Memory Analysis and Significance Test for Agent Behaviours

DaeEun Kim
University of Leicester, University Road, Leicester, LE1 7RH, United Kingdom
DaeEun.Kim@le.ac.uk

ABSTRACT

Many agent problems in a grid world have a restricted sensory information and motor actions. The environmental conditions need dynamic processing of internal memory. In this paper, we handle the artificial ant problem, an agent task to model ant trail following in a grid world, which is one of the difficult problems that purely reactive systems cannot solve. We provide an evolutionary approach to quantify the amount of memory needed for the agent problem and explore a systematic analysis over the memory usage. We apply two types of memory-based control structures, Koza's genetic programming and finite state machines, to recognize the relevance of internal memory. Statistical significance test based on beta distribution differentiates the characteristics and performances of the two control structures.

Categories and Subject Descriptors: I.2.8 [Artificial Intelligence]: Problem Solving, Control Methods, and Search; I.2 [[Artificial Intelligence]: Miscellaneous; H.4 [Information Systems Applications]: Miscellaneous

General Terms: Algorithms

Keywords: grid world problem, finite state machines, genetic programming, internal states, computational effort

1. INTRODUCTION

Many agent problems in a grid world have been handled to understand agent behaviours in the real world or pursue characteristics of desirable controllers. Normally grid world problems have a set of restricted sensory configurations and motor actions. Memory control architecture is often needed to process the agent behaviours appropriately. Finite state machines and recurrent neural networks were used in the artificial ant problems [7]. Koza [13] applied genetic programming with a command sequence function to the artificial ant problem. Teller [24] tested a Tartarus problem by using an indexed memory. Wilson [25] used a new type of memory-based classifier system for a grid world problem, the Woods problem.

GECCO'06, July 8–12, 2006, Seattle, Washington, USA.

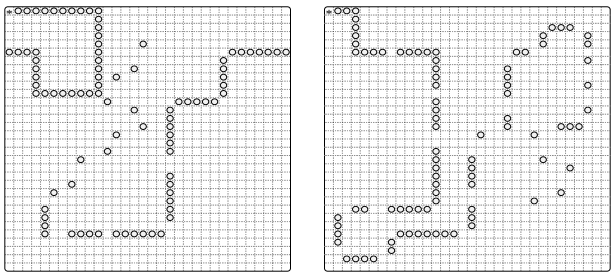


Figure 1: Artificial ant trails (a) John Muir trail (b) Santa Fe trail (*: ant agent, ○: food)

The artificial ant problem is a simple navigation task that imitates ant trail following. In this problem, an agent must follow irregular food trails in the grid world to imitate an ant's foraging behavior. The trails have a series of turns, gaps, and jumps on the grid and ant agents have one sensor in the front to detect food. Agents have restricted information of the surrounding environment. Yet they are supposed to collect all the food on the trails. The first work, by Jefferson et al. [7], used the John Muir trail, and another trail, called Santa Fe trail, was studied with genetic programming by Koza [13]. The trails are shown in Figure 1.

This problem was first solved with a genetic algorithm to test the representation problem of controllers by Jefferson et al. [7]. A large population of artificial ants (65,536) were simulated in the John Muir trail with two different controller schemes, finite state machines and recurrent neural networks. Each ant has a detector to sense the environment and an effector to wander about the environment; one bit of sensory input to detect food and two bits of motor actions to move forward, turn right, turn left and think (no operation). Its fitness was measured by the amount of food it collects in 200 time steps. At each time step, the agent can sense the environment and decide on one of the motor actions. The behaviors of ant agents in the initial population were random walks. Gradually more food trails were traced by evolved ants. Koza [13] applied genetic programming to the artificial ant problem with the Sante Fe trail (see Figure 1(b)), which has more gaps and turns between food pellets. In his approach, the control program has a form of S-expression (LISP) including a sequence of actions and conditional statements.

Many evolutionary approaches related to the artificial ant problem considered the fitness as the amount of food that

the ant has eaten within a fixed number of time steps [7, 13, 1, 23]. The approaches suggested a new design of control structures to solve the problem, and showed the best performance that they can achieve. The artificial ant problem is an agent problem that needs internal memory to record the past sensory readings or motor actions. However, there has been little discussion for the intrinsic properties related to memory to solve the problem, although the control structures studied so far have a representation of internal memory. Agent problems in a grid world have been tackled with a variety of control structures [13, 15, 23, 8, 9], but there has been little study to compare control structures for memory effect.

