This in turn leads to a population that is vulnerable to mutants exploiting the cooperative character of the MaxCoop behaviour, for example by the NashSeek behaviour. Thus, the Nash equilibrium that characterises the population dominated by MaxCoop-punishment is not an evolutionarily stable one, as can be seen in the simulations. The effect is the same as the one in the iterated Prisoner's Dilemma that makes Tit-for-tat an evolutionarily non-stable strategy (cf. IPD; see, for example, (Boyd & Lorberbaum 1987, Boyd 1989)).

The evolution of a population dominated by MaxCoop (with some punishment mechanism) is not unexpected. A population of players (using identical strategies) can always enforce a certain score s^*, if this score is larger than (or equal to) the smallest punishment score s_P and smaller than (or equal to) the maximum cooperation score s_C, $s_P \leq s^* \leq s_C$. This means that the population can punish a new strategy (mutant) that enter, unless it adopts the same strategy and plays so that the score s^* is reached in the long run. This argument follows the idea behind the Folk Theorem that appears in various forms in the game-theoretic literature (Fudenberg and Tirole 1991, Dutta 1995).

The repeated game with stochastic observable payoffs offers a simple model world in which questions on the evolution of cooperation may be investigated. As we have exemplified one may also make a transition from the simpler Prisoner's Dilemma game by changing a parameter. The model captures the uncertainties on which future situations we may find our opponents and ourselves. The model may easily be extended to include noise in the form of mistakes or misunderstanding, see, e.g., (Molander 1985, Lindgren 1992, 1997). The extension of the model to a spatial setting is also

on its way.

References

R. Axelrod (1984). *The Evolution of Cooperation.* New York: Basic Books.

R. Axelrod (1987). The evolution of strategies in the iterated Prisoner's Dilemma. In L. Davis (Ed.) *Genetic Algorithms and Simulated Annealing*, pp. 32–41. Los Altos, CA: Morgan Kaufmann.

R. Boyd (1989). Mistakes allow evolutionary stability in the repeated Prisoner's Dilemma game. *Journal of Theoretical Biology* **136**, pp. 47–56.

R. Boyd and J. P. Lorberbaum (1987). No pure strategy is evolutionarily stable in the repeated Prisoner's Dilemma game. *Nature* **327**, pp. 58–59.

P. K. Dutta (1995). A Folk Theorem for Stochastic Games. *Journal of Economic Theory* **66**, pp. 1–32.

A. Eriksson and K. Lindgren (2001). In preparation.

D. Fudenberg and J. Tirole (1991). *Game Theory.* Cambridge, MA: MIT Press.

T. Ikegami (1994). From genetic evolution to emergence of game strategies. *Physica D* **75**, pp. 310–327.

W. B. Langdon (2000). Scaling of fitness spaces. *Evolutionary Computation* **7**(4), pp. 399–428.

K. Lindgren (1992). Evolutionary phenomena in simple dynamics. In C. G. Langton *et al* (Eds.), *Artificial Life II*, pp. 295-311. Redwood City, CA: Addison-Wesley.

K. Lindgren (1997). Evolutionary dynamics in game-theoretic models. In B. Arthur, S. Durlauf, and D. Lane (Eds.) pp. 337-367. *The economy as an evolving complex system II.* Addison-Wesley.

K. Lindgren and M. G. Nordahl (1994). Evolutionary dynamics of spatial games. *Physica D* **75**, 292-309.

K. Matsuo (1985). Ecological characteristics of strategic groups in 'dilemmatic world'. In *Proceedings of the IEEE International Conference on Systems and Cybernetics*, pp. 1071–1075.

J. Maynard-Smith (1982). *Evolution and the Theory of Games.* Cambridge: Cambridge University Press.

J. H. Miller (1989). The coevolution of automata in the repeated iterated Prisoner's Dilemma. Sante Fe, NM: Santa Fe Institute working paper pp. 89–003.

P. Molander (1985). The optimal level of generosity in a selfish, uncertain environment. *Journal of Conflict Resolution* **29**, pp. 611–618.

M. A. Nowak and R. M. May (1993). Evolutionary games and spatial chaos. *Nature* **359**, 826-829.

M. A. Nowak, K. Sigmund, and E. El-Sedy (1995). Automata, repeated games and noise. *Journal of Mathematical Biology* **33**, pp. 703–722.

A. Rapoport and A. M. Chammah (1965). *Prisoner's Dilemma.* Ann Arbor: University of Michigan Press.

B. Selman, H. Levesque, and D. Mitchell (1992), A New Method for Solving Hard Satisfiability Problems. *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI–92)*, San Jose, CA, pp. 440–446.

E. A. Stanley, D. Ashlock, and L. Tesfatsion (1993). Iterated Prisoner's Dilemma with choice and refusal of partners. In C. G. Langton (Ed.) *Artificial Life III*, pp. 131–175. Reading, MA: Addison-Wesley.

R. Sugden (1986). *The Economics of Rights, Cooperation and Welfare.* Oxford: Basil Blackwell.

J. Wu and R. Axelrod (1995). How to cope with noise in the iterated Prisoner's Dilemma. *Journal of Conflict Resolution* **39**, pp. 183–189.

An Ant Colony Optimization Approach to Dynamic TSP

Michael Guntsch[1], **Martin Middendorf**[2], **Hartmut Schmeck**[3]
Institute AIFB
University of Karlsruhe
D-76128 Karlsruhe, Germany
{[1]mgu,[2]mmi,[3]hsch}@aifb.uni-karlsruhe.de

Abstract

An Ant Colony Optimization (ACO) approach for a dynamic Traveling Salesperson Problem (TSP) is studied in this paper. In the dynamic version of the TSP cities can be deleted or inserted over time. Specifically, we consider replacing a certain number of cities with new ones at different frequencies. The aim of the ACO algorithm is to provide a good solution quality averaged over time, i.e. the average taken of the best solution in each iteration is optimized. Several strategies for pheromone modification in reaction to changes of the problem instance are investigated. The strategies differ in their degree of locality with respect to the position of the inserted/deleted cities and whether they keep a modified elitist ant or not.

1 Introduction

Evolutionary methods are in general capable of reacting to dynamic changes of an optimization problem. Different ways to trim Evolutionary Algorithms (EAs) for dynamic problems has have been proposed over the last years (see (Branke, 1999) for a short overview). A key aspect is whether information connected to solutions found for older stages of a problem can be used to quickly find a good solution for the problem after a change occurred.

In this paper we explore strategies to apply Ant Colony Optimization (ACO) for solving dynamic optimization problems (see (Bonabeau, 1998), (Dorigo and Di Caro, 1999) for an overview of ACO). The particular test problem we study is a dynamic Traveling Salesperson Problem (TSP) where instances change at certain intervals through the deletion and insertion of cities. This problem is general enough to be interesting as a benchmark problem and allows modifying the degree of change easily. As a practical application, consider the case of a factory with a fluctuating set of active machines. In case of a failure in the production system it is necessary to start an inspection tour immediately to check all previously active machines. This makes it beneficial to constantly know a short tour for the set of active machines.

We use modified and extended strategies from a previous study done by two of the authors where the reaction of the ant algorithms to a single change of the problem instance was investigated (Guntsch and Middendorf, 2001). The only other dynamic optimization problems to which ACO has been applied are routing problems in communication networks where the traffic in the network continually changes (e.g. (Di Caro and Dorigo, 1998), (Schoonderwoerd et al., 1996)). The ants were used to continually measure the travel time between pairs of nodes and this information is used to update the routing tables (which contain the pheromone information). This allows the system to adapt to new traffic situations but it does not provide any means for reacting explicitly to single changes.

Dynamic optimization problems are most interesting when changes of the problems instances occur frequently and each change is not too large so that it is likely that the new optimal solutions will be in some sense related to the old ones. In this case a simple restart of the algorithm which discards all old information after a change has occurred might not be a good strategy. Instead, maintenance of some previously determined knowledge in the form of pheromone information should be beneficial. To this end a tradeoff must be found between the opposing goals of preserving pheromone information and resetting enough to allow the ants to explore new relevant areas of the search space in later iterations. Based on this general idea, we propose and test three different strategies and combi-

nations thereof to make ant algorithms more suitable for the optimization in dynamic environments. Moreover, we propose an elitist strategy for use in dynamic environments. A standard elitist strategy for ant algorithms is that an elitist ant updates the pheromone values in every generation according to the best solution found so far. But after a change of the problem instance the best solution found so far will usually no longer represent a valid solution to the new instance. Instead of simply forgetting the old best solution, we study an alternative approach where the old best solution is adapted so that it becomes a reasonably good solution for the new instance.

The basic structure of the ant algorithm is presented in Section 2. In Section 3, the strategies for modifying the pheromone information are described. The test problems and the used parameter values are provided in Section 4, with the results discussed in Section 5. The paper concludes with a summary in Section 6.

2 The Ant Algorithm

In this section we describe only the general approach of our algorithm for the TSP. The strategies added for applying it to the dynamic TSP are presented later in Section 3. Ant algorithms have been applied for the (static) TSP problem by several authors (Bullnheimer et al., 1999), (Dorigo, 1992), (Dorigo et al. 1996), (Dorigo and Gambardella, 1995), (Dorigo and Gambardella, 1997), (Stützle and Hoos, 1997). Our algorithm for the TSP follows (Dorigo et al., 1996).

In every iteration each of m ants constructs one tour through all the given n cities. Starting at a random city an ant builds up a solution iteratively by always selecting the next city based on heuristic information as well as pheromone information. Pheromone information serves as a form of memory by indicating which choices were good in the past. The heuristic information, denoted by η_{ij}, and the pheromone information, denoted by τ_{ij}, are indicators of how good it seems to move from city i to city j. The heuristic value is $\eta_{ij} = 1/d_{ij}$ where d_{ij} is the distance between city i and city j.

With probability q_0, where $0 \leq q_0 < 1$ is a parameter of the algorithm, an ant at city i chooses the next city j from the set S of cities that have not been visited so far which maximizes $[\tau_{ij}]^\alpha [\eta_{ij}]^\beta$, where α and β are constants that determine the relative influence of the heuristic values and the pheromone values on the decision of the ant. With probability $1 - q_0$ the next city is chosen according to the probability distribution over S determined by

$$p_{ij} = \frac{[\tau_{ij}]^\alpha [\eta_{ij}]^\beta}{\sum_{h \in S} [\tau_{ih}]^\alpha [\eta_{ih}]^\beta}$$

Before doing a global pheromone update some of the old pheromone is evaporated on all edges according to $\tau_{ij} \mapsto (1 - \rho) \cdot \tau_{ij}$ where parameter ρ determines the evaporation rate. For pheromone update the ant that found the best solution in that generation updates pheromone along its solution, i.e. for every city i some amount of pheromone is added to element (i, j) of the pheromone matrix when j is the successor of i in the tour. Observe that pheromone is added to exactly two edges incident to a node i. The amount of pheromone added is $\rho/4$, that is $\tau_{ij} \mapsto \tau_{ij} + \frac{1}{4}\rho$. We also apply an elitist strategy where one elitist ant updates pheromone along the best solution that has been found so far.

For initialization we set $\tau_{ij} = 1/(n-1)$ for every edge (i, j). Observe, that for every city i the sum of the pheromone values on all incident edges is one, which is not changed by the pheromone update.

3 Reacting to a Change

The ant algorithm that was described in the last section can not handle the dynamic TSP. When a change of the problem instance occurred it is necessary to initialize the pheromone information for the new cities. Moreover, it might also be important to modify the pheromone information for the old cities that were not deleted. We describe three strategies and combinations thereof for resetting part of the pheromone information in reaction to a change of the problem instance. Resetting information is achieved by equalizing the pheromone values to some degree, which effectively reduces the influence of experience on the decisions an ant makes to build a solution.

Strategies for the modification of pheromone information have been proposed before to counteract stagnation of ant algorithms. In (Gambardella et al., 1999) it was proposed to reinitialize the whole pheromone matrix while (Stützle and Hoos, 1997) suggested to increase the pheromone values proportionately to their difference to the maximum pheromone value. Similar to these approaches we use a global pheromone modification strategy which reinitializes all the pheromone values by the same degree. However, this method, which we call Restart-Strategy, is limited because it does not take into account where the change of the problem instance actually occurred. Usually, the most extensive resetting of pheromone values should be performed in the close vicinity of the inserted/deleted

cities. A more locally oriented update strategy is the "η-Strategy" which uses heuristic based information, distances between cities in this case, to decide to what degree equalization is done on the pheromone values on all edges incident to a city j. The "τ-Strategy" uses pheromone based information, i.e. the pheromone values on the edges, to define another concept of "distance" between cities. Equalization of pheromone values is then again performed to a higher degree on the edges of "closer" cities.

All three strategies work by distributing a reset-value $\gamma_i \in [0:1]$ to each city i which determines the amount of reinitialization the pheromone values on edges incident to i according to the equation

$$\tau_{ij} \mapsto (1-\gamma_i)\tau_{ij} + \gamma_i \frac{1}{n-1} \qquad (1)$$

In case of a problem with symmetric η-values like Euclidean TSP, the average of the reset-values $(\gamma_i+\gamma_j)/2$ is used instead of γ_i in equation 1 for modifying the pheromone value on the edge connecting cities i and j. An inserted city i always receives an unmodifiable reset-value of $\gamma_i = 1$, resulting in all incident edges to i having the initial pheromone value of $1/(n-1)$. We will now describe in more detail how the different strategies assign the values γ_i.

3.1 Basic Strategies

The Restart-Strategy assigns each city i the strategy-specific parameter $\lambda_R \in [0,1]$ as its reset-value, i.e. $\gamma_i = \lambda_R$.

In the η-Strategy, each city i is given a value γ_i proportionate to its distance from the nearest inserted/deleted city j. This distance d^η_{ij} is derived from η_{ij} in such a way that a high η_{ij} implies a high d^η_{ij} and that scaling the heuristic η-values has no effect:

$$d^\eta_{ij} = 1 - \frac{\eta_{avg}}{\lambda_E \cdot \eta_{ij}}$$

with $\eta_{avg} = \frac{1}{n(n-1)} \sum_{i=1}^{n} \sum_{k \neq i} \eta_{ik}$ and the strategy-specific parameter $\lambda_E \in [0,\infty)$ scaling the width of the distance-cone. A city i then receives $\gamma_i = \max\{0, d^\eta_{ij}\}$ (see the example in Figure 1).

The τ-Strategy uses a distance measure based on pheromone information to calculate the reset-values. The pheromone-distance d^τ_{ik} between two cities i and k is basically defined as the maximum over all paths P_{ik} from i to k of the product of pheromone-values on the edges in P_{ik}. To prevent any incompatibility due to the size of absolute values, the pheromone-values on the edges are scaled by the maximum possible pheromone value on an edge τ_{max} [1]. Formally,

$$d^\tau_{ik} = \max_{P_{ik}} \prod_{(x,y)\in P_{ik}} \frac{\tau_{xy}}{\tau_{max}}.$$

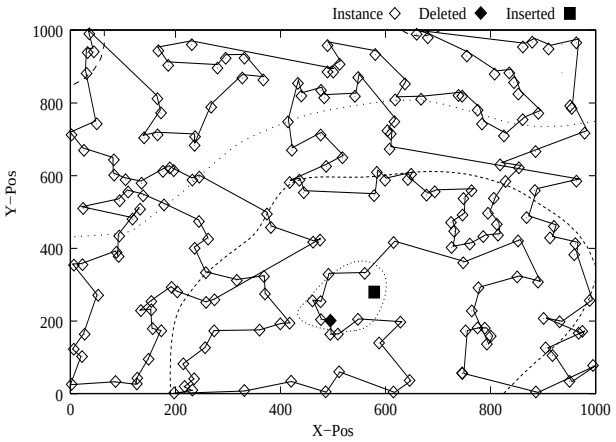


Figure 1: η-Strategy: TSP test instance with best found solution; contures show different level of reset-values - highest reset occurs near inserted/deleted city.

For the case of insertion, we set the pheromone value of the edges from the inserted city to the two closest cities, i.e. those with the highest value for η_{ij}, to τ_{max} during the application of this strategy, since the new city does not yet have any utilisable pheromone information. With J being the set of all cities that are inserted or deleted during the same change, only the maximum value $\max_{j\in J} d^\tau_{ij}$ is recored for each city i. When multiplied with a strategy-specific parameter $\lambda_T \in [0,\infty)$, with the result limited to 1 for application of equation 1, this gives the reset-value for city i: $\gamma_i = min\{1, \lambda_T \cdot d^\tau_{ij}\}$.

3.2 Combined Strategies

A combination of the global Restart-Strategy with one of the two more locally acting η- or τ-Strategies could be advantageous in a situation where strong local resetting near the inserted/deleted cities is necessary to incorporate a change while a lower global resetting is needed to maintain the flexibility for the algorithm to change the best tour found more strongly if beneficial. This combination can be realized by having each of the two strategies involved distribute reset-values according to their respective scheme and then

[1] τ_{max} is 0.5 for symmetric and 1.0 for asymmetric TSP.

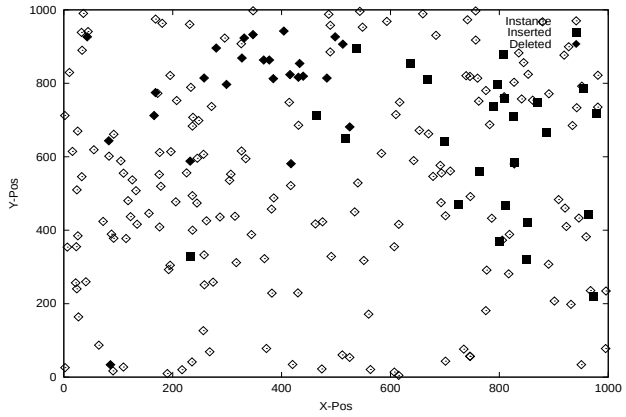


Figure 2: TSP test instance with $k = 25$ deleted and inserted cities determined with $p = 2$.

choosing for each city i the maximum of the two reset-values determined by the two strategies. Formally, if the first strategy distributes $\gamma_{i,1}$ and the second $\gamma_{i,2}$, then $\gamma_i = \max\{\gamma_{i,1}, \gamma_{i,2}\}$.

3.3 Keeping the Elitist Ant

Whenever a change to the instance that the algorithm is running on occurs, the elitist ant, which enforces the best solution found so far, no longer represents a valid solution. Consequently, it must be dropped and a new elitist ant is determined after the first iteration of ants has worked on the changed instance. The possible loss of information from dropping the old best solution can be alleviated somewhat by modifying the former best tour so that it once again yields a valid and presumably good solution to the changed instance. We use two greedy heuristics for performing this modification: i) all cities that were deleted from the instance are also deleted from the old best tour, effectively connecting their respective predecessors and successors, ii) the cities that were added are inserted individually into the tour at the place where they cause the minimum increase in length. The tour derived from this process is the new tour of the elitist ant. We call this method KeepElitist. Clearly, this modification can be combined with the strategies explained above.

4 Test Setup

For our tests we chose subproblems of the Euclidean TSP instance `rd400` from the (TSP-Library, 2001). Specifically, 200 random cities were taken away from the 400 making up the problem instance to form a spare pool of cities before the start of the algorithm, leaving the instance with 200 cities. During the run of the algorithm the actual problem instance was changed every t iterations by exchanging k cities between the actual instance and the spare pool, i.e. k cities were deleted from the actual instance and the same number of cities from the spare pool were inserted. When deciding which cities to delete, the first city j was chosen at random and all other cities i according to a probability distribution defined by η_{ij}^p, with p being a parameter that determines the relative influence of the distances between i and j. The cities that were inserted were chosen analogously from the spare pool. An example is shown in Figure 2.

We tested all combinations of parameter values $k \in \{1, 5, 25\}$, $t \in \{50, 200, 750\}$, and $p \in \{0.0, 2.0\}$. Note that for $k = 1$, the parameter p has no effect as only one city is removed/inserted. For each configuration (k, t, p), 10 test runs of 8999 iterations were done (in iteration 9000, the next change would occur for all tested t), each starting with a different random subset of 200 cities. All results that were used as a basis for comparison are averages over these 10 runs. Only the results during iterations 3000-8999 were used to measure the performance of the applied strategies, since the behaviour of the ant algorithm during the first iterations is not representative for the latter stages.

The parameter values for the ant algorithm used in the tests were $m = 10$ ants, $\alpha = 1$, $\beta = 5$, $q_0 = 0.9$, and $\rho = 0.05$. The heuristic weight of $\beta = 5$ has been used by several authors (e.g. (Bullnheimer at al. 1999), (Stützle and Hoos, 1997)) for TSP.

We tested the parameters $\lambda_R \in \{0.25, 0.5, 0.75, 1.0\}$ for the Restart-Strategy, $\lambda_E \in \{0.5, 1.0, 2.0, 5.0\}$ for the η-Strategy, and $\lambda_T \in \{0.5, 1.0, 1.5, 2.0\}$ for the τ-Strategy. A parameter value of 0.0, which is equivalent for all strategies and corresponds to not applying the strategy at all, was also tested. Furthermore, we combined the Restart-Strategy with $\lambda_R \in \{0.25, 0.5\}$ with the η- and τ-Strategy using their respective parameter-values above to determine if such a combination can yield better results than the "pure" strategies by itself. Finally, all of the above settings were tested with and without keeping a modified elite ant as described in Section 3.3 after the exchange of cities.

Besides the best solutions found by the ant algorithm we also recorded the normalized entropy $E \in [0, 1]$ of the pheromone matrix in every iteration, which is defined as

$$E = \frac{1}{n \log n} \sum_{i=1}^{n} \sum_{j=1}^{n} -\tau_{ij} \log(\tau_{ij})$$

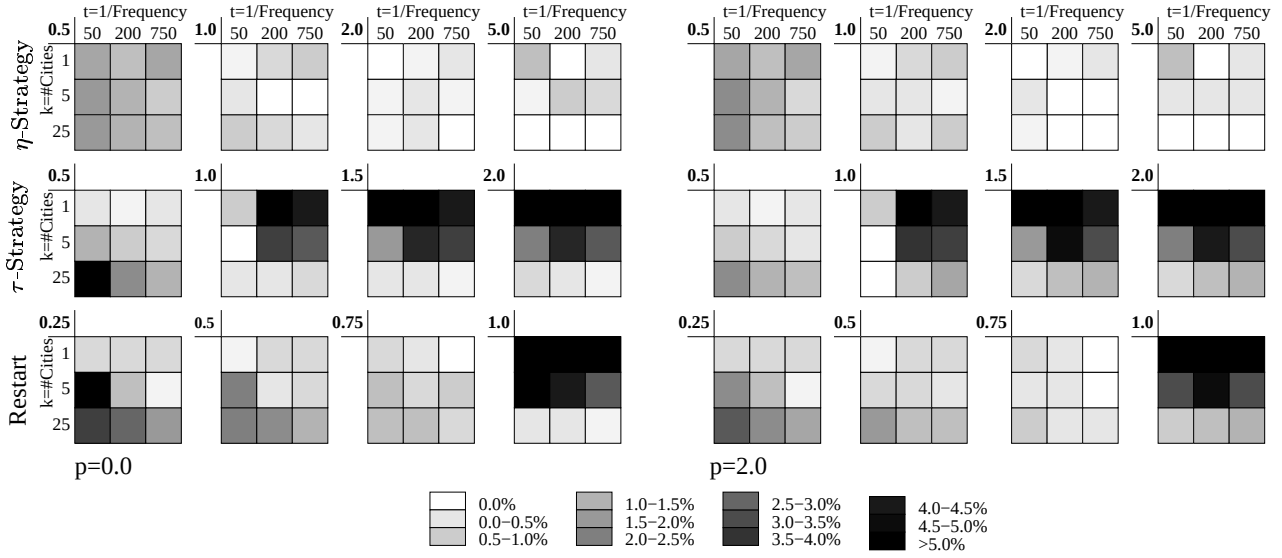


Figure 3: Relative performance of Restart-, η- and τ-strategy for $p = 0$ (left) and $p = 2$ (right) and different values of k and t: loss in quality of the best found solution averaged over iterations 3000-8999 compared to the best performing variant.

Normalized entropy has been used previously in (Guntsch and Middendorf, 2001) to help understand the current state of the ant algorithm for a given iteration.

5 Empirical Evaluation

A comparison of the Restart-, η- and τ-Strategies for the respective parameter values described in Section 4 is shown Figure 3. Judging from "average darkness", the best overall strategy is the the η-Strategy with a parameter $\lambda_E = 2.0$, especially for a high degree of proximity for the cities inserted and removed. The τ-Strategy with $\lambda_T = 1.0$ provides good to very good solutions when changes occur quickly, i.e. for $t = 50$. The Restart-Strategy, when given enough time and not confronted with changes that are too severe, is also able to achieve good solutions for $\lambda_R = 0.75$. A complete restart, i.e. using the Restart-Strategy with $\lambda_R = 1.0$, is only comparable to the other strategies for the cases where many cities are exchanged, even beating some of the other strategies when they do not reset enough information. This would likely increase if even more cities were transferred as the changed problems would become almost independent of one another.

As for the influence of the proximity-value p, is seems that the difference in the solution-quality achieved by the individual strategies becomes less for $p = 2$ compared to $p = 0$. For the local strategies, a stronger proximity of the exchanged cities is beneficial because a cluster of cities being inserted or deleted will cause a distribution of reset-values that is not as much dependent on the number of cities comprising the cluster as on their position in the graph or their degree of connectivity in the pheromone matrix. Therefore, although the transfer of cities might be large, the local confinement of this change makes it easier to incorporate for the local strategies. The Restart-Strategy, however, also benefits from a higher degree of proximity. This is probably again due to the "bad" pheromone information being more centralized than for the case of equal distribution, and therefore easier to deal with for the ant algorithm.

Figure 4 shows a more detailed view of the optimization behavior for the individual strategies and its dependency on their respective λ-parameters (λ_E, λ_T, λ_R) for the case of $(k, t, p) = (1, 50, 0)$, i.e. frequent occurring small changes. In particular, the effect on increasing the λ-parameter-values can be seen. The η-Strategy only slowly becomes worse in terms of solution quality, despite resetting a lot of pheromone for high values of λ_E, as is indicated by the entropy-curves in Figure 4. In contrast, the τ-Strategy shows a significant loss of performance for $\lambda_T > 1$, even though the entropy curve indicates that the increase in reset information is only moderate. For the Restart-Strategy, we see U-shaped parameter-dependency curves for resulting solution quality, which shows that not resetting enough as well as resetting too much information has

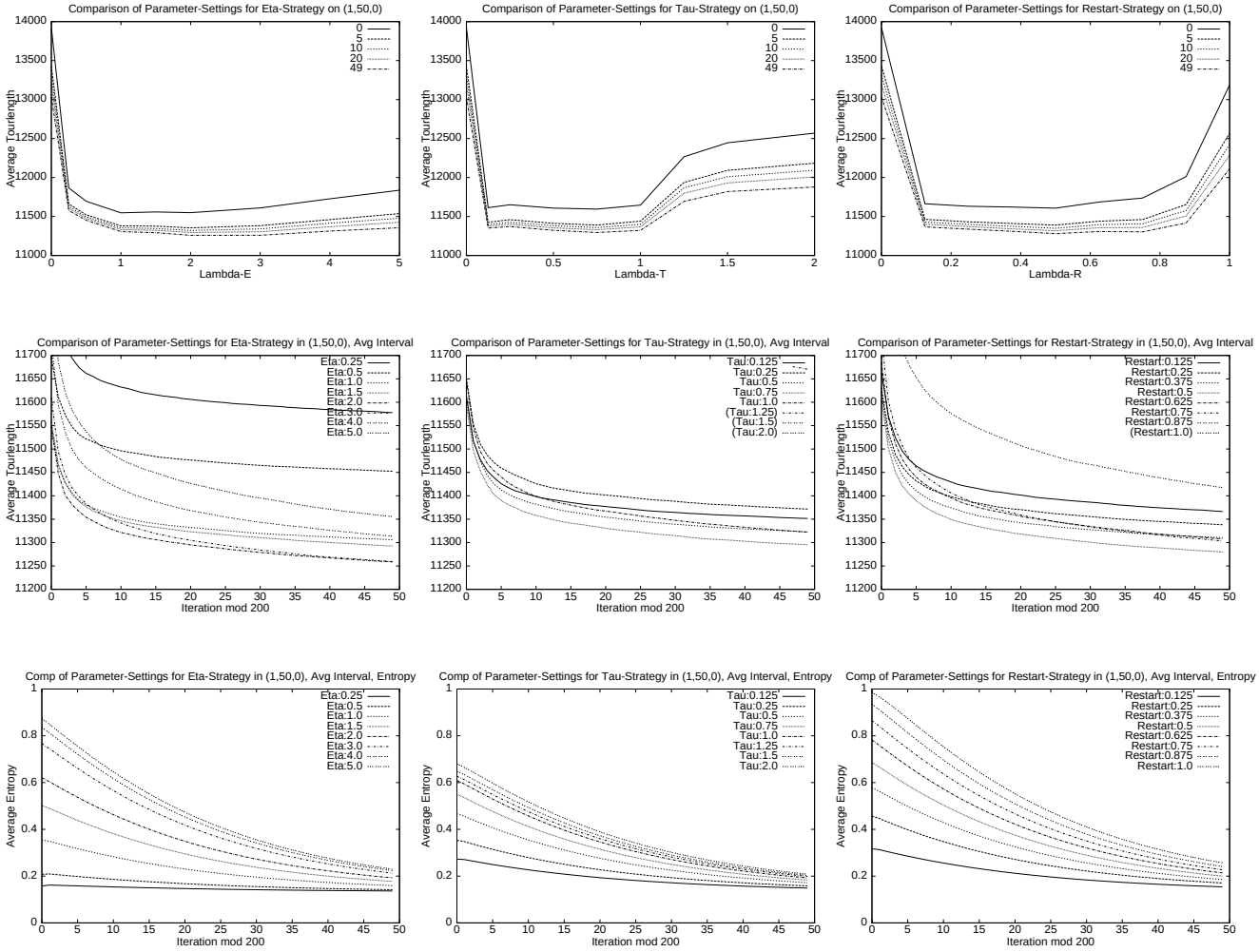
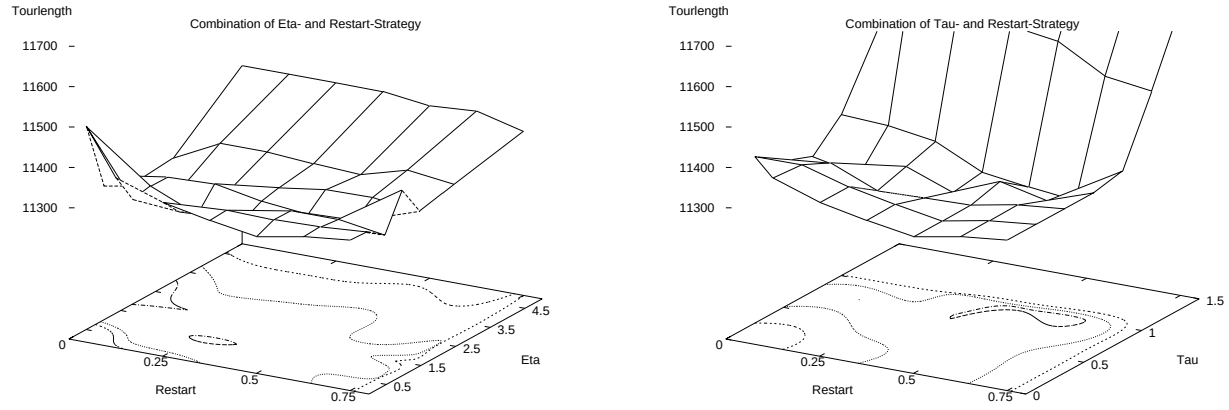


Figure 4: Average quality of best found solution with respect to number of iteration after a change forη-Strategy (left), τ-Strategy (center), and Restart-Strategy (right) for different parameter values $\lambda_E, \lambda_T, \lambda_R$ and the configuration $(k, t, p) = (1, 50, 0)$. The upper row shows curves of average solution quality for certain iterations after a change, the middle row the averaged behavior for a certain parameter over a t-Interval, and the lower row shows the corresponding entropy values to the middle row.

a negative effect on the derived solution. As with the local strategies, the biggest performance gain can be observed when going from doing nothing, i.e. using a parameter-value of 0.0, to doing even just a little, i.e. setting $\lambda_E = 0.25$, $\lambda_T = 0.125$, and $\lambda_R = 0.125$. The curves for the Restart-Strategy also show that the difference from resetting almost all pheromone information to actually resetting all of it is enormous in terms of solution quality when using this strategy.

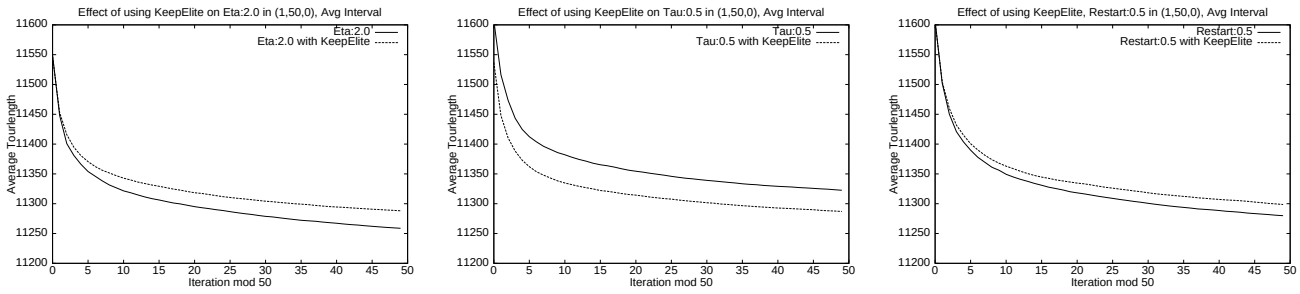
As mentioned in Section 4, we also analyzed the performance of combinations of the local η- and τ-Strategies with the Restart-Strategy. For some cases, this combination provided better solutions than any of the strategies could achieve by itself. An example of this is the configuration $(k, t, p) = (1, 50, 0)$ shown in Figure 5, for which we performed additional parameter-tests to make a more precise analysis. The contour lines for the combination of the η- and Restart-Strategy show that there are two areas in which good performance was achieved, one of them a true combination with $\lambda_E = 1$ and $\lambda_R = 0.25$, and the other one, which is better in terms of solution quality as well as larger, with $\lambda_E \in \{2, 3\}$ and $\lambda_R = 0$. This suggests that the η-Strategy does not benefit much, if at all, from being combined with Restart. For the τ-Strategy on the other hand we see a promising area located around a combination of medium λ_T and λ_R values, specif-



(a) η-Restart Combination

(b) τ-Restart Combination

Figure 5: Combinations of the local η- and τ-Strategies with the Restart-Strategy for the problem-configuration $(k, t, p) = (1, 50, 0)$.



(a) η-Strategy

(b) τ-Strategy

(c) Restart-Strategy

Figure 6: Combination of the individual strategies with KeepElitist for the configuration $(k, t, p) = (1, 50, 0)$.

ically for $\lambda_T = 1.0$ and $\lambda_R \in \{0.5, 0.675\}$, and also for $\lambda_T = 0.75$ and $\lambda_R = 0.375$. Thus the combination of the τ- and Restart-Strategy performs better than either strategy by itself, and also better than the η-Strategy in this case, justifying its application.

Finally, we combined the KeepElitist method with the individual strategies as well as the combinations of the η- and τ-Strategies with the Restart-Strategy. Figure 6 shows how this modified the average behavior of the pure strategies with their respectively best λ-parameter on configuration $(k, t, p) = (1, 50, 0)$. As can be observed, for the η- and Restart-Strategy the combination on average entailed a worse solution, while for the τ-Strategy the effect on average was an improvement. Overall, the heuristic of keeping a modified elite ant was beneficial only when the number of cities k that was inserted and deleted was not too large and when the time for adapting to the problem t was small. If too many cities were exchanged, then the heuristic would no longer provide a good solution, and the ants would find a better solution in the first iteration after the change. This case is not dangerous, since keeping the modified elitist ant would be the same as not keeping it; only the "new" elitist ant would update the pheromone matrix. The second case in which keeping an elitist ant does not entail better solutions is when the interval t between changes is long enough to permit the algorithm to adapt very well to the new instance, and the guidance provided by an early good solution leads toward stagnation in the end. This case is potentially dangerous, as the elitist ant survives the first generation(s) and influences the pheromone ma-

trix, thereby restricting the search space to a region that is perhaps not very promising.

6 Conclusion

In this paper, we studied strategies for helping ant algorithms deal with a highly dynamic TSP. We modified three strategies proposed for the case of a single change of the problem instance. Using combinations and a heuristic for keeping a modified elitist ant, we were able to find better solutions for various problem classes than the pure strategies by themselves. We have also distinguished what type of problem classes seem to favor which strategy for dealing with changes, and what type of parameter to use for the different strategies in such a case.

Overall, we have shown empirically that the local strategies perform best when problem changes occur frequently so that the algorithm does not have enough time to reset "blindly" and reoptimize the entire instance. Future work could clarify where exactly the boundary lies between sensible local resetting and global resetting. Also, the state of convergence that the ant algorithm has achieved could codetermine the ideal strength of resetting in reaction to a change. Lastly, it might be that the strategies for resetting pheromone could successfully be applied in a static environment when stagnation of the search process is imminent for the entire instance or parts of it.

References

B. Bullnheimer, R.F. Hartl, and C. Strauss (1999). A New Rank Based Version of the Ant System - A Computational Study. *Central European Journal of Operations Research* **7**: 25-38.

E. Bonabeau, M. Dorigo, and G. Theraulaz (1999). *Swarm Intelligence: From Natural to Artificial Systems*, Oxford University Press, New York.

J. Branke (1999). Evolutionary approaches to dynamic optimization problems - a suvey. In A. Wu (ed.) GECCO Workshop on Evolutionary Algorithms for Dynamic Optimization Problems, 134-137. San Mateo, CA: Morgan Kaufmann.

G. Di Caro and M. Dorigo (1998). AntNet: Distributed Stigmergetic Control for Communications Networks. *Journal of Artificial Intelligence Research* **9**: 317-365.

M. Dorigo (1992). Optimization, Learning and Natural Algorithms (in Italian). PhD Thesis, Dipartimento di Elettronica, Politecnico di Milano, Italy, pp.140.

M. Dorigo and G. Di Caro (1999). The ant colony optimization meta-heuristic. In D. Corne, M. Dorigo, F. Glover (eds.), *New Ideas in Optimization*, 11-32, McGraw-Hill.

M. Dorigo and L.M. Gambardella (1995). Ant-Q: A Reinforcement Learning approach to the traveling salesman problem. In *Proceedings of ML-95, Twelfth International Conference on Machine Learning*, 252-260. San Mateo, CA: Morgan Kaufmann.

M. Dorigo and L.M. Gambardella (1997). Ant colony system: A cooperative learning approach to the travelling salesman problem," *IEEE Transactions on Evolutionary Computation* **1**: 53-66.

M. Dorigo, V. Maniezzo, and A. Colorni (1996). The Ant System: Optimization by a Colony of Cooperating Agents. *IEEE Trans. Systems, Man, and Cybernetics – Part B* **26**: 29-41.

L.M. Gambardella, E.D. Taillard, and M. Dorigo (1999). Ant Colonies for the Quadratic Assignment Problem. *Journal of the Operational Research Society* **50**: 167-176.

M. Guntsch and M. Middendorf (2001). Pheromone Modification Strategies for Ant Algorithms applied to Dynamic TSP. To appear in Proceedings of EvoWorkshops 2001 - First European Workshop on Evolutionary Computation in Combinatorial Optimization (EvoCop2001), Lake Como, Italy. Springer LNCS Series.

D. Merkle, M. Middendorf, and H. Schmeck (2000). Ant Colony Optimization for Resource-Constrained Project Scheduling. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000)*, 893-900. San Mateo, CA: Morgan Kaufmann.

R. Schoonderwoerd, O. Holland, J. Bruten, and L. Rothkrantz (1996). Ant-based Load Balancing in Telecommunications Networks. *Adaptive Behavior* **5**: 168-207.

T. Stützle and H. Hoos (1997). Improvements on the ant system: Introducing MAX(MIN) ant system. In G. D. Smith et al. (eds.), *Proc. of the International Conference on Artificial Neutral Networks and Genetic Algorithms*, 245-249. Springer-Verlag.

T. Stützle and H. Hoos (1999). *MAX-MIN* Ant System. *Future Generation Computer Systems* **16**: 889-914.

TSP-Library (2001). http://www.iwr.uni-heidelberg.de/iwr/comopt/software/TSPLIB95/.

Body-Brain Co-evolution Using L-systems as a Generative Encoding

Gregory S. Hornby
Computer Science Department
Brandeis University
Waltham, MA 02454-9110
hornby@cs.brandeis.edu

Jordan B. Pollack
Computer Science Department
Brandeis University
Waltham, MA 02454-9110
pollack@cs.brandeis.edu

Abstract

We co-evolve the morphology and controller of artificial creatures using two integrated generative processes. L-systems are used as the common generative encoding for both body and brain. Combining the languages of both into a single L-system allows for linkage between the genotype of the controller and the parts of the morphology that it controls. Creatures evolved by this system are more complex than previous work, having an order of magnitude more parts and a higher degree of regularity.

1 INTRODUCTION

As computers become more powerful the richness of virtual worlds is limited only by what can be designed to inhabit them. How can we construct beautiful and complex designs, objects and creatures for them? 3D virtual creatures have been evolved in simulation [Komosinski & Rotaru-Varga, 2000], and simple robots have been evolved for automatic manufacture [Lipson & Pollack, 2000]. Both of these works have used a direct encoding for the creature morphology and controller. In this we return to the spirit of [Sims, 1994], in which a graph structure was the generative encoding evolved for creating both the body and brain of virtual creatures.