Internal memory is an essential component in agent problems in a non-Markovian environment [19, 15, 11]. Agents often experience a perceptual aliasing problem [1] in non-Markovian environment. For instance, in the artificial ant problem an ant agent has two sensor states with one sensor, food detected or not in the front, but it needs different motor actions on the same sensor state, depending on the environmental feature. Thus, a memoryless reactive approach is not a feasible solution for the problem. There have been memory-encoding approaches to solve agent problems or robotic tasks in a non-Markovian environment or partially observable Markov decision process (POMPDP). Lanzi [16] has shown that internal memory can be used by adaptive agents with reinforcement learning, when perceptions are aliased. Also there have been researches using a finite-size window of current and past observations and actions [19, 18]. Stochastic approaches or reinforcement learning with finite state controllers have been applied to POMDPs [20, 4, 22]. Bakker and de Jong [3] proposed a means of counting the number of internal states required to perform a particular task in an environment. They estimated state counts from finite state machine controllers to measure the complexity of agents and environments. They initially trained Elman networks by reinforcement learning and then extracted finite state automata from the recurrent neural networks. As an alternative memory-based controller, a rule-based state machine was applied to robotic tasks to see the memory effect [10]. Kim and Hallam [11] suggested an evolutionary multiobjective optimization method over finite state machines to estimate the amount of memory needed for a goal-search problem.

Generally, finding an optimal memory encoding with the best behaviour performance in non-Markovian environments is not a trivial problem. To solve the artificial ant problem, we will follow the evolutionary robotics approach. In the evolutionary approach, the behaviour performance of an ant agent is scored as fitness and then the evolutionary search algorithm with crossover and mutation operators tries to find the best control mapping from sensor readings to actions with a given memory structure. Here, we focus on the question of how many memory states are required to solve the artificial ant problem in non-Markovian environment or what is the best performance with each level of memory amount. This issue will be addressed with a statistical analysis of fitness distribution.

In this paper, we introduce a method of quantitative comparisons among control structures, based on the behaviour performances. Then we compare two different control structures, genetic programming controllers and finite state machines. To discriminate the performances of a varying number of memory states, we provide a statistical significance analysis over the fitness samples.

[1]When the environmental features are not immediately observable or only partial information about the environment is available to an agent, the agent needs different actions with the same perceived situation.

```
(if-food-ahead (move)
    (progn3 (left)
            (progn2 (if-food-ahead (move)
                                   (right))
                    (progn2 (right)
                            (progn2 (left) (right))))
            (progn2 (if-food-ahead (move)
                                   (left))
                    (move))))
```

Figure 2: Control strategy for Santa Fe trail by Koza's genetic programming (reprinted from [20])

2. MEMORY-ENCODING STRUCTURES

2.1 Genetic programming approach

Koza [13] introduced a genetic programming approach to solve the artificial ant problem. The control structure follows an S-expression as shown in Figure 2. The ant problem has one sensor to look around the environment, and the sensor information is encoded in a conditional statement `if-food-ahead`. The statement has two conditional branches depending on whether or not there is a food ahead. The `progn` function connects an unconditional sequence of steps. For instance, the S-expression `(progn2 left move)` directs the artificial ant to turn left and then move forward in sequence, regardless of sensor readings. The `progn` function in the genetic program corresponds to a sequence of states in a finite automaton.

In Koza's approach, a fitness measure for evolving controllers was defined as the amount of food ranging from 0 to 89, traversed within a given time limit. An evolved genetic program did not have any numeric coding, but instead a combination of conditional statements and motor actions were represented in an S-expression tree without any explicit state specification. The evaluation of the S-expression is repeated if there is additional time available. Figure 2 is one of the best control strategies found [13].

Following Koza's genetic programming approach, we will evolve S-expression controllers for the Sante Fe trail in the experiments. Here, we use only two functions, `if-food-ahead` and `progn2` to restrict evolving trees into binary trees, and `progn3` can be built with a combination of the primitive function `progn2`. In the evolutionary experiments, the number of terminal nodes in an S-expression tree will be taken as a control parameter. As a result, we can see the effect of a varying number of leaf nodes, that is, a variable-length sequence of motor actions depending on the input condition. Later we will explain how the control parameter is related with the amount of memory that the S-expression tree uses.