Previously we showed that an evolutionary algorithm (EA) using a Lindenmayer system (L-system) as a generative encoding outperformed an EA using a non-generative encoding on an automated design problem [Hornby & Pollack, 2001]. We then used this system to evolve complex morphologies for 2D robots with motorized joints, each controlled by an oscillator [Hornby et al., 2001]. Here we describe extensions of this work to 3D creatures and to the integration of neural networks as controllers. Advantages of neural networks are that they can generate more complex locomotion patterns than unconnected oscillators and allow for later progression to the evolution of morphologies with sensors and reactive controllers. Integrating the commands for morphology and controller in the genotype creates a linkage between them, like the encoding of [Sims, 1994], and should reduce disruption under recombination.

L-systems have been used previously for the development of artificial neural networks. In [Kitano, 1990] an L-system on matrices was used to generate the connectivity matrix of a network. This method does not naturally extend to the co-evolution of morphology along with the neural controller. More compatible with our original system of producing a string of build commands is the technique of [Boers & Kuiper, 1992]. Here groupings of symbols inside brackets are used to specify connectivity of the network. Drawbacks to this system are that a symbol is used for each neuron, which limits its ability to scale to large networks. Our system creates networks in a method similar to that of cellular encoding [Gruau, 1994], with operators acting on links instead of on the nodes, as in [Luke & Spector, 1996].

Using this system we evolve both the neural controllers and morphologies of creatures for locomotion. Whereas the generative encoding of [Sims, 1994] allows for repetition of segments, it did not produce hierarchies of regularity. Our generative encoding system is a more powerful language with loops, sub-procedure-like elements, parameters and conditionals and achieves an order of magnitude more parts than the previous work of [Komosinski & Rotaru-Varga, 2000], [Lipson & Pollack, 2000] and [Sims, 1994].

In the following sections we first outline the design space and describe the components of our generative

design system, then we present our results and finally close with a discussion and conclusion of our work.

2 EXPERIMENTAL METHOD

In these experiments we evolve a generative genotype that specifies how to construct both the morphology and controller of a locomoting creature. A Lindenmayer system (L-system) is used as the generative specification system for both body and brain and is optimized by an evolutionary algorithm (EA). The system consists of the network and morphology constructor, the L-system parser, the evolutionary algorithm and the physics and network simulator.

2.1 MORPHOLOGY CONSTRUCTOR

The morphology constructor and simulator is a 3D extension of the 2D work in [Hornby et al., 2001]. The morphology constructor builds a model from a string of build commands to a LOGO-style turtle [Abelson & deSessa, 1982] using a command language similar to that of L-system languages for creating plants [Prusinkiewicz & Lindenmayer, 1990]. As the turtle moves, bars are created and these become the morphology of the creature. The commands instruct the turtle to move forward or backward and to change orientation, and there are commands for creating actuated joints.

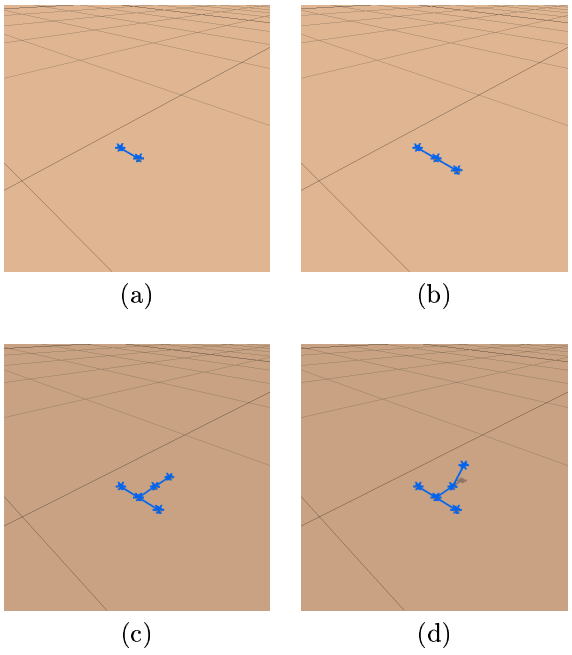


Figure 1: Building And Simulating A 3D Creature

The commands for this language are: '[' ']', store/retrieve the current state (consisting of the current location and orientation) to and from a stack; { *block* }(n), repeat the enclosed block of build commands n times; *forward*, moves the turtle forward in the current direction, creating a bar if none exists or traversing to the end of the existing bar; *backward*, goes back up the parent of the current bar; *revolute-1*, forward, end with a joint with range 0° to 90° about the current Z-axis; *revolute-2*, forward, end with a joint with range −45° to 45° about the current Z-axis; *twist-90*, forward, end with a joint with range 0° to 90° about the current X-axis; *twist-180*, forward, end with a joint with range −90° to 90° about the current X-axis; *up*(n), rotate heading $n \times 90°$ about the turtle's Z axis; *down*(n), rotate heading $n \times -90°$ about the turtle's Z axis; *left*(n), rotate heading $n \times 90°$ about the turtle's Y axis; *right*(n), rotate heading $n \times -90°$ about the turtle's Y axis; *clockwise*(n), rotate heading $n \times 90°$ about the turtle's X axis; and *counter-clockwise*(n), rotate heading $n \times -90°$ about the turtle's X axis.

When evolving oscillating motors, and not neural controllers, the oscillation function of a joint is specified as in [Hornby et al., 2001] by adding a parameter to *revolute-1*, *revolute-2*, *twist-90*, and *twist-180* to specify the rate of oscillation and by adding the following two commands to control the phase off: *increase-offset*(n), increase phase offset by $n \times 25\%$ and *decrease-offset*(n), decrease phase offset by $n \times 25\%$.

An example of a creature constructed using this language is shown in figure 1. The single bar in figure 1.a is built from the string, *[left(1) forward(1)]*, and the two bar structure in figure 1.b is built from, *[left(1) forward(1)] [right(1) forward(1)]*. The final creature is made from the command sequence, *[left(1) forward(1)] [right(1) forward(1)] revolute-1(1) forward(1)*, and is shown in figure 1.c. Figure 1.d displays the creature with the actuated joint moved half-way through its joint range.

2.2 NETWORK CONSTRUCTOR

The method for constructing the neural controllers for the artificial creatures is similar to that of cellular encoding [Gruau, 1994], with two main differences. The first difference between this work and that of cellular encoding is that strings of build commands are used instead of trees of build commands, although the *push* and *pop* operators (described later) add a branching ability to the strings. The other difference is that build commands operate on the links connecting the nodes as with edge encoding [Luke & Spector, 1996] instead of on the nodes of the network. Advantages of edge

encoding are that at most one link is created with a build command so each build command can specify the weight to attach to that link and, unlike cellular encoding, sub-sequences of build commands will construct the same sub-network independent of where in the build-tree they are located.

Commands for constructing the network operate on links between neurons and use the most recently created link as the current one. *Push* and *pop* operators, '[' and ']', are used to store and retrieve the current link – consisting of the from-neuron, the to-neuron and index of the link into the to-neuron (for when there are multiple links between neurons) – to and from the stack. This stack of edges allows a form of branching to occur in an encoding – an edge can be pushed onto the stack followed by a sequence of commands and then a pop command makes the original edge the current edge again. For the following list of commands the current link connects from neuron A to neuron B.

- decrease-weight(n) – Subtracts n from the weight of the current link. If the current link is a virtual link, it creates it with weight $-n$.
- duplicate(n) – Creates a new link from neuron A to neuron B with weight n.
- increase-weight(n) – Add n to the weight of the current link. If the current link is a virtual link, it creates it with weight n.
- loop(n) – Creates a new link from neuron B to itself with weight n.
- merge(n) – Merges neuron A into neuron B by copying all inputs of A as inputs to B and replacing all occurrences of neuron A as an input with neuron B. The current link then becomes the nth input into neuron B.
- next(n) – Changes the from-neuron in the current link to its nth sibling.
- output(n) – Creates an output-neuron, with a linear transfer function, from the current from-neuron with weight n. The current-link continues to be from neuron A to neuron B.
- parent(n) – Changes the from-neuron in the current link to the nth input-neuron of the current from-neuron. Often there will not be an actual link between the new from-neuron and to-neuron, in which case a virtual link of weight 0 is used.
- reverse – Deletes the current link and replaces it with a link from B to A with the same weight as the original.

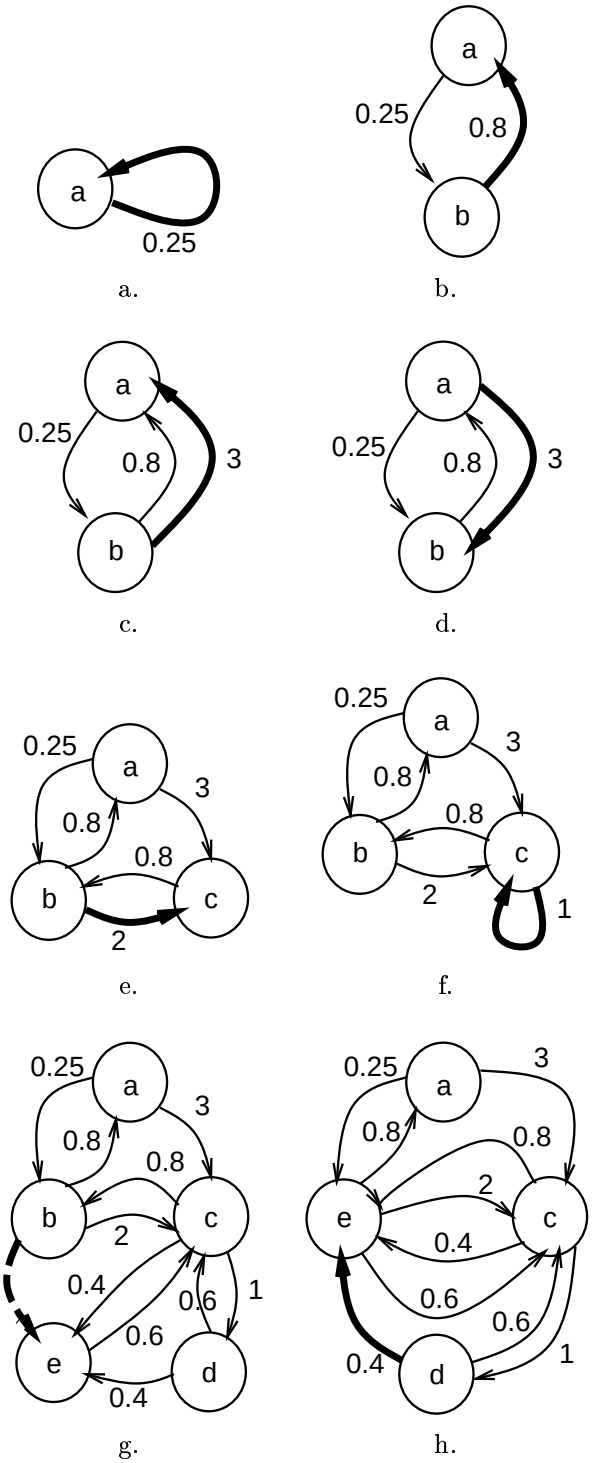


Figure 2: Constructing A Network

- set-function(n) – Changes the transfer function of the to-neuron in the current link, B, with: 0, for sigmoid; 1, linear; and 2, for oscillator.

- split(n) – Creates a new neuron, C, with a sigmoid transfer function, and moves the current link from A to C and creates a new link connecting from neuron C to neuron B with weight n.

The sequence of networks in figure 2 are intermediate networks in parsing, *split(0.8) duplicate(3) reverse split(0.8) duplicate(2) reverse loop(1) split(0.6) duplicate(0.4) split(0.6) duplicate(0.4) reverse parent(1) merge(1)*. Networks start with a single neuron, a, with an *oscillator* function, which has a single link, of weight 0.25, feeding to itself, figure 2.a. The sequence of intermediate networks after: *split(0.8)*, is b; *duplicate(3)*, is c; *reverse*, is d; *split(0.8) duplicate(2) reverse*, is e; *loop(1)*, is f; *split(0.6) duplicate(0.4) split(0.6) duplicate(0.4)* (which results in the current link being a virtual link from neurons b to e), is g; and after *merge(1)*, the final network is shown in h.

Neurons in the network are initialized to an output value of 0.0 and are updated sequentially by applying a transfer function to the weighted sum of their inputs with their outputs clipped to the range ± 1. The different transfer functions are: a sigmoid, using $tanh(sum\ of\ inputs)$; linear; and an oscillator. The oscillator maintains a state value, which it increases by 0.01 each update. Its output is its state value plus the weighted sum of its inputs mapped to a triangle-wave function with period 4 and a minimum of -1 and maximum of 1. Use of an oscillator increases the bias towards networks whose outputs cycle over the sigmoid-only networks used in [Komosinski & Rotaru-Varga, 2000, Lipson & Pollack, 2000] but is a more simple model than that of [Sims, 1994] which had a variety of transfer functions and oscillating neurons.

2.3 COMBINING BODY AND BRAIN

To simultaneously create a creature's neural controller and morphology, the languages for constructing a neural network and for constructing a body are combined. When processing the command string, a neural-construction command affects the construction of the neural controller and a morphology-construction command affects the construction of the body, with a few modifications. *Push* and *pop* operators, '[' and ']' are used to store and retrieve the current construction state, which now consists of the current link and the current location and orientation on the body. To give the neural controller control of the body, each time a joint command (*revolute-1, revolute-2, twist-90 or twist-180*) is executed, the neural-command *output(1)* is also called. This output neuron then controls the joint angle of the actuated joint.

Once a string of build commands has been executed and the resulting creature is constructed, its behavior is evaluated in a quasi-static kinematics simulator, similar to that used by [Lipson & Pollack, 2000]. First the neural network is updated to determine the desired angles of each actuated joint. Then the kinematics are simulated by computing successive frames of moving joints in small angular increments of at most 0.06°. After each update the structure is then settled by determining whether or not the creature's center of mass falls outside its footprint and then repeatedly rotating the entire structure about the edge of the footprint nearest the center of mass until it is stable.

2.4 PARAMETRIC L-SYSTEMS

The strings of build commands are generated by a context-free, parametric Lindenmayer-system (P0L-system). L-systems are a grammatical rewriting system introduced to model the biological development of multicellular organisms [Lindenmayer, 1968]. Rules are applied in parallel to all characters in the string just as cell divisions happen in parallel in multicellular organisms. For example, the L-system,

$$\begin{aligned} a: &\ \rightarrow a\ b \\ b: &\ \rightarrow b\ a \end{aligned}$$

if started with the symbol a, produces the following strings,

$$a$$
$$ab$$
$$abba$$
$$abbabaab$$

A parametric L-system [Lindenmayer, 1974] is a class of L-systems in which production rules have parameters and algebraic expressions can be applied when parameter values to successors. Parameter values can also be used in determining which production rule to apply. For example, the P0L-system,

$$\begin{aligned} a(n): &\ (n > 1) &&\rightarrow a(n-1)\ b(n) \\ a(n): &\ (n \leq 1) &&\rightarrow a(0) \\ b(n): &\ (n > 2) &&\rightarrow b(n/2)\ a(n-1) \\ b(n): &\ (n \leq 2) &&\rightarrow b(0) \end{aligned}$$

When started with $a(4)$, the P0L-system produces the following sequence of strings,

$$a(4)$$
$$a(3)b(4)$$
$$a(2)b(3)b(2)a(3)$$
$$a(1)b(2)b(1.5)a(2)b(0)a(2)b(3)$$
$$a(0)b(0)b(0)a(1)b(2)b(0)a(1)b(2)b(1.5)a(2)$$
$$a(0)b(0)b(0)a(0)b(0)b(0)a(0)b(0)b(0)a(1)b(2)$$
$$a(0)b(0)b(0)a(0)b(0)b(0)a(0)b(0)b(0)a(0)b(0)$$

Advantages of a parametric L-system are a P0L-system can produce a family of structures, with the specific structure created being determined by the starting parameters. Similarly, parameters can be used so that repeating patterns of connections will have different weights. An example of a P0L-system for a network is,

$P0(n0)$:
$n0 > 1.0 \rightarrow \quad P1(n0)\ P0(n0-1)$
$n0 > 0.0 \rightarrow \quad loop(1)\ P1(1)\ parent(1)\ merge(1)$

$P1(n0)$:
$n0 > 1.0 \rightarrow \quad split(0.8)\ duplicate(n0)\ reverse$
$n0 > 0.0 \rightarrow \quad \{split(0.6)\ duplicate(0.4)\}(2)\ reverse$

This L-system consists of two productions, each containing two condition-successor pairs and when started with P0(3) produces the sequence of four strings: *a*, *P1(3) P0(2)*; *b*, *split(0.8) duplicate(3) reverse P1(2) P0(1)*; *c*, *split(0.8) duplicate(3) reverse split(0.8) duplicate(2) reverse loop(1) P1(1) parent(1) merge(1)*; and *d*, *split(0.8) duplicate(3) reverse split(0.8) duplicate(2) reverse loop(1) {split(0.6) duplicate(0.4)} reverse parent(1) merge(1)*. This last is interpreted as: *split(0.8) duplicate(3) reverse split(0.8) duplicate(2) reverse loop(1) split(0.6) duplicate(0.4) split(0.6) duplicate(0.4) reverse parent(1) merge(1)*, and the network that it constructs is shown in figure 2.h.

2.5 EVOLUTIONARY ALGORITHM

The evolutionary algorithm used to evolve L-systems is the same as [Hornby & Pollack, 2001]. The initial population of L-systems is created by making random production rules. Evolution then proceeds by iteratively selecting a collection of individuals with high fitness for parents and using them to create a new population of individual L-systems by applying mutation or recombination. Mutation creates a new individual by copying the parent individual and making a small change to it. Changes that can occur are: replacing one command with another; perturbing the parameter to a command by adding/subtracting a small value to it; changing the parameter equation to a production; adding/deleting a sequence of commands in a successor; or changing the condition equation. Recombination takes two individuals, $p1$ and $p2$, as parents and creates one child individual, c, by making it a copy of $p1$ and then inserting a small part of $p2$ into it. This is done by replacing one successor of c with a successor of $p2$, inserting a sub-sequence of commands from a successor in $p2$ into c, or replacing a sub-sequence of commands in a successor of c with a sub-sequence of commands from a successor in $p2$.

3 EXPERIMENTAL RESULTS

In this section we present results in evolving 3D locomoting creatures using both oscillating joints as controllers and using neural networks as controllers. To evolve creatures that locomote we set their fitness to be a function of the distance moved by the creature's center of mass less the distance ground points were dragged along the ground – this penalty encourages creatures to evolve stepping or rolling motions over sliding motions.

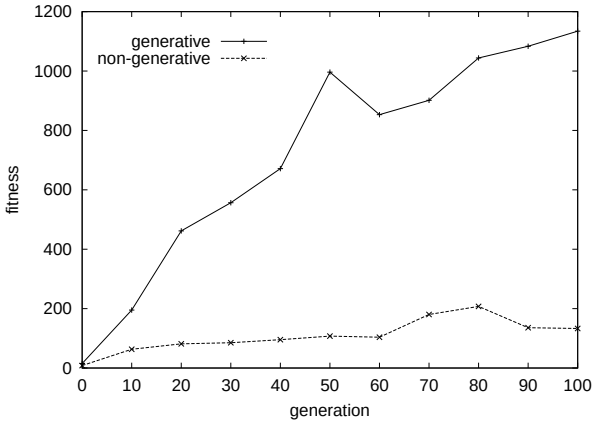
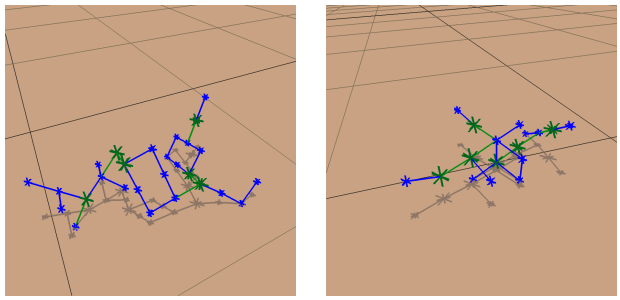


Figure 3: Generative vs. Non-generative Encodings

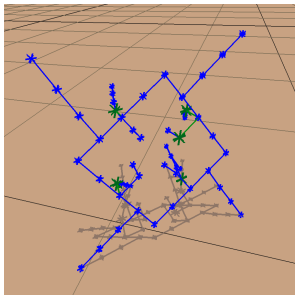


(a) (b)

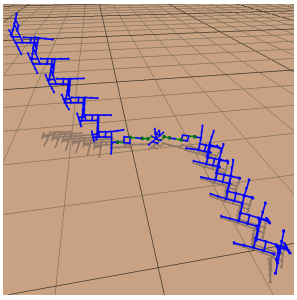
Figure 4: Results with Non-generative Encoding

Initially we ran two sets of experiments to compare a generative encoding against a non-generative encoding. In these experiments we added the constraint that creatures could not have a sequence of more than 4 bars in a row that was not part of a cycle as a representative limit to physically plausible creatures while not providing any shaping bias[1]. The evolutionary algorithm was configured to run with a population of

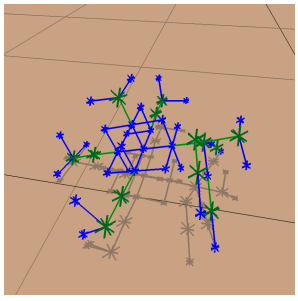
[1]In a true dynamics simulator actual torques on joints would be calculated and then a constraint on the allowable torque could be used.



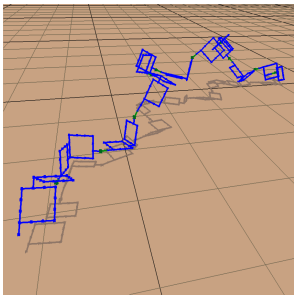
(a)



(b)

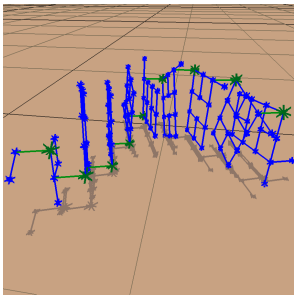


(c)

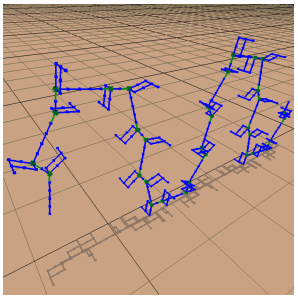


(d)

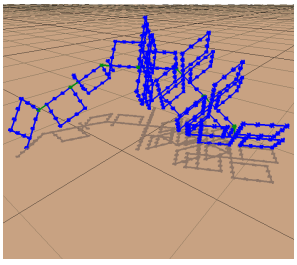
Figure 5: Results with Generative Encoding



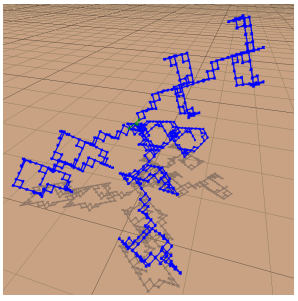
(a)



(b)



(c)



(d)

Figure 6: Other Oscillator Creatures

100 individuals for 100 generations and morphologies had an upper limit of 350 bars. The non-generative encoding was allowed to use up to 10000 build commands and the generative encoding used 2 parameters and 15 production rules, with 2 condition-successor pairs for each production rule, with each successor having a maximum of 20 build commands. 10 trials were run with each encoding type, and the average of the fittest individual found at each generation is plotted in the graph in figure 3. Two individuals evolved using the non-generative encoding are shown in figure 4 and four individuals evolved using the L-system as a generative encoding are shown in figure 5. In addition to producing faster creatures, the L-system encoding produced creatures with greater self-similarity and had more parts – the average number of bars in the fittest creatures was 16 using the non-generative encoding and 120 with the L-system encoding.

Other evolutionary runs using the L-system encoding were made with different fitness functions and a higher upper limit on the number of allowed bars. The individuals in figure 6 were evolved against a fitness function that rewarded for having closed loops in the morphology – a is a sequence of rolling rectangles with 169 bars; b is an undulating serpent with 339 bars; c is an asymmetric rolling creature with 306 bars; and d is a four-legged walking creature with 629 bars.

Next we ran experiments combining the neural-network construction language from section 2.2 with morphology construction language of section 2.1 to evolve creatures with neural controllers. To encourage networks with complex dynamics to evolve, individuals were rewarded for the average number of inputs to hidden units and for the range in values of the output units. The evolutionary algorithm was configured to run with a population of 100 individuals for a maximum of 500 generations. Experiments were run using 20 production rules, 3 condition-successor pairs and 2 parameters for each production rule for which approximately half the runs produced interesting creatures Examples of evolved creatures are shown in figure 7: a has 49 neurons and moves by first stretching out its arms, then twisting its body as it closes up to move sideways; b has 41 neurons and moves by falling over each time it wraps up into a circle and unwraps; c has 24 neurons and moves by using the two lower squares as an arm to push it forward; d has 150 neurons and moves by coiling up into a circle to roll; and e has 19 neurons and moves by using its tail to roll it along like a wheel. The network of the creature in figure 7.f is shown in figure 8. In addition to being fairly regular, its linear sequence of outputs also corresponds to the linear sequence of joints in its morphology. It moves by twisting itself to roll sideways.

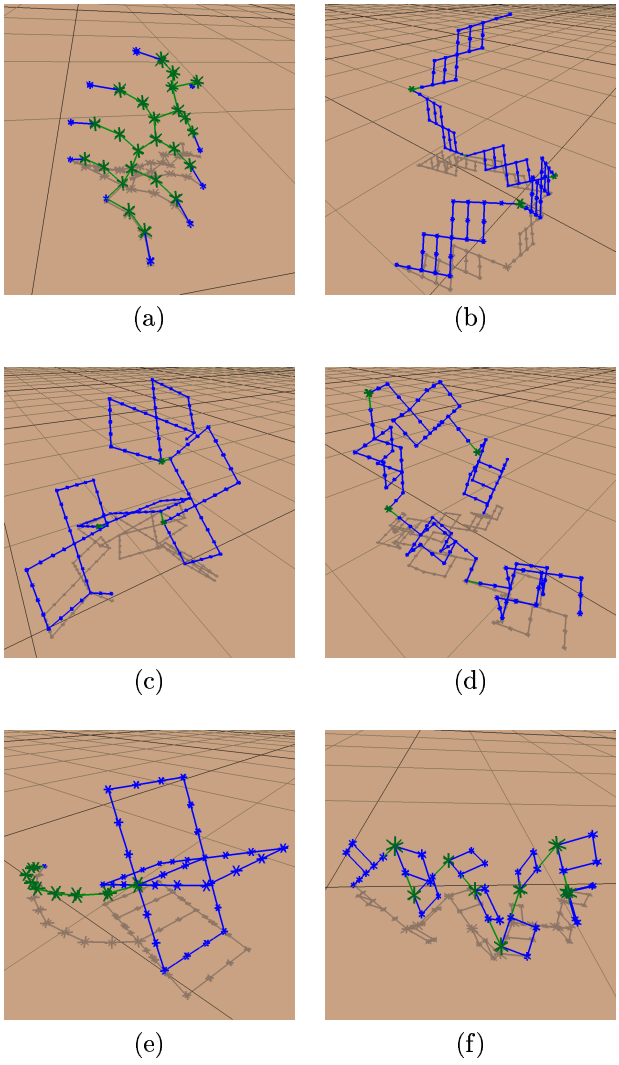


Figure 7: Neural Network Controlled Creatures

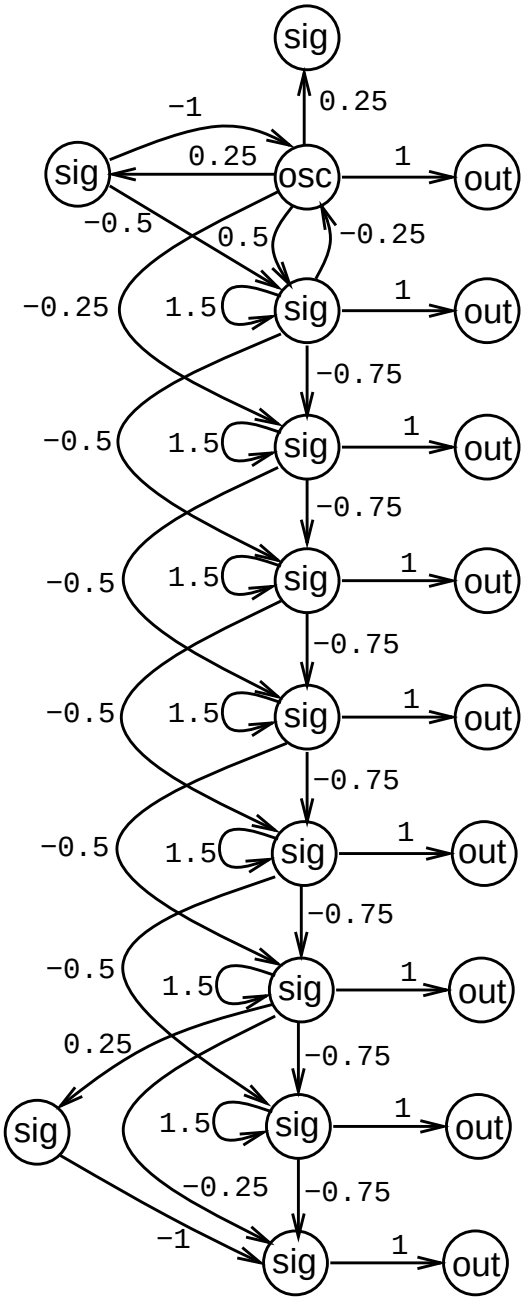


Figure 8: Evolved Neural Network

4 DISCUSSION

As the number of parts in a creature increase, so does the difficulty in hand-designing a controller for it. This system for evolving P0L-systems automatically produced creatures, and their controllers, with hundreds of parts and varying degrees of regularity.

We also used the results of one evolutionary run as the starting population for another run, the creature in figure 6.b is the result of one run seeded with a previously evolved creature. This method of using multiple runs of evolution is one way to use EAs to optimize and explore the design space of the input creature or to create creatures with similar locomotion styles. Alternatively, a simple creature can be designed by hand and the evolutionary system can improve upon it.

For this work we used parametric L-systems as a way to increase complexity over basic L-systems. Other types of L-systems that could have been used are probabilistic L-systems and context sensitive L-systems [Prusinkiewicz & Lindenmayer, 1990]. Probabilistic production rules do not have a condition part, rather they have multiple successors, each with a probability that it will be used to replace the predecessor. While good for generating a variety of similar struc-

tures, this system is non-deterministic – which makes it unsuitable for developing structures that need to be re-created the same each time. Another way in which variation can be applied to an L-system is through the addition of context. Context sensitive L-systems examine the characters to the left and right of the character to be rewritten to determine which successor to replace it with. While this class of L-systems is deterministic, not having parameters results in the inability to take advantage of parametric terminals, such as the oscillators whose parameter specifies the speed of oscillation. Using parameters also has the advantage of allowing one production rule to be used to generate a class of objects. In this way parameters are analogous to the arguments of a function in a computer program and the evolution of an L-system becomes like the evolution of a computer program, as in genetic programming [Koza, 1992].

5 CONCLUSION

An integrated encoding for generatively creating both creature morphology and neural controller was achieved by using evolutionary techniques to evolve P0L-systems. Using this system, the morphologies and controllers were evolved for locomoting creatures. Creatures evolved using the generative encoding moved faster than creatures evolved using the non-generative encoding. In comparison to related work, these evolved creatures consisted of an order of magnitude more parts and had a higher degree of regularity than [Sims, 1994, Komosinski & Rotaru-Varga, 2000, Lipson & Pollack, 2000].

Acknowledgements

This research was supported in part by the Defense Advanced Research Projects Administration (DARPA) Grant, DASG60-99-1-0004. The authors would like to thank the members of the DEMO Lab: A. Bucci, E. DeJong, S. Ficici, P. Funes, S. Levy, H. Lipson, O. Melnik, S. Viswanathan and R. Watson.

References

[Abelson & deSessa, 1982] Abelson, H. & deSessa, A. A. (1982). *Turtle Geometry.* M.I.T. Press.

[Boers & Kuiper, 1992] Boers, Egbert J. W. & Kuiper, Herman (1992). Biological metaphors and the design of modular artificial neural networks. Master's thesis, Leidea University, the Netherlands.

[Gruau, 1994] Gruau, Frédéric (1994). *Neural Network Synthesis Using Cellular Encoding and the Genetic Algorithm.* PhD thesis, Ecole Normale Supérieure de Lyon.

[Hornby et al., 2001] Hornby, G. S., Lipson, H., & Pollack, J. B. (2001). Evolution of generative design systems for modular physical robots. In *Intl. Conf. on Robotics and Automation.*

[Hornby & Pollack, 2001] Hornby, Gregory S. & Pollack, Jordan B. (2001). The advantages of generative grammatical encodings for physical design. In *Congress on Evolutionary Computation.*

[Kitano, 1990] Kitano, H. (1990). Designing neural networks using genetic algorithms with graph generation system. *Complex Systems*, 4:461–476.

[Komosinski & Rotaru-Varga, 2000] Komosinski, M. & Rotaru-Varga, A. (2000). From directed to open-ended evolution in a complex simulation model. In Bedau, McCaskill, Packard, & Rasmussen (Eds.), *Artificial Life 7*, pp. 293–299.

[Koza, 1992] Koza, J. R. (1992). *Genetic Programming: on the programming of computers by means of natural selection.* MIT Press, Cambridge, Mass.

[Lindenmayer, 1968] Lindenmayer, A. (1968). Mathematical models for cellular interaction in development. parts I and II. *Journal of Theoretical Biology*, 18:280–299 and 300–315.

[Lindenmayer, 1974] Lindenmayer, A. (1974). Adding continuous components to L-Systems. In Rozenberg, G. & Salomaa, A. (Eds.), *L Systems*, Lecture Notes in Computer Science 15, pp. 53–68. Springer-Verlag.

[Lipson & Pollack, 2000] Lipson, H. & Pollack, J. B. (2000). Automatic design and manufacture of robotic lifeforms. *Nature*, 406:974–978.

[Luke & Spector, 1996] Luke, Sean & Spector, Lee (1996). Evolving graphs and networks with edge encoding: Preliminary report. In Koza, J. (Ed.), *Late-breaking Papers of Genetic Programming 96*, pp. 117–124. Stanford Bookstore.

[Prusinkiewicz & Lindenmayer, 1990] Prusinkiewicz, P. & Lindenmayer, A. (1990). *The Algorithmic Beauty of Plants.* Springer-Verlag.

[Sims, 1994] Sims, Karl (1994). Evolving Virtual Creatures. In *SIGGRAPH 94 Conference Proceedings*, Annual Conference Series, pp. 15–22.

A Collective Genetic Algorithm

Thomas Miconi

thomas.miconi@lip6.fr
LIP6
8 rue du Capitaine Scott
75015 Paris, France

Abstract

We take a look at the problem of collective evolution, and set the following goal : designing an algorithm that could allow a given population of agents to evolve incrementally, while they are performing their (possibly collaborative) task, with nothing more than a global fitness function to guide this evolution. We propose a simple algorithm that does just that, and apply it to a simple test problem (aggregation among animats controlled by feed-forward neural networks). We then show that under this form, this algorithm can only generate homogeneous systems. Seeing this as an unacceptable limitation, we modify our system in order to allow it to generate heterogeneous populations, in which semi-homogeneous sub-populations (i.e. sub-species) emerge and grow (or regress) naturally until a stable state is reached. We successfully apply this modified algorithm to a very simple toy-problem of simulated chemistry.

1 INTRODUCTION

1.1 PROBLEM STATEMENT

We are looking for an algorithm that could allow us to evolve populations of (possibly heterogeneous) agents, under the following conditions :

1. Global fitness function : we can only evaluate the global performance of the population, and have no way to evaluate directly the individual performance of each agent.

2. On-line evolution : we want this evolution to occur "on-line", incrementally, in an open-ended fashion. The population must evolve continuously over time, while it is performing its task, so that it can adapt to changes in the environment : evolution should not happen in a separate simulation environment, but during the actual "lifetime" of the population.

3. Generality : We want to have a very general method, which could be applied to a wide range of agent types, without any assumption about their nature or inner mechanics.

Requirement 1 stems from the fact that in many collective behaviour problems, when the population must be evaluated after its global result, it may very difficult (or impossible) to reliably share the estimated fitness among the agents. Requirements 2 and 3 are highly desirable if the algorithm is to be used for all kinds of real-world problems.

Note that there is in fact an inclusive-OR relation between requirements 1 and 2 : if we have a reliable individual evaluator, then we should be able to scrap condition 1 and still use our algorithm to reap the benefits of incremental evolution. On the other hand, if we do not need open-ended evolution, we might still use the first part of the algorithm for evaluation purpose and apply it to a classical reproduction scheme.

1.2 RELATED WORK

Evolution is intrinsically a collective process between many intermixing genotypes. Its efficiency as a search technique has been demonstrated analytically, by Holland's Schema Theorem (Holland, 1975; Goldberg, 1989), and empirically, by uncountable applications. However, it is overwhelmingly used as an optimisation technique for individual agents.

There have been numerous attempts at collective evolution, though. The one that inspired our work was (Zaera et al, 1996)'s evolutionary breeding of fish-like

animats to perform various simple collective tasks (dispersion, aggregation, flocking). The system was quite rigidly constrained, however : all animats were strictly identical, and evolution occurred in a classical GA-like process : generate, evaluate, reproduce, then start again, each phase being applied to the whole population. Heterogeneity or open-ended evolution were not considered.

One remarkable achievement in collective evolution is (Luke, 1998)'s generation of soccer players for the Robocup environment (Kitano et al, 1995). The author was able to evolve competitive teams of soccer players through an adapted version of Genetic Programming (Koza, 1990). There also was an attempt at building heterogeneous teams, which failed due to lack of time : because of the enormous search space (even though it had been significantly reduced through massive use of domain-specific knowledge), and because of the delays imposed by the Robocup server software, the GP runs took days to produce successful results. Of course, incremental, real-time evolution was out of the question.

Incrementality and open-endedness in collective evolution is best exemplified by the SAGA paradigm (Harvey, 1992). A succinct definition of it could be : "incremental, open-ended evolution of variable-length genotypes". Lifelong evolution and adaptation to ever-growing complexity in the environment are explicit goals of SAGA. Furthermore, incremental adaptation of a nearly-converged population ("quasi-species") in a growingly complex environment is also described as a potential alternative to the "all-out space search" of traditional GA. However, SAGA was not designed with collective behaviour in mind : agents are supposed to be evaluated one at a time. The notions of "population" and "species" apply only at the genotype level : the actual, phenotypic agents that are produced are essentially autistic.

Perhaps closest to our work is Alastair Channon's Geb world (Channon & Damper, 1998), in which simple animats with a very limited action repertoire (move, kill, reproduce) are supposed to evolve towards ever more complex behaviours. Being highly influenced by SAGA, this work uses nearly-converged species as a way of preserving and expanding "evolutionary emergence", i.e. the capacities and features that have been evolved. However, this work leans much more on artificial life than on artificial intelligence, in that there is no such thing as a fitness function : animats are selected solely on their aptitude to survive and reproduce. Trying to constrain this system in order to make it perform even a very simple task is (as we have painfully learned) extremely difficult.

To the best of our knowledge, no single work has ever been proposed to meet all the requirements expressed in our problem statement, even less so with heterogeneous populations.

2 THE BASIC COLLECTIVE GENETIC ALGORITHM

2.1 EVALUATION

Any genetic algorithm requires an evaluation phase. We stated that our only evaluation method is a global fitness function that assesses the performance of the whole population. However, for GA-like selection, we must give a "mark" to each agent, or at least we must be able to compare any two agents within the population.

To do this, we use a very simple rating scheme : *the score of an agent is determined by the behaviour of the system when that agent is not present*. In other words, to evaluate an agent's usefulness, we temporarily "remove" it from the system, and we evaluate the remaining population. If that agent had a positive influence on the performance of the population, the global fitness will naturally decrease when we remove that agent. Similarly, if that agent had a negative impact on the overall performance, the global fitness will increase after it is removed. To put it briefly : the higher the fitness of the remaining population, the lower the usefulness of the removed agent.

In other words, to evaluate an agent we:

1. "Remove" that agent from the population (that is, we prevent it from exerting any influence on other agents, or on the global score of the population).
2. Evaluate the score of the remaining population.
3. Give that score to the currently evaluated agent, as a *negative score* (i.e. the lower, the better).

This evaluation system is pretty much an independent sub-algorithm in its own right, which can be used in any situation where one needs to evaluate the usefulness of each individual agent through global evaluation.

One advantage of it is that it does not only consider the immediate "productivity" of that particular agent : it takes into account the indirect effects this agent may cause by its mere existence. Some agents may not be directly productive, but their activity may indirectly contribute to an increased overall productivity ("helper" agents). On the other hand, some agents may be have a neutral behaviour by themselves, and yet be very damaging for the population as a whole (e.g. agents that "stand in the way" of others, preventing them from performing their task). An individual evaluator cannot see this, but our evaluation system can detect it, and reward (or punish) these agents accordingly. Those features make this method remarkably suitable for the generation of collaborative systems.

2.2 REPRODUCTION

Having a tool to measure each agent's impact on the system, we can use it to drive the evolutionary process through reproduction and crossover. However, it should be clear that this "individual fitness" cannot be used for a

classical GA algorithm, where evaluation and reproduction are applied separately to the whole population at every generation. Our evaluation of each agent's score only holds within a given context, and that context is the rest of the population; if we change other agents' behaviour, this score will become obsolete and will have to be re-calculated. Evaluation and reproduction can therefore not be performed separately : if the system is to be modified it must be progressively, step by step, agent after agent.

Our reproduction scheme is the following: At every round, we :

1- Choose two agents randomly within the population

2- Evaluate each one in turn by the method described above.

3- Create an offspring through one-point crossover between the two chosen agents (this actually creates two children; we select one of them randomly).