2.2 Finite state machines

A simple model of memory-based systems is a Boolean circuit with flip/flop delay elements. A Boolean circuit network with internal memory is equivalent to a finite state machine [12]. Its advantage is to model a memory-based system with

state	input 0	input 1
q_0	q_1, L	q_0, M
q_1	q_2, R	q_2, M
q_2	q_3, R	q_3, R
q_3	q_4, L	q_4, M
q_4	q_0, M	q_0, M

(a)

state	input 0	input 1
q_0	q_7, R	q_6, M
q_1	q_6, N	q_2, R
q_2	q_5, R	q_5, M
q_3	q_0, R	q_1, L
q_4	q_2, L	q_5, M
q_5	q_6, N	q_4, M
q_6	q_0, R	q_6, M
q_7	q_2, R	q_2, L

(b)

Table 1: Finite state machines for Santa Fe trail problem (input 1: food ahead, input 0: no food ahead, output set is L (turn left), R (turn right), M (move forward), N (no-operation)) (a) 405 time steps, with 5 states (b) 379 time steps, with 8 states

a well-defined number of states, and allows us to quantify memory elements by counting the number of states. The incorporation of state information helps an agent to behave better, using past information, than a pure reaction to the current sensor inputs. Finite state machines have been used in evolutionary computation to represent state information [5].

A Finite State Machine (FSM) can be considered as a type of Mealy machine model [12], so it is defined as $M = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$ where q_0 is an initial state, Q is a finite set of states, Σ is a finite set of input values, Δ is a set of multi-valued outputs, δ is a state transition function from $Q \times \Sigma$ to Q, and λ is a mapping from $Q \times \Sigma$ to Δ, where $\lambda(q, a) \in \Delta$. $\delta(q, a)$ is defined as the next state for each state q and input value a, and the output action of machine M for the input sequence $a_1, a_2, a_3, ..., a_n$ is $\lambda(q_{x_0}, a_1), \lambda(q_{x_1}, a_2), \lambda(q_{x_2}, a_3)$, ..., $\lambda(q_{x_{n-1}}, a_n)$, where q_{x_0}, q_{x_1}, $q_{x_2}, ..., q_{x_n}$ is the sequence of states such that $\delta(q_{x_k}, a_{k+1}) = q_{x_{k+1}}$ for $k = 0, .., n-1$.

FSM can be used as a quantifiable memory structure, but developing an optimal state transition mapping needs an exhaustive search and so we apply a genetic algorithm to find desirable FSM controllers for the agent problem. An FSM controller is denoted as a numeric chromosome consisting of integer strings, unlike Koza's genetic programming. The chromosome represents a state transition table in a canonical order and its initial state is 0 by default. That is, the gene coding is defined here as a sequence of the pair (state number, state output) of each sensor value in canonical order of state number. For example, the gene coding in Table 1(b) is represented as: $7R6M\ 6N2R\ 5R5M\ 0R1L\ 2L5M\ 6N4M\ 0R6M\ 2R2L$, where motor actions will also be coded numerically. A set of sensor configurations is defined for each internal state, following the Mealy machine notation. The encoding of the Mealy machine can easily represent sequential states. However, it needs an encoding of complete sensory configurations for each state and scales badly with growing machine complexity. This control structure is useful to agents with a small number of sensors, since the chromosome size in finite state machines is exponentially proportional to the number of sensors. The artificial ant has one sensor, and the Mealy machine will have a reasonable size of chromosome even for a large number of internal states.

If there is a repeated sequence of actions, (A_1, A_2, A_3, ..., A_n) to be executed, it can be run by FSM controllers with at most n internal states. Koza's genetic program has a sequence of conditioned actions and so it can be converted into a finite state automaton without difficulty. Terminal nodes in a genetic program of S-expression define motor actions of an ant agent, and the tree traversal by a depth-first-search algorithm relying on sensor readings will guide a sequence of actions. We can simply assign each action in sequence to a separate internal state, and the sequence order will specify the state transition. For example, the function `progn2` or `progn3` has a series of actions, and so the corresponding states defined for the actions will have unconditional, sequential state transitions. The function `if-food-ahead` will have a single internal state for its two actions[2] in the branches, because the actions depending on the sensor reading (input 0 or 1) can be put together in a slot of the internal state. With this procedure, we can estimate the number of internal states that a genetic program needs, as the total number of terminal nodes minus the number of `if-food-ahead`'s. The above conversion algorithm will be applied to evolved S-expression trees in the experiments and we can compare the two types of controllers, FSM and S-expression controllers, in terms of memory states. The FSM built with the algorithm may not be of minimal form in some case, because a certain state may be removed if some sequence of actions are redundant, or if nested and consecutive `if-food-ahead`'s appear in the evolved tree (some motor actions for no food may not be used at all). However, the algorithm will mostly produce a compact form of FSMs.

Table 1(a) is a FSM converted from the genetic program shown in Figure 2; the genetic programming result has a redundant expression `(progn2 (left) (right))` in the middle of the S-expression and it was not encoded into the FSM. If one looks into the controller in Table 1(a), the behaviour result is almost the same as the Koza's genetic programming result in Figure 2, even if they are of different formats. Sequential actions were represented as a set of states in the finite automaton. When we evolved FSMs such that ants collect all the food in the grid world, the controllers in Table 1(b) as well as Table 1(a) were obtained in a small number of generations. The FSM controller in Table 1(a) takes 405 time steps to pick up all 89 pellets of food, while Table 1(b) controller needs 379 time steps. As an extreme case, a random exploration in the grid would collect all the food if there is a sufficient time available. Thus, the efficiency, that is, the number of time steps needed to collect all the food can be a criterion for better controllers. In this paper, we will consider designing efficient controllers with a given memory structure and explore the relation between performance and the amount of memory.

As alternative control structures, recurrent neural networks [7, 2] and multiple interacting programs [1] have been applied to the artificial ant problem. The control structures have an advantage of representing a complex dynamic operation, but their representations are not quantifiable in terms of internal states. Especially recurrent neural networks have been popularly used in many agent problems with non-Markovian environment [21], but it would be a difficult task to identify the internal states directly or recognize the relevance and role of internal memory. In contrast, finite state machines and Koza's genetic programming can quantify the amount of internal memory needed for a given

[2]The function `if-food-ahead` is supposed to have two children nodes or two subtrees. If an action is not observed immediately at the left or right child, the first terminal node accessed by the tree traversal will be chosen for the internal state.

agent problem and allow us to analyze the role of internal states on the behaviour performance. In the experiments, we will show this quantitative approach and compare the two different control structures.

3. BETA DISTRIBUTION MODEL

If there exists a decision threshold to evaluate the performance, we can score a trial run as success or failure. In this case we can consider the success rate as a performance measure. Estimation of success rate can be achieved by empirical data, that is, we can take a finite number of trial runs, and count the number of successful runs. Then the relative frequency of success can be an estimate of the success rate. However, this measure does not reflect the total number of runs. The estimated rate may have large deviation from the real success rate when the number of runs is small. For instance, one successful run out of four trial runs should have a different analysis and meaning with 10 successful runs out of 40 runs, although the relative frequency is the same. Thus, we will explore how to estimate the success rate more accurately.

When it is assumed that $\alpha + \beta$ experiments experience α successes and β failures, the distribution of success rate p can be approximated with a beta distribution. The probability distribution of success rate can be obtained with Bayesian estimation, which is different from the maximum likelihood estimation of $p = \frac{\alpha}{\alpha+\beta}$. The beta probability density function for the success rate is given by

$$f(p, \alpha, \beta) = \frac{1}{B(\alpha+1, \beta+1)} p^{\alpha}(1-p)^{\beta}$$

where $B(\alpha+1, \beta+1) = \int_0^1 p^{\alpha}(1-p)^{\beta} dp = \frac{\Gamma(\alpha+1)\cdot\Gamma(\beta+1)}{\Gamma(\alpha+\beta+2)}$ and $\Gamma(n+1) = n\Gamma(n)$.