4- Replace the agent with the highest negative score (i.e. the less useful agent) by this offspring

This cycle *is* the collective genetic algorithm proper. It may be seen as an extension of the tournament selection method for classical GAs. There can be several variations on it, some of which may bring very interesting results. We will examine one of them in more detail in the following sections.

We can see that this algorithm is fundamentally incremental. There is no such thing as a "generation" : the population changes gradually over time, while being constantly evaluated - that is, while constantly performing its task. This fits remarkably well in an open-ended evolution scheme.

3 APPLICATION

Note : All the software described in this paper is available at http://miriad.lip6.fr/~miconi .

Let us see how this algorithm can be applied to a simple evolutionary problem : the evolution of aggregation among a population of animats.

3.1 THE PROBLEM

This experiment takes place in a 2-D toroidal world, where simple animats roam freely. These animats are controlled by classical three-layer feed-forward neural networks. The weights of these networks are real numbers in the [-1; 1] domain.

The networks have four inputs : North, East, South, West. Each of these inputs is stimulated according to the number of other animats in the corresponding direction. The degree of stimulation caused by each animat is proportional to its distance:

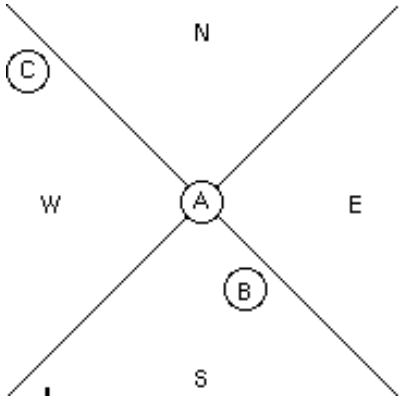


Figure 1: The 4 input zones for animat A. Animat B will cause a strong stimulation on input South, while animat C will cause a lower stimulation on input West

The middle layer has twelve neurons. They perform a summation of their inputs, and apply a sigmoid function to it, before sending the result to the output neurons. There are only two output neurons : one gives the vertical speed of the animats, and the other one gives the horizontal speed. Of course the output values (contrarily to input values) can be negative.

This aggregation problem was successfully addressed by (Zaera et al., 1996) in a more classical way, at the population level : all animats in one given simulation round had the same genotype, and thus the same network. Those populations were evaluated, and the global score was attributed to this genotype. Other genotypes were evaluated in the same fashion, and a traditional GA was applied to these genotypes.

In our experiment, the global evaluator is simply the total sum of the distances between each animat and all the other ones. Evolution occurs in the way described above : an individual is chosen randomly and "removed" from the population (which actually means that other animats cannot see it, and it is not taken into account for the global evaluation). The population is evaluated globally, by calculating the distances between every pair of animats (except for the currently evaluated animat), during 100 timesteps. This score (which is in fact a negative score, as it gives a measure of the global fitness when this animat is not present) is attributed to this animat. The process is repeated with another randomly-selected animat. Then the scores are compared, and the animat with the higher score (i.e. the less useful one) is replaced by the offspring of the two evaluated animats. The offspring is created through one-point crossover, "spiced" with a mutation rate

varying between 0,5% and 2% for each run. Mutation is performed by increasing or decreasing a weight by some random amount between 0 and 0,5 (all weights being comprised in the [-1; 1] range). We use a very small population (10 animats).

3.2 RESULTS

At the beginning of the experiment, animats wander randomly through the toroidal space.

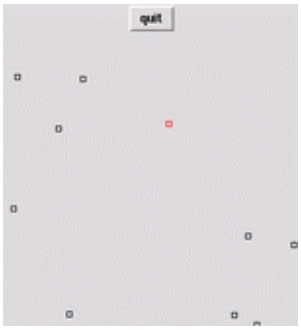


Figure 2: Initial setting

But after a few thousands of rounds (remember there is no such thing as a "generation" in this algorithm : a round means two evaluations and one reproduction), aggregation occurs.

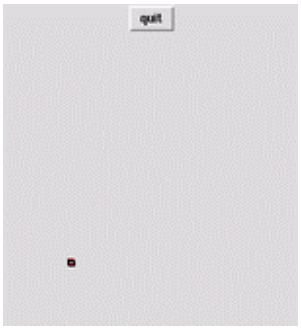


Fig. 3: Animats move in tight flock

All runs led to successful aggregation. However, significant differences have been found in the time it takes to achieve this aggregation, and in the stability of this aggregation afterwards. The mutation rate seems to have an important role, which is not very surprising given the small size of the population. For example, the following run was performed with a mutation rate of 0,5%:

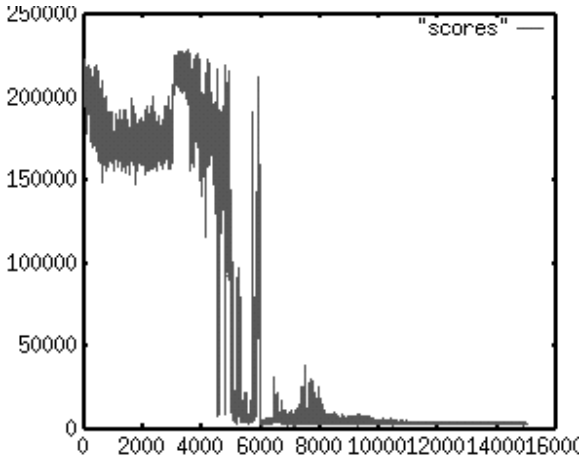


Fig. 4: Fitness curve, 0,5% mutation rate : successful aggregation. The x-axis indicates the number of evaluations.

On the other hand, the following run (performed with a mutation rate of 0,66%) led to "absolute aggregation" : all the animats eventually settled on a very small zone, and stopped moving. This result represents the optimal solution for this problem :

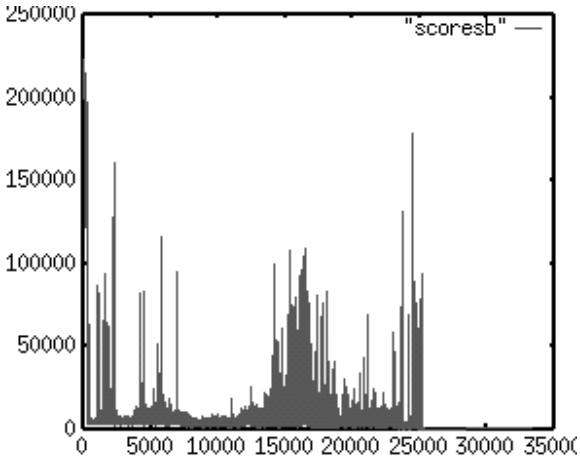


Fig. 5: 0,666% mutation rate; after a semi-stable aggregation phase, the fitness curve becomes confounded with the x-axis : the animats are totally aggregated and motionless.

3.3 TOWARDS HETEROGENEOUS SYSTEMS

We have shown that our algorithm can solve a simple

problem of collective evolution quite efficiently. However, under its present form, it exhibits one specific feature that is quite disturbing in that it imposes a strong restriction on the systems that can be produced.

Evolution occurs through massive cross-over (and marginally through mutation) between different genotypes. This means that after some period of time, there will be a general trend of convergence between all the genotypes. They won't be perfectly identical, and will keep changing, but at equilibrium state they will essentially share the same properties. This effect is regarded as a harmful in traditional GAs, and maintaining the diversity of the population has been a primary concern since the beginning of evolutionary computation (see Schraudolph & Belew, 1992). However, in our case, convergence is not *necessarily* a bad thing : it means that the population has successfully adapted to its environment. This is one of the traits this algorithm shares with SAGA (see above) : convergence is not an undesirable effect that should be prevented, it is the natural outcome of evolution and adaptation.

However, this imposes a strong limit to our system as it stands : it can only generate functionally homogeneous systems, where all animats converge to share similar traits. It can therefore not handle situations in which several different types of agents in the population are required : stable heterogeneous system are intrinsically beyond the reach of this algorithm in its basic form.

We want to go beyond that. We want our algorithm to be able to generate stable heterogeneous systems, where different species emerge, coexist and collaborate. By "emergence", we mean that we expect to find simultaneously the characteristics of those species, and the proportion of each species within the population. Both tasks should be done in parallel, with no other guidance than global evaluation.

We found that this could be achieved very simply, through a modification of the reproduction process.

4 HETEROGENEOUS SYSTEMS

4.1 ENHANCEMENT OF THE BASIC ALGORITHM

Let us recapitulate the main steps of the algorithm under its present form :

1- Choose an individual randomly in the population.

2- Remove it from the population and evaluate the global behaviour of the population without this agent. Store the result as the agent's "negative score". Put the agent back into the population.

3- Choose another agent in the population.

4- Same as 2, with this newly selected agent.

5- Replace the agent with the highest "negative score" by the offspring of the two chosen agents.

We will not change this basic organisation. However, we will add a restriction on the reproduction mechanism. This restriction will not be enforced in step 5 (crossing-over), but in step 3, that is, the choice of the second mating agent.

To do this, we specify that agents must be numbered, that is, each agent must be given a unique index number at initialisation time. The attribution is totally arbitrary. For example, if the agents are to be stored in an array, each agent could be given its rank in the array as an index. The only requirement is that this index must not change throughout the whole duration of the experiment.

These indices are circular, that is, if there are N agents, the successor of agent N is agent 1. This allows us to define a distance over the agent space : the distance between two agents is the absolute value of the difference of their indices. The distance between agents 4 and 6 is 2, the distance between agents 9 and 6 is 3, the distance between agents N-1 and 2 is 3, etc.

This being done, we are ready to enforce the following constraint on the reproduction process : *reproduction can only occur between agents within a given distance*. That is, agent M can only reproduce with agents within the [M-r; M+r] range, where r is the "reproduction radius", the maximal radius within which reproduction is allowed.

In our algorithm, this will be enforced as follows : at every round, the first agent to be evaluated is still chosen randomly, but the second agent is chosen within a limited range around that first agent. For example, if the reproduction radius r is 5 and the first selected agent is agent 2, then the second agent will be chosen within the [N-3; 7] range.

What are the consequences of this restriction ? To answer this question, we must study what happens when an agent is evaluated. If the evaluation yields a bad result, this agent is eliminated and replaced with a new one (although this new agent still has some genetic material borrowed from the old one through crossing-over). If the evaluation result is good, the agent will remain unchanged, and propagate its genes around it through its offspring.

But since reproduction can only take place within a limited range, this means that this propagation will first happen within the immediate vicinity of that agent. In other words, if a very good agent appears somewhere, it will quickly propagate its genes to its neighbours, leading to a concentration of genes around it. Those neighbours, in case they gain an evolutionary advantage from those acquired genes, will in turn pass them to their own neighbours. This means that outstanding agents will initiate a *gene percolation* process, leading to the emergence of groups of agents sharing similar genes and similar traits - i.e. the emergence of species.

This gene percolation process will go on until two species

of agents (let us call them A and B) come to meet with each other; that is, until a boundary emerges, with A-agents on one side, B-agents on the other side, and a mixture of both in the middle. At this point, suppose one agent of type A is first selected for evaluation, followed by an agent of type B. These evaluations will tell us whether the system needs more agents of type A, or more agents of type B, and gene propagation will occur accordingly : the most useful agent will be preserved, while the other one will be replaced by a mixture of both.

In other words, useful genes will first be concentrated into species, then will percolate from neighbour to neighbour until they "collide" with another species. Then, the boundary will fluctuate, and the "territory" of each species will be determined by its relative usefulness in comparison to others', until an equilibrium is reached. We have achieved our goal : evolving heterogeneous system through global evaluation alone.

4.2 APPLICATION

We will apply this enhanced version of the algorithm to a very simple chemistry-like toy-problem.

We have a pool of N cells, each of which produces a particular type of molecule according to its genotype. All cells have a 4-genes genotype, where each gene can have the value (allele) 0, 1 or 2.

We know that three types of cells, Pa, Pb and Pc (genotypes : 0000, 1111 and 2222), produce respectively the molecules A, B and C. We state that these molecules react together in the following way :

$$A + B = 2 * P + Z$$
$$P + Z = Nil$$
$$C + Z = Nil$$

In other words : the reaction of A and B produces two molecules of product P, plus one molecule of byproduct Z. Z is a corrosive, and can therefore react with the P molecules and annihilate them. But C can reduce this effect by annihilating the Z molecules.

The problem is the following : generating a pool of cells that maximizes the quantity of product P. We can see that the global fitness function may be expressed as follow :

$$f = Np - \max((Nz - Nc), 0)$$
$$= 2*\min(Na, Nb) - \max((\min(Na, Nb), 0)$$

where Nx is the number of molecules of product X in the current output. Since P- and Z-molecules can only be produced through the reaction of a A-B pair, the number of P-/Z-molecules that can be produced is proportional to min(Na, Nb), i.e. the maximum number of A, B pairs available.

We use the same process as before : we evaluate two agents by "removing" them (i.e. not taking them into account in the evaluation of the production), and replace the one with the highest "negative score" by an offspring created through one-point crossover. There is no mutation.

At the beginning of the experiment, all cells have a random genotype.

1001
2021
1220
1020
0122
...

But after a few seconds, the population converges towards a less chaotic state :

2222
2222
2122
2222
2111
1111
1111
...

Analysis of the resulting population shows the proportions of each genotype fluctuate within a short range around the following equilibrium :

$$Na \sim= Nb \sim= Nc \sim= 33 \sim= N/3$$

The fact that this solution is optimal can be quickly verified by the reader for a small value of N (eg if N=12, it is easy to see that the global fitness function reaches a maximum for Na = Nb = Nc = 4).

4.3 ADAPTATION

The system has found an optimal solution to a simple problem involving the generation of a heterogeneous system. This was our main requirement. However, we would want to check whether the system meets another important requirement that was specified at the beginning of this discussion : adaptability. If the conditions of the environment change brutally, will the system be able to

adapt its solution accordingly ?

To assess the adaptability of the system, we modify our experimental settings in the following way : we specify that after T timesteps (T being large enough to reach equilibrium state), the reaction between A and B becomes:

$$A + B = P + 2 * Z$$

That is, the reaction of A and B yields only one molecule of product P and two molecules of by-product Z. All other equations are unchanged. The global fitness function thus becomes :

$$f = Np - \max((Nz - Nc), 0)$$
$$= \min(Na, Nb) - \max((2*\min(Na, Nb) - Nc), 0)$$

which means a much tougher environment, since the reaction yields twice as much corrosive Z-molecules as desirable P-molecules.

This drastic altering of the pseudo-chemical rules does have an impact on the system. As soon as this modification in the rules occur, the proportions of each genotype change and quickly converge towards a new equilibrium :

$$Na \sim= Nb \sim= 25 = N/4$$
$$Nc \sim= 50 = N/2$$

Once again, it is easy to verify the optimality of this solution for a small value of N : if N = 12, the fitness function clearly reaches a maximum with Na = Nb = 3 and Nc = 6. The optimum that is reached under these harder conditions is of course much lower than the previous one. Those results are clearly indicated in the fitness curve :

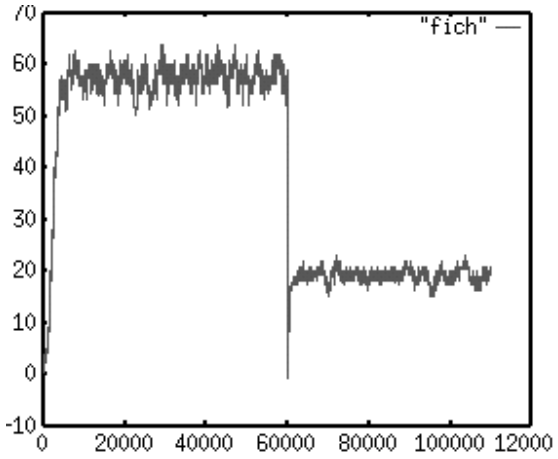


Fig. 6: Fitness curve of the population : the sharp drop corresponds to the change in the pseudo-chemical rules (Note : despite the impressive figures on the x-axis, the whole experiment described in this curve took about ten seconds on a PII-366MHz laptop computer)

5 CONCLUSIONS

We have designed an algorithm through which we can apply collective incremental evolution to a population of (possibly heterogeneous) agents, by using a global, explicit fitness function alone. It is abstract enough to be used in a broad variety of situations, with virtually any kind of agents. To our knowledge, no other technique has been proposed to date, which fulfils all these requirements.

However, while this algorithm is able to solve the very simple problems that we have used as a test bed, further (and tougher) testing is obviously needed. Our current goal is to design larger experiments in order to assess the behaviour of our system when confronted to harder problems. We are currently working on collaborative foraging, with both homogeneous and heterogeneous populations. In the longer run, we plan to apply this system to physical agents, e.g. collectivities of robots such as those described in (Drogoul & Picault, 1999).

Besides testing, further work also includes experimenting with variations on the basic evolutionary process : “gross evaluation” (in which several agents are removed from the population at each evaluation) might be a way to speed-up the initial phase of evolution. Different reproduction schemes could be used. The sensitivity of the algorithm in regard to internal parameters is also being studied : population size, for example, may represent a trade-off between diversity and efficiency, while evaluation time is clearly a trade-off between efficiency and accuracy. These investigations could lead to a significant improvement of our system.

References:

(Channon & Damper, 1998) "Perpetuating Evolutionary Emergence", A.D. Channon and R.I. Damper. *Proceedings of the Fifth International Conference on Simulation of Adaptive Behavior (SAB '98)*.

(Drogoul & Picault, 1999) "MICRobES: Vers des Collectivités de Robots Socialement Situés", A. Drogoul and S. Picault. *Actes des 7èmes Journées Francophones Intelligence Artificielle Distribuée et Systèmes Multi-Agents (JFIADSMA'99).*

(Goldberg, 1989) "Genetic Algorithms in Search, Optimization and Machine Learning", D. E. Goldberg. *Addison-Wesley, Reading, Massachussets, USA*.

(Harvey, 1992) "Species Adaptation Genetic Algorithms: A Basis for a Continuing SAGA", I. Harvey. *Proceedings of the First European Conference on Artificial Life*.

(Holland, 1975) "Adaptation in Natural and Artificial Systems", J. Holland. *University of Michigan Press, Ann Arbor, USA*.

(Kitano et al, 1995) "Robocup: The Robot World Cup Initiative", H. Kitano, M. Asada, Y. Kuniyoshi, I. Noda and E. Osawa. *Proceedings of the International Joint Conference on Artifical Intelligence (IJCAI-95) Workshop on Entertainment and AI/Alife.*

(Koza, 1990) "Genetic Programming: A paradigm for genetically breeding populations of computer programs to solve problems", J.R. Koza. *Technical Report STAN-CS-90-1314, Dept. of Computer Science, Stanford University*.

(Luke, 1998) "Genetic Programming Produced Competitive Soccer Softbot Teams for RoboCup97", S. Luke. *Proceedings of the Third Annual Genetic Programming Conference (GP '98).*

(Schraudolph & Belew, 1992) "Dynamic Parameter Encoding for Genetic Algorithms", N.N. Schraudolph and R.K. Belew. *Machine Learning Journal, vol. 9, n. 1, 9-22*.

(Zaera et al, 1996) "(Not) Evolving Collective Behaviours in Synthetic Fish", N. Zaera, D. Cliff and J. Bruten. *Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior (SAB '96)*.

Understanding the Role of Learning in the Evolution of Busy Beavers: a Comparison Between the Baldwin Effect and a Lamarckian Strategy

Francisco B. Pereira[1,2], Ernesto Costa[2]

[1]Instituto Superior de Engenharia de Coimbra, Quinta da Nora, 3030 Coimbra, Portugal
[2]Centro de Informática e Sistemas da Universidade de Coimbra, Polo II, 3030 Coimbra, Portugal
{xico, ernesto}@dei.uc.pt
phone: +351 239790000

Abstract

In this paper we study how individual learning interacts with an evolutionary algorithm in its search for good solutions to the Busy Beaver problem. Two learning strategies, the Baldwin Effect and Lamarckian learning, are compared with an extensive set of experiments. Results show that the Baldwin Effect is less sensitive to specific issues concerning the definition of the learning model and it is more effective in adjusting its learning power to maximise the search performance of the evolutionary algorithm. Some insight about the specific role that evolution and learning play during search is also presented.

1 INTRODUCTION

Evolution and learning are the two major forces that promote the adaptation of individuals to the environment. Evolution, operating at the population level, includes all mechanisms of genetic changes that occur in organisms over generations. Learning operates at a different time scale. It gives to each individual the ability to modify its phenotype during its life in order to increase its adaptation to the environment and, hence, its chance to be selected for reproduction. In standard evolutionary computation (EC) optimisation, learning has usually been implemented as local search algorithms. These methods iteratively test several alternatives in the neighbourhood of the learning individual trying to discover better solutions. At the end of the learning process, the quality of an individual will be, not only the measure of its initial fitness, but also of its ability to improve, which leads to a better understanding of the fitness landscape. In our research we are interested in studying how learning and evolution may be combined in computer simulations.