Now we can define confidence intervals about success/failure tests for a given strategy. Assume that a random variable X with a beta distribution has the upper bound b_u and the lower bound b_l for confidence limits such that $P(b_l \leq X \leq b_u) = 1 - \epsilon$ and $P(X < b_l) = \epsilon/2$. Then we can assert that $[b_l, b_u]$ is a $(1-\epsilon)\cdot 100$ percent confidence interval. If a success probability p is beta-distributed, the confidence limits b_l, b_u can be obtained by solving the following equations:

$$\frac{\epsilon}{2} = \int_0^{b_u} \frac{p^{\alpha}(1-p)^{\beta}}{B(\alpha+1, \beta+1)} dp, \quad (1)$$

$$\frac{\epsilon}{2} = \int_{b_l}^1 \frac{p^{\alpha}(1-p)^{\beta}}{B(\alpha+1, \beta+1)} dp \quad (2)$$

The lower and upper bound probability b_l, b_u will decide the $(1-\epsilon)\cdot 100\%$ confidence interval $[b_l, b_u]$.

Now we compute the computational effort based on success rate. We assume that a given experiment is repeatedly run until a successful controller is found. A better strategy or methodology will have a smaller number of runs to obtain a successful controller. Thus, the computational effort (computing time) needed to obtain the first success can be a criterion for performance evaluation. The effort test was suggested by Lee [17] to compare the performance of different strategies for evolutionary experiments, and he used a measure of the average computing cost needed for the first successful run. In this paper the measure will be extended more rigorously to show the confidence interval of the effort cost.

For a given success rate p over the controller test, the average number of trial runs before the first success run can be calculated as

$$E(X) = \sum_{x=1}^{\infty} xp(1-p)^x = \frac{1-p}{p}$$

where x is the number of trials before the first success. Therefore, the computational effort[3] applied before the first success will be $\frac{1-p}{p}C$ where C is a unit computing cost per run. If the computing cost per run is variable, we take the averaged cost over multiple runs for an approximate estimation of C. If a success rate p has the lower and upper bound probability b_l, b_u by the estimation of confidence interval in Equation (2), the $(1-\epsilon)\cdot 100\%$ confidence effort cost will be estimated with $[\frac{1-b_u}{b_u}C, \frac{1-b_l}{b_l}C]$.

4. EXPERIMENTS

Our evolutionary algorithm will focus on quantifying the amount of memory needed to solve the artificial ant problem. In the ant problem, the fitness function F is defined as follows:

$$F = t_A - \alpha \cdot Q_{food}$$

where t_A is the number of time steps required to find all the food, or the maximum allowed amount of time if the ant cannot eat all of them. In the experiments 400 time steps are assigned for each ant agent's exploration, and Q_{food} is the amount of food the ant has eaten. α is a scaling coefficient, which is set to 2. This fitness function considers finding the minimum amount of time to traverse all the food cells. Thus, smaller fitness means better controller. We will use the Santa Fe trail for the target environment. An ant agent may need varying computing time for its fitness evaluation, because t_A varies depending on the time to collect all the food. When an ant succeeds in collecting all the food before 400 time steps, the exploration process can instantly stop not to wait for the whole 400 time steps to complete. This will influence the computing cost and so the computing cost C for a single run will be measured by the averaged CPU run-time over multiple runs.

In this paper, we will test two types of evolving control structures, FSMs and S-expression trees, and compare the performances of ant agents for a given level of memory amount. We will evolve each control structure with a varying number of internal states and analyze fitness samples collected from the evolutionary algorithms.

For the first set of experiments with FSM controllers, the chromosome of a FSM controller is represented as an integer string as described in section 2. One crossover point is allowed only among integer loci, and the crossover rate is given to each individual chromosome, while the mutation rate is applied to every single locus in the chromosome. The mutation will change one integer value into a new random integer value. We used a tournament-based selection method of group size four. A population is subdivided into a set of groups and members in each group are randomly chosen among the population. In each group of four members, the

[3]We assume the computational effort only consists of experimental runs who result in failure. If we include the first successful run in the effort, the effort can be estimated with $\frac{1-p}{p}C_f + C_s$ where C_f is a failure computing cost and C_s is a success computing cost.

two best chromosomes[4] are first selected in a group and then they breed themselves. A crossover over a copy of two best chromosomes, followed by a mutation operator, will produce two new offspring. These new offspring replace the two worst chromosomes in a group. In the experiments, the crossover rate is set to 0.6 and the mutation rate, 2 over chromosome length, is applied (the above tournament selection takes a half of the population for elitism, and so the high mutation rate will give more chance of diversity to a new population).

For another memory-based controller, the chromosome of a genetic program is defined as an S-expression tree. The crossover operator on the program is defined as swapping subtrees of two parents. There are four mutation operators available for one chromosome. The first operator deletes a subtree and creates a new random subtree. The subtree to be replaced will be randomly selected in the tree. The second operator randomly chooses a node in the tree and then changes the function (`if-food-ahead` or `progn2`) or the motor action. This keeps the parent tree and modifies only one node. The third operator selects a branch of a subtree and reduces it into a leaf node with a random motor action. It will have the effect of removing redundant subtrees. The fourth operator chooses a leaf node and then splits it into two nodes. This will assist incremental evolution by adding a function with two action nodes. In the initialization of a population or the recombination of trees, there is a limit for the tree size. The maximally allowable depth of trees is 6 in the initialization, but there is no restriction of the depth after the recombination. The minimum number of leaf nodes is 1 and the maximum number of leaf nodes will be set up as a control parameter in the experiment. If the number of leaf nodes in a new tree exceeds the limit, the tree is mutated until the limit condition is satisfied. The above tournament-based selection of group size four will also be applied to the genetic programming approach. The crossover rate is set to 0.6 and the mutation rate is 0.2.

4.1 Evolving FSMs

To quantify the amount of memory, a varying number of states, ranging from 1 state to 20 states, are applied to the artificial ant problem. For each different size of state machines, 50 independent runs with a population size of 100 and 10,000 generations are repeated and their fitness distribution is used for the analysis of memory states. Similar evolutionary experiments are applied to the S-expression controllers with a population size of 500 and 2,000 generations[5]. To compare S-expression and FSM controllers, evolved S-expression controllers are converted into FSMs using the algorithm described in section 2.

To find out an appropriate number of states needed to reach a given performance level, experiments with a fixed number of states is repeated 50 times. We first tested the evolution of FSMs. Figure 3 shows the average fitness result for each number of memory states. The error bars of 95% confidence intervals are displayed with the average performance over 50 runs by assuming a t-distribution. The experiment clearly distinguishes the performances of memory states ranging from one to eight states, although more than eight states were not significantly different from eight states.

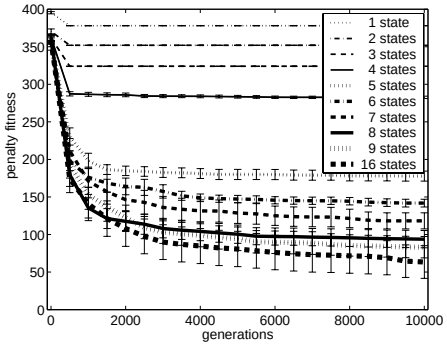


Figure 3: Fitness test experiments in the Santa Fe trail problem

According to the experiments, five memory states are required for ants to traverse all the food cells. A strategy for an ant agent to collect all the food with five internal states was to look around the environment and move forward if there is no food in the surrounding neighbors. If food is found in one of neighbor cells, the ant immediately moves forward in that direction. The plan needs internal states to take a sequence of actions (`left`, `right`, `right`, `left`, `move`) or another sequence (`right`, `left`, `left`, `right`, `move`) with a separate state for one action. If more internal states are given, ants start to utilize environmental features to reduce exploration time, for instance, ants can take three consecutive `move` actions without looking around the neighbors after a right or left turn finds food. With a large number of states, more elaborated turns and moves are found in the behaviour of ant agents.

The memory requirement for the task may be determined by the average fitness of multiple runs and its confidence interval, that is, the t statistic [10]. However, our experiments show that the fitness performance has a relatively large variance (error bar) and so the mean difference between a pair of internal states is not large enough to see the statistical significance. Langdon [14] argued that the artificial ant problem is highly deceptive, since its fitness landscape is rugged with plateaus, valleys and many local optima. Thus, we suggest a measure of success rate to evaluate fitness samples, based on the beta distribution. This statistic test can be applied to any type of fitness samples regardless of whether they are normally or skewedly distributed.

4.2 Beta distribution for memory analysis

Now we apply the beta distribution analysis to the fitness samples for FSMs with a varying number of internal states. We assume that if there exists any decision threshold of fitness for success/failure for a given pair of controllers such that it causes significant difference of performance, then the control structures are distinguishable in performance. Between seven and eight states, we place a decision threshold of fitness 90 to determine their success/failure. Then we obtain 5 successes among 50 trials for seven internal states and 27 successes for eight states. If we

[4]More than two chromosomes may have tie rank scores and in this case chromosomes will be randomly selected among the individuals.