In this paper we use the Busy Beaver (BB) problem as the testbed to study the above-mentioned interactions. In 1962, Tibor Rado proposed this problem in the context of the existence of non-computable functions [13]. It can be defined as follows: suppose a Turing Machine (TM) with a two-way infinite tape and a tape alphabet={blank, **1**}. The question Rado asked was: what is the maximum number of **1**'s that can be written by a N-state halting TM when started on a blank tape? This number, which is a function of the number of states, is denoted by $\sum(N)$. A TM that produces $\sum(N)$ non-blanks cells is called a Busy Beaver. The BB is considered one of the most interesting theoretical problems and, since its proposal, has attracted the attention of many researchers. Some values for $\sum(N)$ and the corresponding TMs are known today for small values of N. As the number of states increases, the problem becomes harder and, for $N \geq 5$, there are several candidates that set lower bounds on the value of $\sum(N)$. To prove that a particular candidate is the N-state BB we must perform an exhaustive search over the space of all N-state TMs and verify that no other machine produces a higher number of ones. This is extremely complex due to the halting problem. In the original setting, the problem was defined for 5-tuple TMs. One of the main variants consists in considering 4-tuple TMs. In the next section we present a formal definition of the BB problem for both variants.

The search space of the BB problem possesses several characteristics, such as its dimension and its complexity, that make it extremely appealing to the EC field. We performed some empirical analysis on the topology of the landscape and verified that, in different areas of the search space, there are small groups of neighbour valid solutions to the BB problem. The size of these groups and the quality of the TMs that compose them varies but, nevertheless, they tend to be surrounded by large low fitness areas composed by invalid solutions. The combination of these factors makes the space highly irregular and very prone to premature convergence. The first attempt to apply EC techniques to the BB problem was reported by Terry Jones [6], who used a genetic algorithm to search for specific instances of the 5-tuple BB. In 1999, our research group obtained a remarkable success in our first effort to apply EC algorithms to the 4-tuple variant of the problem [8]. Several new lower bounds were set, leading to a large increase in the productivity of 6 and 7-state 4-tuple TMs.

Following our research interests, in a previous work [12] we studied the influence that two different learning models had in the performance of an evolutionary algorithm when seeking for solutions to the 4-tuple BB.

The difference between the two models concerned the number of modifications that they were allowed to perform in the structure of the TM in each learning step. We presented results of several experiments performed within a Lamarckian framework and showed that, given the complexity of the search landscape, a learning procedure that is able to perform several modifications in the structure of the individual in each learning step is most beneficial.

In this paper we study the behaviour of the two learning models with a different strategy, known as the Baldwin Effect, and compare the results with the ones previously obtained with the Lamarckian framework. In recent years both strategies have been usefully applied in several experiments and its impact has been studied in various domains [1], [2], [3], [5], [14], [16]. Here we present a detailed set of tests performed in a very complex search landscape. Our goal is to try to understand the role that evolution and learning play during the search process. We also would like to establish some concrete rules to determine, under which conditions, does learning really provide help to evolution in its task of sampling the space.

The structure of the paper is the following: in the next section we present a formal definition of the BB problem. In section 3 we describe our evolutionary model, including the learning procedures used. Section 4 comprises some experimental details about the simulation. In section 5 we present results of the experiments performed and analyse them. Finally, in section 6, we review the main conclusions of this work.

2 THE BUSY BEAVER PROBLEM

A deterministic TM can be specified by a sextuple $(Q,\Pi,\Gamma,\delta,s,f)$, where: Q is a finite set of states, Π is an alphabet of input symbols, Γ is an alphabet of tape symbols, δ is the transition function, $s \in Q$ is the start state and $f \in Q$ is the final state [17]. The transition function can assume several forms. The most usual one is:

$$\delta: Q\times\Gamma \rightarrow Q\times\Gamma\times\{L,R\}$$

where L denotes move the head left and R move right. Machines with a transition function with this format are called 5-tuple TMs. A common variation consists in considering a transition function of the form:

$$\delta: Q\times\Gamma \rightarrow Q\times\{\Gamma\cup\{L,R\}\}$$

Machines of this type are known as 4-tuple TMs. When performing a transition, a 5-tuple TM will write a symbol on the tape, move the head left or right and enter a new state. A 4-tuple TM either writes a new symbol on the tape or moves its head, before entering the new state.

The original definition of the BB [13] considered deterministic 5-tuple TMs with N+1 states (N states and an anonymous halt state). The tape alphabet has two symbols, Γ={blank, **1**}, and the input alphabet has one, Π={**1**}. The productivity of a TM is defined as the number of **1**'s present, on the initially blank tape, when the machine halts. Machines that do not halt have productivity zero. $\sum(N)$ is defined as the maximum productivity that can be achieved by a N-state TM. This TM is called a Busy Beaver.

In the 4-tuple variant, productivity is usually defined as the length of the sequence of **1**'s produced by the TM when started on a blank tape, and halting when scanning the leftmost one of the sequence, with the rest of the tape blank. Machines that do not halt, or, that halt on another configuration, have productivity zero [4]. Thus, the machine must halt when reading a **1**, this **1** must be the leftmost of a string of **1**'s and, with the exception of this string, the tape must be blank. In our research we focus on the 4-tuple variant.

δ	By blank		By one	
Q	New State	Action	New State	Action
1	5	**1**	*f*	**1**
2	4	0	7	R
3	4	**1**	6	L
4	3	**1**	4	R
5	2	R	3	L
6	1	0	7	L
7	1	R	5	0

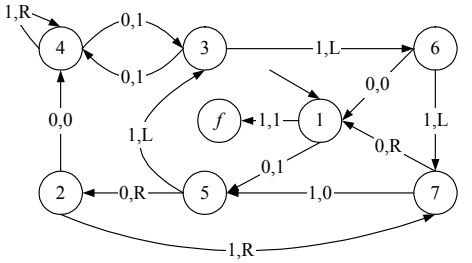
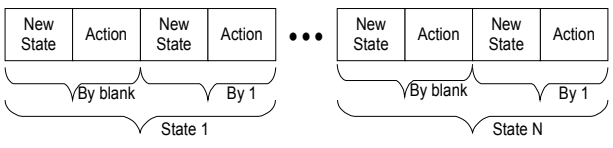


Figure 1: A seven-state 4-tuple TM and its corresponding transition table. The blank symbol is represented by 0.

3 EXPERIMENTAL MODEL

3.1 REPRESENTATION

In the experiments reported in this paper we are searching for good candidates for the 4-tuple BB(7). Without loss of generality we consider Q={1,2,3,4,5,6,7,*f*}, set 1 as the initial state and *f* as the final state. Since Γ={blank, **1**}, the essential information needed to represent a potential solution is reduced to the state transition table. Figure 1 shows a 4-tuple TM with 7 states (plus the halting state *f*) and its state transition table. To codify the information contained in the table we use an integer string with 28 genes (4 genes per state) with the following format:



3.2 SIMULATION AND EVALUATION

To evaluate an individual we simply decode the information from the chromosome and then simulate the resulting TM. Due to the halting problem we must establish a limit for the maximum number of transitions (MaxT). Machines that don't halt before this limit are considered non-halting TMs. To assign fitness we consider the following factors in decreasing order of importance:

- Halting before reaching the predefined limit for the number of transitions;
- Accordance to the 4-tuple rules [4];
- Productivity;
- Number of used transitions;
- Number of steps made before halting.

We consider all these factors to assign fitness because we intend to explore differences between "bad" individuals. With this fitness function a TM that never leaves state 1 is considered worse than another one that goes through 3 or 4 states, even if both are non-halting machines and have the same productivity. In some preliminary experiments this approach proved to be more effective than using productivity alone as fitness.

3.3 LEARNING MODELS

After evaluation an individual might be selected for learning. In this research we present results from experiments performed with two local search procedures.

Random Local Search (RLS).
Given a current TM (machine that was built with the information encoded in the chromosome of the individual selected for learning) perform the following actions:

1. Select one transition T used in the simulation of the current TM.
2. Randomly modify the action performed by transition T[1].
3. Evaluate the resulting TM.
4. If the fitness of the resulting TM is equal or higher than the fitness of the current TM, then the resulting TM becomes the current one.
5. If the maximum number of steps has been equalled stop learning. Otherwise go to 1.

In each learning step RLS performs one modification in the structure of the TM, accepting it if it does not lead to a decrease in the fitness of the individual. Changes are limited to actions performed by transitions. To ensure that this restriction is not biasing results, we performed some additional experiments enabling RLS to change either actions or new states and verified that there is no significant variation in the outcomes.

The most important difference between RLS and Multi Step Learning (MSL) is that, with this second model, in each learning step an individual performs 2 or 3 changes in its structure. An algorithmic description of MSL follows:

Multi Step Learning (MSL).
Given a current TM (machine that was built with the information encoded in the chromosome of the individual selected for learning) perform the following actions:

1. Select one transition T1 used in the simulation of the current TM and that does not lead to the final state.
2. Randomly modify the action performed by transition T1.
3. With a probability of 0.5 randomly modify the state to where transition T1 leads. The final state is not considered as a possibility when selecting the new destiny.
4. Let S be the state to where T1 leads. Select, with equal probability, one transition T2 from state S.
5. Randomly modify the action performed by transition T2.
6. Evaluate the resulting TM.
7. If the fitness of the resulting TM is equal or higher than the fitness of the current TM, then the resulting TM becomes the current one.
8. If the maximum number of steps has been equalled stop learning. Otherwise go to 1.

Modifications in the structure of the TM are done in components that are directly connected, starting with the action of one transition, then the destiny state of the transition (it changes with 0.5 probability) and, finally, the action of one of the transitions from this state. With MSL, an individual has the possibility to jump to a point in space that is not so close to its current position.

3.4 LEARNING STRATEGIES

There are two different ways to combine evolution and learning in adaptive systems, inspired in two biological theories. We will test both learning strategies with each one of the models proposed.

3.4.1 Lamarckian Learning

Lamarckian theory of evolution claims that phenotypic characteristics acquired by individuals during their lifetime are somehow encoded in their genes and directly inherited by their descendants. Even though this theory proved to be wrong in biological systems, the idea has been usefully applied in several experiments in the EC field. In our experiments, when using a Lamarckian strategy, at the end of the learning period, all changes induced in the current TM are coded back to the genotype of the learning individual.

3.4.2 The Baldwin Effect

Baldwin proposed a non-Lamarckian view of evolution, where acquired characteristics could only be indirectly inherited. This process, known as the Baldwin Effect,

[1] Possible actions for one transition: write blank, write **1**, move left or move right.

occurs in two steps [15]: first, phenotypic plasticity allows an individual to adapt to successful changes. These changes lead to an increase in the fitness of the individual and so it will tend to proliferate in the population. Given sufficient time, a characteristic that was once learned may become innate. To ensure that this last step, known as genetic assimilation, occurs we use equation 1 to assign fitness to an individual *x* that just finished its learning period.

$$F(x) = F_L(x) * (1 - \alpha * \frac{Improv}{Max\_Steps}) \quad \textit{(equation 1)}$$

Where:

- *F(x)*: Final fitness assigned to individual *x*;
- $F_L(x)$*:* Raw fitness of the individual after learning (fitness of the current TM at the end of the learning period);
- *Max_Steps*: Maximum number of steps of the learning process;
- *Improv*: Number of steps of the learning period that conducted to an increase in the fitness of individual *x*;
- α: Constant representing the cost of learning. In the experiments presented in this paper it has value 0.1.

A detailed analysis of the conditions that are required for genetic assimilation to occur are beyond the scope of this work (consult [10] for such a discussion). In this paper it suffices to say that, if we do not perform the modification codified by equation 1, there will be no difference between the fitness of individuals who innately contain the genetic information about areas of increased fitness and individuals that are able to learn this trait during lifetime. This flattening of the space around optima removes the evolutionary pressure of the simulation. In experiments with a Lamarckian strategy this modification is not required because all changes induced by learning are directly encoded in the genotype.

4 EXPERIMENTAL SETTINGS

The experiments presented concern the search for the 4-tuple BB(7). The settings of the evolutionary algorithm are the following: Number of evaluations: 200,000,000; Population Size: 500; Elitist Strategy; Tournament Selection with tourney size 5; Single Point Mutation; Mutation rate: 0.025; Graph Based Crossover; Maximum graph crossover size: 4; Crossover rate: 0.7; MaxT (Maximum number of transitions): 100,000.

Graph based crossover was presented in [11]. It was designed to work with individuals with a graph-like structure and to manipulate them in a way that is consistent with its representation. The main idea of this operator is the exchange of sub-graphs between individuals. Maximum graph crossover size defines the number of states belonging to each sub-graph. Results presented confirmed that, in this domain, it clearly outperforms classical crossover operators. In experiments with learning we make use of another parameter, the Learning Rate (LR), which is defined as the probability of an individual being subject to learning. With this parameter it is possible to restrict the number of individuals that learn in each generation. We present results from experiments with 5 different values: {0.1, 0.25, 0.5, 0.75, 1.0}. During learning the number of steps performed by an individual is set to 10 and remains fixed for all experiments. Each step counts as one evaluation. The initial population is randomly generated and for every set of parameters we performed 30 runs with the same initial conditions and different random seeds. Even though values for different settings were set heuristically we performed some tests with other values and verified that, within a moderate range, there was no significant difference in the outcomes.

5 RESULTS

In this section we present results from two distinct sets of experiments: the first one includes experiments performed with Lamarckian learning and the second one experiments performed with the Baldwin Effect. Each set comprises 10 different experiments: 5 using the RLS learning model and another 5 using the MSL learning model. The only difference between experiments using the same learning model is the LR value. Additionally, we present results from one experiment where individuals do not learn during evolution (*NoLearn*) to serve as a comparison measure.

In tables 1 and 2 we present, for all different experiments, the productivity of the best individual of the final generation, in each one of the 30 runs. Table 1 presents results concerning experiments with the Baldwin Effect, whilst table 2 has results from experiments performed with Lamarckian learning. In both tables the column labelled *NoLearn* presents results from the experiment where individuals did not learn. Before the application of EC techniques to the 4-tuple BB(7), the productivity of the best known candidate was 37 [7]. We adopt this pre-EC record as the threshold of minimum quality and focus our attention in runs that were able to find TMs with higher productivity. A brief perusal of the results shows that EC techniques enabled the regular discovery of good solutions. With all settings it was possible to find TMs with productivity > 37. The results obtained also suggest that the solution space for the 4-tuple BB(7) is not continuous in terms of productivity. We were able to find a few TMs with productivity around 100 and then we found several TMs with productivities that range between 161 and 164. This discontinuity is also visible in the 5-tuple variant of the problem [9].

Focusing our analysis in the last two rows of each one of the two tables it is possible to see that, in a diversity of situations, both learning strategies were able to help evolution. There are, however, important distinctions that require a detailed analysis. With Lamarckian learning, only experiments with the MSL model were able to outperform the *NoLearn* algorithm in a consistent way. A standard evolutionary approach found TMs with productivity > 37 in 10% of the runs (3 out of 30 runs).

With RLS only one LR value (LR=0.25) was able to marginally obtain a better result. At the contrary, results obtained by some experiments with the MSL model are clearly better than those obtained by RLS model and the standard evolutionary algorithm. With LR=0.1, 20% of the runs were able to find TMs with productivity > 37 (6 out of 30 runs). This percentage is even slightly higher when LR=0.25. Then, as the LR value increases, the advantage of Lamarckian learning experiments decreases. It is not surprising that the performance decreases as the value of LR increases. Since it re-encodes all changes back to the genotype, Lamarckian learning is a very strong mechanism and it pushes the search very fast to a local optimum. Given the complex landscape that we are dealing with, this effect is even magnified. The considerable advantage of MSL over RLS within a Lamarckian framework is, probably, due to the ability that the first model has to perform several modifications in the structure of the learning individual in each learning step. This gives to experiments with a Lamarckian strategy a higher chance of escaping from local optima. This situation was analysed in detail in a previous work [12].

Results achieved with the Baldwin Effect present two important distinctions. The first one is that the variation in results obtained by experiments using different learning models (RLS or MSL) is less significant. The Baldwin Effect took advantage of both models to improve the performance of the evolutionary algorithm, even though results with MSL are somewhat better. The other important difference is that the variation of the LR value does not seem to produce dramatic variations in the performance of the search algorithm. With just one exception (RLS model, LR = 1.0), all other experiments using the Baldwin Effect were able to outperform the results obtained by the standard evolutionary algorithm. This suggests that the behaviour of an EC algorithm that uses the Baldwin Effect is more robust and less dependable on specific learning issues (such as the kind of model employed or the learning rate).

Even though differences are not so evident as they are with Lamarckian learning, there is, nevertheless, a small advantage of MSL over RLS. When searching for solutions in highly irregular landscapes, giving to learning individuals the possibility to enlarge their neighbourhood region (this region includes all points to where an individual is allowed to jump in just one learning step) seems to provide them an important advantage. This advantage is particularly visible in experiments that use strategies that are prone to premature convergence, such as it is the case of Lamarckian learning.

In addition to increasing the likelihood of finding promising solutions, learning also helps evolution to discover them earlier in the search process. In tables 3 and 4 we present the periods in the simulation when TMs with productivity > 37 were found. In table 3 results concern experiments using the Baldwin Effect, whereas in table 4 they concern experiments with Lamarckian learning. It is possible to see that both learning strategies were able to consistently find promising TMs before 100 million evaluations. The *NoLearn* experiment only started to find such machines after this point. Moreover we can see that the main difference in efficiency between the Baldwin Effect and Lamarckian learning occurs before 50 million evaluations. In the period ranging from 1 to 50 million evaluations, experiments with the Baldwin Effect found 19 promising TMs. In the same period experiments with Lamarckian learning only found 10 TMs with productivity > 37.

Table 1: Productivity of the best individual of the final generation for each one of the 30 runs. Results concern experiments performed with the Baldwin Effect.

		NoLearn	**Baldwin Effect**									
			Random Local Search (RLS)					Multi Step Learning (MSL)				
			0.1	0.25	0.5	0.75	1.0	0.1	0.25	0.5	0.75	1.0
Productivity of the best TM	<= 35	12	14	22	17	19	25	16	16	20	23	21
	36-37	15	11	4	8	6	4	8	7	3	3	4
	38-160											
	161-164	3	5	4	5	5	1	6	7	7	4	5

Table 2: Productivity of the best individual of the final generation for each one of the 30 runs. Results concern experiments performed with Lamarckian learning.

		NoLearn	**Lamarckian Learning**									
			Random Local Search (RLS)					Multi Step Learning (MSL)				
			0.1	0.25	0.5	0.75	1.0	0.1	0.25	0.5	0.75	1.0
Productivity of the best TM	<= 35	12	18	17	22	21	20	16	22	19	22	19
	36-37	15	12	9	7	6	8	8	1	7	6	8
	38-160							1				
	161-164	3	1	4	1	3	2	5	7	4	2	3

Table 3: Period in the simulation when TMs with productivity > 37 were found. Results concern experiments performed with the Baldwin Effect and are divided over 6 temporal intervals.

		NoLearn	Baldwin Effect									
			Random Local Search (RLS)					Multi Step Learning (MSL)				
			0.1	0.25	0.5	0.75	1.0	0.1	0.25	0.5	0.75	1.0
Evaluations (Millions)	0-1											
	1-10		1					1	1	2		1
	10-50		1	2	2	3		3	3	2	2	1
	50-100		2	1	1	1	1	2	2		1	2
	100-150	2	1		1				1	1	1	
	150-200	1		1	1	1				2		1
	Totals	3	5	4	5	5	1	6	7	7	4	5

Table 4: Period in the simulation when TMs with productivity > 37 were found. Results concern experiments performed with Lamarckian learning and are divided over 6 temporal intervals.

		NoLearn	Lamarckian Learning									
			Random Local Search (RLS)					Multi Step Learning (MSL)				
			0.1	0.25	0.5	0.75	1.0	0.1	0.25	0.5	0.75	1.0
Evaluations (Millions)	0-1											
	1-10			1			1	1	1			
	10-50			1		2		3	1	1	1	
	50-100			1		1		1	2	3		2
	100-150	2	1	1					2			1
	150-200	1			1		1	1	1		1	
	Totals	3	1	4	1	3	2	6	7	4	2	3

Results from tables 3 and 4 confirm that, despite variations in efficiency, learning is really helping evolution in its search for good solutions for the BB problem. We would like to determine now some conditions that are required for such help to take place. We collected some results from the experiments that clarify what might be happening during the search process and in what way do evolution and learning interact. In tables 5 and 6 (respectively for the Baldwin Effect and Lamarckian learning experiments) we present the contribution of evolution and learning to the discovery of new best individuals during simulation. Contribution from evolution includes all new best individuals generated by crossover and/or mutation and contribution from learning includes all new best individuals that result from the application of one learning model. Contributions from each one of experiments are divided over 5 temporal intervals. We focus our analysis in the periods that range from 1 million to 100 million evaluations since, considering tables 3 and 4, this is the interval where learning experiments showed to be most advantageous. Looking to the results shown in tables 5 and 6, there are two important features that are common to experiments with better performance. The first one is the number of improvements obtained by evolution. If we take as comparison measure the number of improvements obtained by the standard EC algorithm (*NoLearn*) we see that most of the Baldwin Effect experiments were able to achieve a similar number of improvements due to evolution. In a considerable number of settings this value is even superior, which is a remarkable result, especially if we consider that a large number of evaluations is devoted to learning (e.g., in experiments with LR=0.1 half of the evaluations are spent in the learning process). This result reveals that learning is not preventing evolution from sampling the space. At the contrary, it is supporting evolution in this task, improving the rate at which new best solutions are found by crossover and mutation. Also, with the Baldwin Effect strategy, there is no significant difference in the number of improvements due to evolution obtained in similar experiments that use each one of the two models. This result confirms our previous conclusion that this strategy is able to take advantage from both learning models to improve its search performance.