[5]This parameter setting showed good performance within the limit of 5×10^5 evaluations. This may be due to the fact that genetic programming tends to develop new good offspring through crossover of individuals in a large sized population rather than with the mutation operator [21].

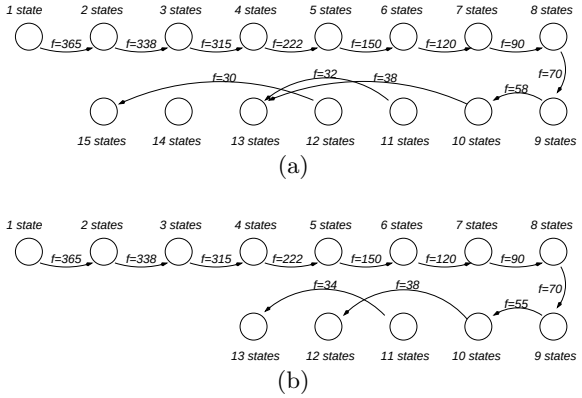


Figure 4: A partial order relation of FSMs with varying number of memory states in terms of behaviour performance (by beta distribution) (a) 50 trials (b) 25 trials (the arrow label indicates the threshold fitness for success/failure)

calculate the confidence interval of success rate in beta distribution, seven states and eight states have success rates, [0.044, 0.214] and [0.403 0.671], respectively, and their effort costs will be $[\frac{1-0.214}{0.214}C_7, \frac{1-0.044}{0.044}C_7] = [3.67C_7, 21.73C_7]$ and $[\frac{1-0.671}{0.671}C_8, \frac{1-0.403}{0.403}C_8] = [0.49C_8, 1.48C_8]$, where C_7, C_8 is the average computing cost for a single run in failure mode, that is, an experimental run which does not reach the fitness 90. Here, it is assumed that the cost C_7, C_8 is almost identical for the two types of state machines. Then the confidence intervals are significantly different and thus we can clearly see the performance difference of the two control structures. In fact, the computing cost for each run is variable by different exploration time. In the evolutionary experiments the averaged CPU run-time over multiple runs resulting in failure was $\bar{C}_{f7} = 39.69$ sec, $\bar{C}_{f8} = 41.96$ sec for seven and eight states, respectively, while the averaged CPU run-time for success was $\bar{C}_{s7} = 38.20$ sec, $\bar{C}_{s8} = 38.36$ sec. The difference of the computing costs does not influence the above significance analysis, because the ratio of the costs is close to 1.

By the analysis on the experiments with FSMs, we found that finite states ranging from one to ten have distinctive difference in performance. There was no significant difference between 10 states and 11 states, or between 11 states and 12 states. However, FSMs with 13 states showed significantly better performance than FSMs with 10 states. Figure 4(a) shows a partial order relation of FSMs with varying number of states, which was estimated by the confidence interval of success rate or effort cost. Evolving more than 15 states produced slightly better controllers than evolving 15 states, but the difference among the fitness samples was not significant. In the same procedure of memory analysis, we tested the statistical significance with a smaller number of trials, 25 trials. We still obtained similar partial order relation among a varying number of states, although the performance difference between 12 states and 15 states became insignificant. It implies that the beta distribution model can be used for performance comparison even with a small number of trials. The above results show that the beta distribution analysis can find significance of performance differences where fitness samples have a large variance due to outliers or where

```
(if-food-ahead
     (move)
     (progn2 (right)
             (progn2
                (if-food-ahead
                   (progn2
                       (progn2 (move) (move))
                       (move))
                   (progn2
                     (progn2 (left) (left))
                     (if-food-ahead
                        (move) (right))))
                (if-food-ahead
                    (progn2 (move) (move))
                    (move)))))
```

(a)

state	no food ahead	food ahead
q_0	q_1, R	q_0, M
q_1	q_4, L	q_2, M
q_2	q_3, M	q_3, M
q_3	q_6, M	q_6, M
q_4	q_5, L	q_5, L
q_5	q_6, R	q_6, M
q_6	q_0, M	q_7, M
q_7	q_0, M	q_0, M

(b)