On the other way results obtained by Lamarckian learning present significant variations. There is an important disparity between the number of improvements due to evolution presented by experiments with different learning models and LR values. In agreement with our previous analysis the larger number of improvements is seen in experiments with the MSL model and LR≤0.5.

The second important feature concerns the relative weight of learning in the process of discovering new best solutions. Values in parenthesis in the columns labelled

Learn represent the percentage of all improvements that were due to learning in the corresponding period. For both learning strategies there seems to be an inverse relationship between the weight of learning in discovering new best solutions and the performance of the search algorithm: experiments that obtain better results tend to present a moderate weight in improvements due to learning. This correlation is valid for all settings, even though there is a natural increase in the weight of learning in experiments that have a higher LR value. Most of the experiments with the Baldwin Effect are able to maintain the weight of learning at a moderate level, which explains the consistent results obtained. It is, nevertheless, visible that in experiments with the MSL model the weight of learning is smaller than in the ones that use the RLS model (for an equivalent LR value). This might justify the slight advantage presented by experiments with MSL over RLS within the Baldwin Effect framework.

In experiments performed with the Lamarckian strategy the weight of learning is always superior to the value obtained in similar conditions by the Baldwin Effect. In accordance to our hypothesis, lowest values are found precisely in experiments that obtained better results. This correlation suggests that it is evolution that should guide the search process. It is, nevertheless, important to keep a moderate exploitation pressure performed by a learning procedure, since results show that this pressure enables the early discovery of good solutions. A different scenario occurs if the percentage of improvements due to learning is too high. In this situation learning acts as the primary guiding force of the search process and evolution plays a secondary role.

Table 5: Contributions from evolution (Ev. columns) and learning (Learn columns) to the improvement of the best solution during simulation. For each experiment, results presented are the sum of 30 runs. Results are divided over 5 temporal periods. Values in parenthesis in the columns labelled Learn represent the percentage of all improvements from that period that were due to learning. Results concern experiments performed with the Baldwin Effect.

	Periods of the Simulation									
Evals. (Millions)	**0-1**		**1-10**		**10-50**		**50-100**		**100-200**	
	Ev.	**Learn**	**Ev.**	**Learn**	**Ev.**	**Learn**	**Ev.**	**Learn**	**Ev.**	**Learn**
NoLearn	1096		146		93		27		43	
RLS 0.1	863	179 (17)	126	17 (12)	74	22 (23)	21	10 (32)	29	4 (12)
RLS 0.25	614	353 (36)	170	35 (17)	47	8 (15)	35	15 (30)	29	9 (24)
RLS 0.5	479	497 (51)	159	86 (35)	87	31 (26)	25	17 (40)	32	24 (43)
RLS 0.75	379	597 (61)	166	77 (31)	84	60 (42)	54	23 (30)	28	16 (36)
RLS 1.0	249	642 (72)	136	90 (40)	65	35 (35)	12	14 (54)	24	18 (43)
MSL 0.1	850	90 (10)	166	8 (5)	87	9 (9)	28	7 (20)	22	2 (8)
MSL 0.25	813	167 (17)	173	25 (13)	150	24 (14)	37	9 (20)	32	5 (14)
MSL 0.5	613	263 (30)	154	24 (14)	83	14 (14)	26	4 (13)	37	12 (25)
MLS 0.75	533	320 (38)	147	20 (12)	61	15 (20)	35	7 (17)	23	5 (18)
MSL 1.0	433	382 (47)	124	40 (24)	91	27 (23)	17	5 (23)	45	11 (20)

Table 6: Contributions from evolution (Ev. columns) and learning (Learn columns) to the improvement of the best solution during simulation. For each experiment, results presented are the sum of 30 runs. Results are divided over 5 temporal periods. Values in parenthesis in the columns labelled Learn represent the percentage of all improvements from that period that were due to learning. Results concern experiments performed with Lamarckian learning.

	Periods of the Simulation									
Evals. (Millions)	**0-1**		**1-10**		**10-50**		**50-100**		**100-200**	
	Ev.	**Learn**	**Ev.**	**Learn**	**Ev.**	**Learn**	**Ev.**	**Learn**	**Ev.**	**Learn**
NoLearn	1096		146		93		27		43	
RLS 0.1	669	302 (31)	110	42 (28)	44	13 (23)	30	13 (30)	31	22 (41)
RLS 0.25	464	418 (47)	91	71 (44)	32	31 (49)	19	17 (47)	16	12 (43)
RLS 0.5	376	600 (62)	59	103 (64)	25	55 (69)	25	16 (39)	4	10 (71)
RLS 0.75	236	617 (72)	39	99 (72)	33	72 (69)	13	28 (68)	3	7 (70)
RLS 1.0	212	768 (78)	44	108 (71)	14	42 (75)	16	48 (75)	4	14 (78)
MSL 0.1	798	222 (22)	141	50 (26)	86	28 (25)	45	14 (24)	17	12 (41)
MSL 0.25	506	316 (38)	95	61 (39)	53	39 (42)	37	19 (34)	25	26 (51)
MSL 0.5	365	505 (58)	86	69 (45)	47	44 (48)	30	23 (43)	12	14 (54)
MLS 0.75	327	506 (61)	84	102 (55)	49	58 (54)	6	10 (63)	14	19 (58)
MSL 1.0	231	572 (71)	53	91 (63)	40	50 (56)	30	33 (52)	12	30 (71)

Since learning in this context is, by definition, a local procedure, search will most likely end up in the nearest local optimum. From the set of results presented it is possible to conclude that the Baldwin Effect strategy does not prevent evolution from sampling the space and it is more effective in adjusting its learning power to avoid premature convergence. Conversely, the Lamarckian strategy is not able to control this pressure. In the experiments we performed, it required specific conditions in what concerns the learning model and LR value to achieve competitive results. In most of the situations, it forced the EC to converge prematurely to some sub-optimal early discovered area.

6 CONCLUSIONS

In this paper we studied the interactions that exist between evolution and learning when searching for good solutions for the BB problem. We presented results of several experiments performed with two different learning strategies, the Baldwin Effect and Lamarckian learning. A careful analysis of the results allowed us to identify some conditions that should be met to maximise search performance: evolution should be the primary force responsible for sampling the landscape, although a moderate contribution from a learning procedure is very important to help evolution in its task. Experimental results also revealed that the Baldwin Effect strategy is less sensitive to specific issues concerning the definition of learning and it gives to the EC algorithm the possibility to maintain the diversity of the population, while exploiting the neighbourhood of areas already sampled by evolution. Additionally, results suggest that local search procedures with a considerable degree of freedom in what concerns the definition of local neighbourhood are more adapted to highly irregular landscapes.

Acknowledgments

This work was partially supported by the Portuguese Ministry of Science and Technology, under Program POSI.

References

1. Arita, T. and Suzuki, R. (2000). Interactions Between Learning and Evolution: The Outstanding Strategy Generated by the Baldwin Effect, In Bedau, M., McCasKill, J., Packard, N. and Rasmussen, S. (Eds.), *Proceedings of Artificial Life VII*, pp. 196-205.
2. Ackley, D. and Littman, M. (1994). A Case for Lamarckian Evolution. In Langton, C. (Ed.), *Artificial Life III*, pp. 3-10, Addison-Wesley.
3. Belew, R. and Mitchell, M. (1996). *Adaptive Individuals in Evolving Populations: Models and Algorithms*, Santa Fe Institute in the Sciences of Complexity, Vol. XXVI, Addison-Wesley.
4. Boolos, G., and Jeffrey, R. (1995). *Computability and Logic*, Cambridge University Press.
5. Hinton, G. E. and Nowlan, S. (1987). How learning can guide Evolution. *Complex Systems*, 1, pp. 495-502.
6. Jones, T. and Rawlins, G. (1993). Reverse Hillclimbing, Genetic Algorithms, and the Busy Beaver Problem. In Forrest, S. (Ed.). *Proceedings of the 5th International Conference on Genetic Algorithms (ICGA-93),* pp.70-75, Morgan Kaufmann.
7. Lally, A., Reineke, J. and Weader, J. (1997). An Abstract Representation of Busy Beaver Candidate Turing Machines, *Technical Report, Van Gogh Group*, Rensselaer Polytechnic Institute.
8. Machado, P., Pereira, F. B., Cardoso, A. and Costa, E. (1999). Busy Beaver: The Influence of Representation. In Poli, R., Nordin, P., Langdon, W. and Fogarty, T. (Eds.). *Proceedings of the Second Workshop on Genetic Programming (EuroGP-99),* pp. 29-38, Springer-Verlag.
9. Marxen, H. and Buntrock, J. (1990). Attacking Busy Beaver 5, *Bulletin of the European Association for Theoretical Computer Science*, Vol. 40.
10. Mayley, G. (1997). Landscapes, Learning Costs, and Genetic Assimilation. *Evolutionary Computation*, Vol. 4(3), 213-234.
11. Pereira, F. B., Machado, P., Costa, E. and Cardoso, A. (1999). Graph-Based Crossover: a Case Study with the Busy Beaver Problem. In Banzhaf, W., Daida, J. Eiben, A. E., Garzon, M., Honavar, V., Jakiela, M. and Smith, R. E. (Eds.), *GECCO-99: Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 1149-1155, Morgan Kaufmann.
12. Pereira, F. B. and Costa, E. (2001). The Influence of Learning in the Evolution of Busy Beavers. To appear in the *Proceedings of EvoLEARN2001*.
13. Rado, T. (1962). On non-computable functions, *The Bell System Technical Journal*, V. 41(3), pp. 877-884.
14. Sasaki, T. and Tokoro, M. (1999). Evolving Learnable Neural Networks Under Changing Environments with Various Rates of Inheritance of Acquired Characters: Comparison of Darwinian and Lamarckian Evolution, *Artificial Life*, Vol. 5, No. 3, pp. 203-223.
15. Turney, P., Whitley, D., and Anderson, R. (1997). Evolution, Learning and Instinct: 100 Years of the Baldwin Effect*, Evolutionary Computation*, Vol. 4(3), iv-viii.
16. Whitley, D., Gordon, S. and Mathias, K. (1994). Lamarckian Evolution, the Baldwin Effect and Function Optimization. In Davidor, Y., Schwefel, H. P. and Manner, R. (Eds.), *Parallel Problem Solving from Nature(PPSN III)*, pp. 6-15. Berlin: Springer-Verlag.
17. Wood, D. (1987). *Theory of Computation*, Harper & Row, Publishers.

Effects of Swarm Size on Cooperative Particle Swarm Optimisers

F. van den Bergh and **A.P. Engelbrecht**
Department of Computer Science
University of Pretoria
South Africa
fvdbergh@cs.up.ac.za, engel@driesie.cs.up.ac.za

Abstract

Particle Swarm Optimisation is a stochastic global optimisation technique making use of a population of particles, where each particle represents a solution to the problem being optimised. The Cooperative Particle Swarm Optimiser (CPSO) is a variant of the original Particle Swarm Optimiser (PSO). This technique splits the solution vector into smaller vectors, where each sub-vector is optimised using a separate PSO. This paper investigates the effect of swarm size on the CPSO, showing that the CPSO does not exhibit the same general trend as the original PSO.

1 INTRODUCTION

The particle swarm optimiser, first introduced in [Eberhart and Kennedy, 1995], has proven to be a useful global optimisation algorithm, with applications in neural network training [Engelbrecht and Ismail, 1999], [van den Bergh and Engelbrecht, 2000], function minimisation [Shi and Eberhart, 1999], [Shi and Eberhart, 1998] and human tremor analysis [Eberhart and Hu, 1999].

The Cooperative Particle Swarm Optimiser (CPSO, or split swarm) is a recent modification to the original PSO algorithm leading to a significant reduction in training time [van den Bergh and Engelbrecht, 2000, van den Bergh and Engelbrecht, 2001]. The cooperative approach increased the number of adjustable parameters in the PSO algorithm significantly. This paper studies the effect the number of particles in the swarm has on the CPSO algorithm, also providing results for the original PSO for reference.

Section 2 provides a brief description of the basic PSO, with Appendix A listing the complete algorithms. Section 3 briefly outlines the expectations for the experiments described in Section 4, the results of which are provided in Section 5. The paper is concluded with some findings and directions for future research in Section 6.

2 PARTICLE SWARM OPTIMISERS

The PSO, like a Genetic Algorithm, is a population based optimisation technique, but the population is now called a *swarm*.

Each individual i has the following attributes: A current position in search space, $\mathbf{x}_i$, a current velocity, $\mathbf{v}_i$, and a personal best position in search space, $\mathbf{y}_i$. During each iteration each particle in the swarm is updated using (1) and (2). Assuming that the function f is to be minimised, that the swarm consists of n particles, and $r_1 \sim U(0,1)$, $r_2 \sim U(0,1)$ are elements from two uniform random sequences in the range $(0,1)$, then:

$$\mathbf{v}_i := w\mathbf{v}_i + c_1 r_1(\mathbf{y}_i - \mathbf{x}_i) + c_2 r_2(\hat{\mathbf{y}} - \mathbf{x}_i) \quad (1)$$

$$\mathbf{x}_i := \mathbf{x}_i + \mathbf{v}_i \quad (2)$$

$$\mathbf{y}_i := \begin{cases} \mathbf{y}_i & \text{if } f(\mathbf{x}_i) \geq f(\mathbf{y}_i) \\ \mathbf{x}_i & \text{if } f(\mathbf{x}_i) < f(\mathbf{y}_i) \end{cases} \quad (3)$$

$$\hat{\mathbf{y}} \in \{\mathbf{y}_0, \mathbf{y}_1, \ldots, \mathbf{y}_n\} \mid f(\hat{\mathbf{y}}) = \min(f(\mathbf{y}_0), f(\mathbf{y}_1), \ldots, f(\mathbf{y}_n)) \quad (4)$$

Note that $\hat{\mathbf{y}}$ is therefore the global best position amongst all the particles. The value of each dimension of every $\mathbf{v}_i$ vector is clamped to the range $[-v_{max}, v_{max}]$ to prevent the PSO from leaving the search space. The value of v_{max} is usually chosen to

be $k \times x_{max}$, with $0.1 \leq k \leq 1.0$ [Eberhart et al., 1996]. Note that this does not restrict the values of $\mathbf{x}_i$ to the range $[-v_{max}, v_{max}]$; it only limits the maximum distance that a particle will move during one iteration.

The variable w in (1) is called the *inertia weight*; this value is typically set up to vary linearly from 1 to near zero during the course of a training run. Larger values for w result in smoother, more gradual changes in direction through search space. Toward the end of the training run smaller inertia coefficients allow particles to settle into the minimum.

Acceleration coefficients c_1 and c_2 also control how far a particle will move in a single iteration. Typically these are both set to a value of 2 [Eberhart et al., 1996], although assigning different values to c_1 and c_2 sometimes leads to improved performance [Suganthan, 1999].

Appendix A lists the algorithm for the original PSO. Two other algorithms are also provided: the 'split' PSO and the 'hybrid' PSO [van den Bergh and Engelbrecht, 2001].

The split algorithm (Figure 2) takes the n-dimensional solution vector and breaks it into n one-dimensional components. Each component is then optimised by a separate PSO. The error function is evaluated using a vector formed by concatenating the components from the n swarms to again form an n-dimensional vector. The hybrid algorithm (Figure 3) combines a standard PSO with a split PSO, sharing information at the end of each iteration.

3 PARTICLE SWARM SIZE

The experiments performed below were designed to study the behaviour of three different versions of the particle swarm optimiser by varying the number of particles allocated to each swarm.

The swarm size is a critical parameter in the original PSO algorithm — too few particles will cause the algorithm to become stuck in local minima, while too many particles will slow down the algorithm.

Assume that a simulation will run over I iterations, using an n-dimensional function. The number of Error Function Evaluations (EFEs) will be equal to the product of the swarm size, P, and the number of iterations, thus $E = I \times P$. This equation holds for the original PSO algorithm.

To compare the plain PSO to the split (and thus also hybrid) swarms, the number of EFEs must be kept constant. The number of EFEs in the split swarm is given by $E = I \times P \times n$ (see Appendix A, [van den Bergh and Engelbrecht, 2000, van den Bergh and Engelbrecht, 2001]).

Thus, for a fixed number of EFEs, a choice has to be made, either choosing a larger swarm (more variety) or having more iterations. It is clear that more iterations will eventually improve the quality of the solution, as long as the whole swarm does not get stuck in a local minimum — this follows from the fact that the position yielding the lowest error so far is stored, so that evaluating the function at the best particle in the swarm yields a strictly non-increasing sequence.

Knowing that a large number of iterations will improve the quality of the solution, the problem is now to find the optimal balance between the number of iterations (I) and swarm size (P) for a fixed value of E. The experiments in the next section will test different types of swarms using varying swarm sizes, where the number of iterations is calculated using $I = E/P$ for the plain swarm, $I = E/(n \times P)$ for the split swarm and $I = E/((n+1) \times P)$ for the hybrid swarm.

4 FUNCTIONS

The following functions were used during testing:

The Sphere function:

$$f_0(\mathbf{x}) = \sum_{i=1}^{n} x_i^2 \tag{5}$$

The Rosenbrock (or banana-valley) function:

$$f_1(\mathbf{x}) = \sum_{i=1}^{n} \left(100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2\right) \tag{6}$$

Schaffer's f_6 function:

$$f_2(\mathbf{x}) = 0.5 - \frac{\left(\sin\sqrt{x^2 + y^2}\right)^2 - 0.5}{\left(1.0 + 0.001\,(x^2 + y^2)\right)^2} \tag{7}$$

The generalised Rastrigin function:

$$f_3(\mathbf{x}) = \sum_{i=1}^{n} \left(x_i^2 - 10\cos(2\pi x_i) + 10\right) \tag{8}$$

The generalised Griewank function:

$$f_4(\mathbf{x}) = \frac{1}{4000}\sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1 \tag{9}$$

Table 1: Parameters Used During Experiments

Function	n	domain	stop
f_0	30	100	0.01
f_1	30	30	100
f_2	2	100	0.00001
f_3	30	5.12	100
f_4	30	600	0.1

These functions represent a mixture in terms of the correlation between the variables used. For example, the Sphere and Rastrigin functions are completely uncorrelated, and thus the variables can be optimised independently. The Griewank function is highly correlated (because it includes a product of all the terms), whereas the Rastrigin and Schaffer functions fall somewhere in between.

Table 1 contains the parameters used to obtain the results below. The 'domain' column in the table specifies the domain size from which the initial random particles are selected. The 'stop' column lists the threshold used as a stopping criterion on the second set of tests, corresponding to the results in Tables 3, 5, 7, 9 and 11.

Two types of experiment were performed: A fixed iteration experiment, reporting the final error and a test to determine how quickly a certain error threshold can be reached.

plain/split/hybrid: A swarm using $c_1 = 2.0$, $c_2 = 2.0$, w decreases linearly over time, and v_{max} is clamped to the domain (refer to Table 1). The algorithm stops when the maximum number of error function evaluations (EFEs) have been used, fixed at 2×10^5.

plainB/splitB/hybridB: A swarm using $c_1 = 2.0$, $c_2 = 2.0$, w decreases linearly over time, v_{max} is clamped to domain. The algorithm stops when the error drops below the 'stop' value listed in Table 1. Note that the values specified in this table correspond to the values used by Eberhart and Shi [Eberhart and Shi, 2000]. Each experiment consisted of 500 runs.

Due to the different structure of the CPSO algorithms (split and hybrid) it is not possible to compare them to the original PSO based on the number of iterations that the algorithm performed. Instead, the number of times that the function under consideration has been evaluated is measured, as this corresponds to the amount of 'work' done. This unit is expressed as the number of Error Function Evaluations, or EFEs.

5 RESULTS

Table 2: Sphere (f_0) After 2×10^5 EFEs

Type	P	Mean
plain	5	5.7283e−07 ± 4.38e−07
	10	4.5000e−19 ± 7.04e−19
	15	4.7235e−24 ± 5.99e−24
	20	1.1770e−26 ± 1.32e−26
	25	3.7339e−26 ± 7.16e−26
	30	9.0862e−26 ± 8.52e−26
split	5	1.0939e−55 ± 1.66e−55
	10	3.4251e−90 ± 6.71e−90
	15	1.6621e−80 ± 2.31e−80
	20	2.0411e−63 ± 1.71e−63
	25	6.0312e−52 ± 1.03e−51
	30	2.4475e−44 ± 2.53e−44
hybrid	5	1.5972e−56 ± 3.12e−56
	10	9.2974e−88 ± 1.82e−87
	15	1.4860e−76 ± 2.88e−76
	20	3.3054e−60 ± 5.62e−60
	25	3.8156e−50 ± 7.10e−50
	30	9.2566e−45 ± 4.54e−45

Table 3: Sphere (f_0) Computational Complexity

Type	P	N	Iters	EFEs	Time
plainB	5	500	14694.5	73476	0.778
	10	500	7717.3	77178	0.817
	15	500	5357.7	80374	0.851
	20	500	4159.1	83193	0.880
	25	500	3420.3	85521	0.906
	30	500	2922.8	87700	0.931
splitB	5	500	273.5	41108	0.091
	10	500	148.3	44633	0.088
	15	500	104.4	47198	0.090
	20	500	81.8	49406	0.093
	25	500	68.0	51390	0.095
	30	500	57.6	52269	0.099
hybridB	5	500	113.9	17729	0.046
	10	500	87.4	27253	0.064
	15	500	71.8	33599	0.076
	20	500	61.2	38268	0.085
	25	500	53.1	41498	0.092
	30	500	47.4	44543	0.101

Tables 2, 4, 6, 8 and 10 conform to the following format: The 'type' column lists the algorithm used, the 'P' column the number of particles per swarm and the

Table 4: Rosenbrock (f_1) After 2×10^5 EFEs

Type	P	Mean
plain	5	1.0296e+02 ± 1.10e+01
	10	5.1942e+01 ± 4.62e+00
	15	4.1858e+01 ± 3.39e+00
	20	3.8880e+01 ± 3.03e+00
	25	4.2280e+01 ± 3.26e+00
	30	4.3518e+01 ± 3.31e+00
split	5	3.3706e+01 ± 2.93e+00
	10	2.0797e+01 ± 2.32e+00
	15	1.2553e+01 ± 1.63e+00
	20	1.0304e+01 ± 1.40e+00
	25	1.0964e+01 ± 1.56e+00
	30	1.1049e+01 ± 1.60e+00
hybrid	5	8.0058e+00 ± 8.24e−01
	10	7.7328e+00 ± 6.52e−01
	15	1.0439e+01 ± 1.31e+00
	20	9.4816e+00 ± 1.40e+00
	25	1.0479e+01 ± 1.49e+00
	30	1.1758e+01 ± 1.73e+00

Table 5: Rosenbrock (f_1) Computational Complexity

Type	P	N	Iters	EFEs	Time
plainB	5	385	15409.6	77051	1.466
	10	447	8380.2	83808	1.333
	15	455	5910.5	88665	1.369
	20	463	4589.2	91794	1.383
	25	460	3784.5	94625	1.429
	30	470	3237.7	97146	1.435
splitB	5	495	206.7	31083	0.181
	10	499	102.8	31012	0.166
	15	500	67.7	30694	0.160
	20	500	44.3	26902	0.139
	25	500	34.6	26346	0.136
	30	500	27.6	25252	0.131
hybridB	5	498	110.2	17174	0.106
	10	499	60.1	18787	0.109
	15	500	45.9	21570	0.122
	20	500	36.1	22696	0.128
	25	500	28.9	22801	0.129
	30	500	23.2	22026	0.130

Table 6: Schaffer (f_2) After 2×10^5 EFEs

Type	P	Mean
plain	5	2.8397e−04 ± 1.47e−04
	10	0.0000e+00 ± 0.00e+00
	15	0.0000e+00 ± 0.00e+00
	20	0.0000e+00 ± 0.00e+00
	25	0.0000e+00 ± 0.00e+00
	30	0.0000e+00 ± 0.00e+00
split	5	1.7458e−02 ± 1.18e−03
	10	1.6827e−02 ± 1.15e−03
	15	1.6970e−02 ± 1.21e−03
	20	1.7530e−02 ± 1.23e−03
	25	1.6337e−02 ± 1.15e−03
	30	1.7342e−02 ± 1.21e−03
hybrid	5	6.0146e−03 ± 4.22e−04
	10	4.8884e−03 ± 4.35e−04
	15	4.9378e−03 ± 4.35e−04
	20	4.5638e−03 ± 4.35e−04
	25	4.4222e−03 ± 4.34e−04
	30	4.2799e−03 ± 4.32e−04

Table 7: Schaffer (f_2) Computational Complexity

Type	P	N	Iters	EFEs	Time
plainB	5	475	9575.6	47881	0.064
	10	494	4610.9	46115	0.055
	15	495	3252.7	48798	0.057
	20	499	2462.4	49258	0.056
	25	499	1977.5	49449	0.056
	30	500	1670.1	50119	0.057
splitB	5	44	2189.9	21905	0.175
	10	51	1186.7	23743	0.164
	15	46	697.9	20949	0.163
	20	41	531.6	21280	0.163
	25	48	473.8	23717	0.160
	30	43	368.5	22137	0.161
hybridB	5	211	6090.1	91364	0.183
	10	261	2265.1	67976	0.139
	15	260	1413.4	63639	0.132
	20	243	1008.1	60532	0.134
	25	277	865.7	64986	0.126
	30	269	706.1	63615	0.127

Table 8: Rastrigin (f_3) After 2×10^5 EFEs

Type	P	Mean
plain	5	4.4038e+01 ± 1.02e+00
	10	3.8128e+01 ± 8.31e−01
	15	3.5212e+01 ± 7.25e−01
	20	3.4109e+01 ± 7.14e−01
	25	3.2252e+01 ± 6.80e−01
	30	3.0904e+01 ± 6.42e−01
split	5	0.0000e+00 ± 0.00e+00
	10	0.0000e+00 ± 0.00e+00
	15	0.0000e+00 ± 0.00e+00
	20	0.0000e+00 ± 0.00e+00
	25	0.0000e+00 ± 0.00e+00
	30	0.0000e+00 ± 0.00e+00
hybrid	5	4.9852e−02 ± 1.95e−02
	10	0.0000e+00 ± 0.00e+00
	15	0.0000e+00 ± 0.00e+00
	20	0.0000e+00 ± 0.00e+00
	25	0.0000e+00 ± 0.00e+00
	30	0.0000e+00 ± 0.00e+00

Table 9: Rastrigin (f_3) Computational Complexity

Type	P	N	Iters	EFEs	Time
plainB	5	498	13059.3	65299	1.001
	10	500	6741.8	67423	1.013
	15	500	4529.4	67950	1.022
	20	500	3469.9	69409	1.043
	25	500	2797.6	69954	1.052
	30	500	2383.8	71531	1.078
splitB	5	500	1.8	363	0.003
	10	500	0.8	381	0.003
	15	500	0.1	432	0.004
	20	500	0.0	545	0.004
	25	500	0.0	666	0.006
	30	500	0.0	786	0.006
hybridB	5	500	2.5	465	0.004
	10	500	0.8	390	0.003
	15	500	0.1	436	0.004
	20	500	0.0	546	0.005
	25	500	0.0	667	0.006
	30	500	0.0	787	0.008

Table 10: Griewank (f_4) After 2×10^5 EFEs

Type	P	Mean
plain	5	3.8076e−02 ± 5.10e−03
	10	2.6663e−02 ± 2.62e−03
	15	2.0078e−02 ± 1.91e−03
	20	1.7960e−02 ± 1.89e−03
	25	1.5639e−02 ± 1.55e−03
	30	1.4775e−02 ± 1.64e−03
split	5	2.6151e−02 ± 2.55e−03
	10	2.2604e−02 ± 2.14e−03
	15	2.4547e−02 ± 2.46e−03
	20	2.3048e−02 ± 2.31e−03
	25	2.1116e−02 ± 2.30e−03
	30	2.1320e−02 ± 2.44e−03
hybrid	5	4.1690e−02 ± 3.63e−03
	10	2.5982e−02 ± 2.56e−03
	15	2.4354e−02 ± 2.33e−03
	20	2.4916e−02 ± 2.53e−03
	25	2.3362e−02 ± 2.45e−03
	30	2.1860e−02 ± 2.22e−03

Table 11: Griewank (f_4) Computational Complexity

Type	P	N	Iters	EFEs	Time
plainB	5	457	14639.4	73200	1.498
	10	489	7639.5	76401	1.410
	15	495	5289.9	79356	1.436
	20	494	4088.5	81779	1.483
	25	499	3368.4	84222	1.507
	30	499	2875.7	86286	1.545
splitB	5	494	222.1	33367	0.335
	10	494	114.1	34341	0.335
	15	489	78.1	35293	0.356
	20	491	59.0	35632	0.352
	25	492	46.9	35425	0.346
	30	489	38.9	35295	0.353
hybridB	5	456	98.0	15254	0.201
	10	484	68.1	21228	0.168
	15	490	53.2	24913	0.176
	20	495	43.3	27084	0.178
	25	488	37.4	29286	0.207
	30	498	32.3	30346	0.195

'Mean' column the average error after 2×10^5 EFEs, followed by the 95% confidence interval width.

The original PSO algorithm, called 'plain' here, exhibits the expected behaviour for the Sphere, Schaffer, Rastrigin and Griewank functions, where the average final error decreases as the number of particles per swarm is increased. This is in part due to the fact that adding more particles results in more starting positions, increasing the probability of having particles in suitable positions for finding the minimum of the function.

The Rosenbrock function appears to behave somewhat erratically, with the error decreasing up to 20 particles, and then increasing again. Even over 500 simulation runs sampling error remains a possibility which cannot be completely disregarded here.

Split swarm behaviour is more interesting, with only the Rosenbrock and Griewank functions exhibiting typical behaviour — the error decreasing with an increase in the number of particles per swarm. The Sphere function exhibits a strongly decreasing trend as the number of particles per swarm (pps) increases, with the exception of the 5 pps case. Schaffer's function, tested with only two dimensions, is the Achilles' heel of the split swarm, not only producing large errors but also behaving erratically as the number of particles is increased. It is currently thought that the split swarm performs better on higher-dimensional functions, preferably functions with little correlation between its variables. Rastrigin's function, on the other hand, is the split swarm's forte, resulting in a perfect error of zero for all pps values tested.

The performance of the split swarm is superior to that of the plain swarm on three of the functions tested (Sphere, Rastrigin, Rosenbrock), comparable on the Griewank function and worse on the Schaffer function. Part of the improved performance can be attributed to the fact that the split swarm generates more particles 'on-the-fly' by combining the particles from the different swarms (see Figure 2), forming between $5 \times 30 = 150$ and $15 \times 30 = 450$ particles, if the function is tested with 30 dimensions. For more details on this algorithm, see [van den Bergh and Engelbrecht, 2001].

By combining features of both the split and the plain swarm, the hybrid swarm produces some interesting results as well. A decrease in error is observed for the Schaffer and Griewank functions when increasing the number of particles per swarm. The variables of these two functions are strongly correlated, thus the plain-swarm component contained in the hybrid algorithm dominates, resulting in the same behaviour pattern as observed with the plain swarm. The Sphere and Rastrigin functions exhibit behaviour similar to the split swarm, with a few outliers at 5 pps distorting the Rastrigin results. The Rosenbrock function shows an increase in error as the number of particles per swarm is increased, behaviour that is not fully explained by either the plain or the split component of the algorithm.

These fixed-iteration tests provide an idea of the quality of the solutions that the different algorithms can attain on the benchmark problems, given an equal number of function evaluations. No analysis of variance tests were performed, as the homoscedasticity assumption was not satisfied, nor were the data points from a Gaussian distribution, as they tended to cluster at the lower bounds.

Tables 3, 5, 7, 9 and 11 should be interpreted as follows: The 'N' column indicates the number of simulation runs that successfully reached the error threshold (out of a maximum of 500), followed by columns listing the average number of iterations and the average number of EFEs required to reach the threshold. The last column lists the average time needed per simulation run, in seconds.

This second set of tests give an indication of the speed with which the algorithms cross an error threshold, thus lower values are better. In this test it is possible that the simulation run could not reach the threshold within 2×10^5 EFEs; such cases were omitted from the calculation of the average number of EFEs required.

For the plain swarm, the trend appears to be that increasing the number of particles per swarm results in an increase in the average number of EFEs required to reach the threshold. The number of EFEs did not vary much within a problem. Fewer particles per swarm increased the likelihood that some of the simulations would not reach the threshold. This behaviour is in line with the findings of the fixed-iteration tests above, so that more particles per swarm leads to better performance, at the cost of using more EFEs on average to reach the same threshold. From this one can conclude that a larger number of particles per swarm should be used for the plain swarm, preferably more than 20 particles.

The split swarm exhibits the expected increase in the number of EFEs for the Sphere and Rastrigin functions, while the Rosenbrock function shows a decrease. The decrease experienced with the Rosenbrock function hints at a sensitivity to the initial positions of the particles in the swarm, where larger swarms have a higher probability of including better starting positions. Lastly, the Griewank and Schaffer functions

show almost no trend, the Schaffer results being somewhat erratic. Note that the split swarm rarely reached the error threshold on the Schaffer test.

Overall, the split swarm reached the threshold significantly faster than the plain swarm, with the troublesome Schaffer function being the only case where it failed to reach the threshold consistently.

Hybrid swarm performance follows the same pattern as the plain swarm, thus an increase in the number of EFEs results as the number of particles is increased. This applies to all functions but the Schaffer function, which is again somewhat erratic. Note that the number of EFEs required to reach the threshold for the Schaffer function was larger than that of either the plain or the split swarm. The hybrid swarm managed to reach the threshold in at least half of the experiments, a considerable improvement over the split swarm.

With the exception of the Schaffer function, the hybrid swarm reached the threshold faster than the plain swarm, and even faster than the split swarm on all but the Schaffer and Rastrigin (marginal difference) functions.

6 CONCLUSIONS AND FUTURE WORK

Some questions about the optimal number of particles per swarm can now be answered. For the plain swarm it appears that larger swarm sizes improve the quality of the solution, but it also increases the number of EFEs required to reach a specific error threshold.

The split swarms appear to have a far smaller optimal choice of between 5 and 20 particles per swarm. This can be explained by looking at the algorithm for the split swarm (Figure 2, Appendix A). The algorithm forms solution vectors by combining different vectors from different swarms, effectively creating more 'variety' out of fewer particles. Thus a total of 300 *different* vectors (actually, 30 sets of vectors where each set of 10 vectors will differ in only one of the 30 components) will be formed for a 30-dimensional function, assuming 10 particles per swarm. The equivalent plain swarm will only have 10 different vectors. This explains why the split swarm has a 'sweet spot' for the swarm size — the point where the optimal balance between variety and the number of iterations is reached.

The hybrid swarm has range of about 5–10 for uncorrelated functions, but behaves more like the plain swarm for correlated functions (*e.g.* Schaffer and Griewank). The plain swarm will also have a 'sweet spot', but this will probably be at a swarm size of more than 30, at which point it will not be able to compete with the split or hybrid swarms in terms of speed.

Note that the optimal number of particles per swarm will depend on the function, but from the results it appears that the split swarm (and thus to some extent the hybrid swarm) requires fewer particles than the plain swarm.

This paper addresses the effect of one of the parameters in the split swarm architecture, namely the number of particles per swarm. The effect of the other variables, like the inertia coefficient or the acceleration coefficients still remains to be investigated. For now, the values that work well for the plain swarm have been used; more experiments have to be performed to check whether these values are still good choices for the split swarm.

References

[Eberhart and Hu, 1999] Eberhart, R. C. and Hu, X. (1999). Human Tremor Analysis Using Particle Swarm Optimization. In *Proceedings of the Congress on Evolutionary Computation*, pages 1927–1930, Washington D.C, USA.

[Eberhart and Kennedy, 1995] Eberhart, R. C. and Kennedy, J. (1995). A New Optimizer using Particle Swarm Theory. In *Proceedings of the Sixth International Symposium on Micro Machine and Human Science*, pages 39–43, Nagoya, Japan. IEEE Service Center.

[Eberhart and Shi, 2000] Eberhart, R. C. and Shi, Y. (2000). Comparing Inertia Weights and Constriction Factors in Particle Swarm Optimization. In *Proceedings of the Congress on Evolutionary Computing*, pages 84–89.

[Eberhart et al., 1996] Eberhart, R. C., Simpson, P., and Dobbins, R. (1996). *Computational Intelligence PC Tools*, chapter 6, pages 212–226. Academic Press Professional.

[Engelbrecht and Ismail, 1999] Engelbrecht, A. P. and Ismail, A. (1999). Training product unit neural networks. *Stability and Control: Theory and Applications*, 2(1–2):59–74.

[Shi and Eberhart, 1998] Shi, Y. and Eberhart, R. C. (1998). A Modified Particle Swarm Optimizer. In *IEEE International Conference of Evolutionary Computation*, Anchorage, Alaska.

[Shi and Eberhart, 1999] Shi, Y. and Eberhart, R. C. (1999). Empirical Study of Particle Swarm Optimization. In *Proceedings of the Congress on Evolutionary Computation*, pages 1945–1949, Washington D.C, USA.

[Suganthan, 1999] Suganthan, P. N. (1999). Particle Swarm Optimizer with Neighbourhood Operator. In *Proceedings of the Congress on Evolutionary Computation*, pages 1958–1961, Washington D.C, USA.

[van den Bergh and Engelbrecht, 2000] van den Bergh, F. and Engelbrecht, A. P. (2000). Cooperative Learning in Neural Networks using Particle Swarm Optimizers. *South African Computer Journal*, (26):84–90.

[van den Bergh and Engelbrecht, 2001] van den Bergh, F. and Engelbrecht, A. P. (2001). A Cooperative Approach to Particle Swarm Optimisation. *IEEE Transactions on Evolutionary Computing*. Currently under review.

A PSO Algorithms

Create and initialise an n-dimensional PSO : S
repeat:
  for $j \in [1..M]$ **:**
    if $f(S.\mathbf{x}_j) < f(S.\mathbf{y}_j)$
      then $S.\mathbf{y}_j = S.\mathbf{x}_j$
    if $f(S.\mathbf{y}_j) < f(S.\hat{\mathbf{y}})$
      then $S.\hat{\mathbf{y}} = S.\mathbf{y}_j$
  endfor
  Perform updates on S using eqns. (1–2)
until stopping criterion is met

Figure 1: Pseudo Code for the Plain Swarm Algorithm

define
  $\mathbf{b}(j, z) \equiv (S_1.\hat{\mathbf{y}}, \ldots, S_{j-1}.\hat{\mathbf{y}}, z, S_{j+1}.\hat{\mathbf{y}}, \ldots, S_n.\hat{\mathbf{y}})$
Initialise n 1-dimensional PSOs : $S_i,\ i \in [1..n]$
repeat:
  for $i \in [1..n]$ **:**
    for $j \in [1..M]$ **:**
      if $f(\mathbf{b}(i, S_i.\mathbf{x}_j)) < f(\mathbf{b}(i, S.\mathbf{y}_j))$
        then $S.\mathbf{y}_j = S.\mathbf{x}_j$
      if $f(\mathbf{b}(i, S_i.\mathbf{y}_j)) < f(\mathbf{b}(i, S_i.\hat{\mathbf{y}}))$
        then $S_i.\hat{\mathbf{y}} = S_i.\mathbf{y}_j$
    endfor
    Perform updates on S_i using eqns. (1–2)
  endfor
until stopping criterion is met

Figure 2: Pseudo Code for the Split Swarm Algorithm

define
  $\mathbf{b}(j, z) \equiv (S_1.\hat{\mathbf{y}}, \ldots, S_{j-1}.\hat{\mathbf{y}}, z, S_{j+1}.\hat{\mathbf{y}}, \ldots, S_n.\hat{\mathbf{y}})$
Initialise n one-dimensional PSOs : $S_i,\ i \in [1..n]$
Initialise an n-dimensional PSO : P
repeat:
  for $i \in [1..n]$ **:**
    for $j \in [1..M]$ **:**
      if $f(\mathbf{b}(i, S_i.\mathbf{x}_j)) < f(\mathbf{b}(i, S.\mathbf{y}_j))$
        then $S.\mathbf{y}_j = S.\mathbf{x}_j$
      if $f(\mathbf{b}(i, S_i.\mathbf{y}_j)) < f(\mathbf{b}(i, S_i.\hat{\mathbf{y}}))$
        then $S_i.\hat{\mathbf{y}} = S_i.\mathbf{y}_j$
    endfor
    Perform updates on S_i using eqns. (1–2)
  endfor
  Select random $k \sim U(1, M) \cdot P.\mathbf{y}_k \neq P.\hat{\mathbf{y}}$
  $P.\mathbf{x}_k = \mathbf{b}(1, S_1.\hat{\mathbf{y}})$
  for $j \in [1..M]$ **:**
    if $f(P.\mathbf{x}_j) < f(P.\mathbf{y}_j)$
      then $P.\mathbf{y}_j = P.\mathbf{x}_j$
    if $f(P.\mathbf{y}_j) < f(P.\hat{\mathbf{y}})$
      then $P.\hat{\mathbf{y}} = P.\mathbf{y}_j$
  endfor
  Perform updates on P using eqns. (1–2)
  for $i \in [1..n]$ **:**
    Select random $k \sim U(1, M) \cdot S_i.\mathbf{y}_k \neq S_i.\hat{\mathbf{y}}$
    $S_i.\mathbf{x}_k = P.\hat{\mathbf{y}}_i$
  endfor
until stopping criterion is met

Figure 3: Pseudo Code for the Hybrid Swarm Algorithm