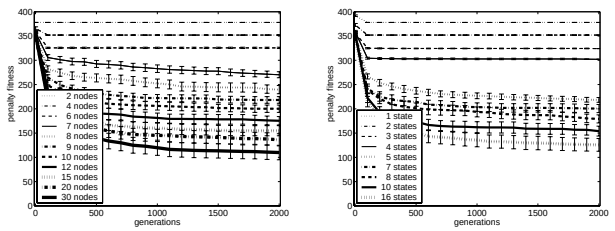
Figure 5: An example of genetic programming result (a) evolved S-expression ($n_t = 12,\ n_f = 4$) (b) converted FSM

t statistic does not produce useful information. The result indirectly supports the validity and usefulness of the beta distribution. We also applied Wilcoxon rank-sum test, and similar partial order relations were obtained.

4.3 Genetic programming approach

For another type of memory-based controllers, we used S-expression controllers. Let n_t, n_f, n_s the number of terminal nodes, the number of the function `if-food-ahead`'s, and the number of internal states in an S-expression tree, respectively. By the conversion algorithm described in section 2, we can build an FSM such that the number of states is $n_s = n_t - n_f$. Figure 5 shows a conversion example for an evolved S-expression tree. The states $\{q_0, q_1, q_5, q_6\}$ in Figure 5(b) have different state transitions or motor actions on the input condition and they correspond to the operation of the function `if-food-ahead`. The other states define the same transition and action on any input condition, which is a copy operation of `progn2`. The conversion algorithm produces a compact form of FSM and helps tracing internal states that the genetic program uses. In this paper we use two control parameters, n_t, n_s to evolve S-expression trees. Once a control parameter is set up for the evolutionary experiments, for instance, the number of terminal nodes is defined, the size of evolved trees should be within the limit size. If an evolved tree is over the limit, the tree should be re-generated by mutation to satisfy the condition.

In the first experiment, a varying number of terminal nodes were used as a control parameter. A large number of terminal nodes can expect more sequence of actions and thus produce better performance. Figure 6(a) shows the average fitness performance with its 95% confidence range by t statistic for a given number of terminal nodes. Evolving 20 terminal nodes and 30 terminal nodes had 3, 22 cases to reach the fitness 90 or below, respectively, which are similar



(a) (b)

Figure 6: Genetic programming result (a) evolve controllers with a fixed number of terminal nodes (b) evolve controllers with a fixed number of states

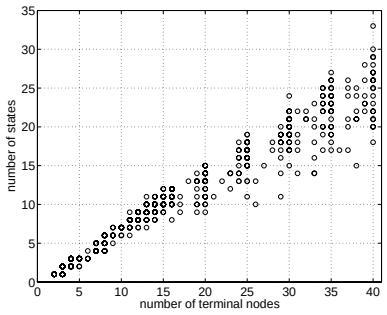


Figure 7: Relation between terminal nodes and internal states in genetic programming approach (a varying number of terminal nodes was a control parameter)

to the performance of 7 states (5 successes) and 8 states (27 successes) among the FSM controllers in Figure 3. However, it is not easy to compare directly the performances of FSM and S-expression controllers, since they should have the same criterion for comparison. The best evolved tree for each run often follows the rule that the number of terminal nodes can be approximated by 1.5 times the number of states (see Figure 7), when the tree is converted into the corresponding FSM structure and the internal states are counted. The best trees with 20 terminal nodes had a range of 9-15 states, and the performance was significantly lower than the performances of FSM controllers evolved with the same range of states. It indirectly entails that evolving FSMs can provide a better solution to encode internal memory.

For the next experiment, we set up the number of states as a control parameter. In a similar process as above, we test a varying number of states. If an evolved tree includes more states than the limit, the tree will be mutated until the limit condition is satisfied. In addition, we set the maximum number of terminal nodes for evolving trees to $1.7n_s$ for each number of states, n_s, using the above relation rule between terminal nodes and states (Figure 7). Without the restriction on terminal nodes, the evolving trees encountered bloat, degrading the performance. As shown in Figure 6(b), the internal states play a critical role on the performance improvement, and the fitness performance is enhanced with more internal states. However, for a given number of states, the genetic programming result is mostly worse than the FSM evolution result shown in Figure 3. The best fitness that the genetic programming achieved for the overall experiments was 44, and it had a similar performance with

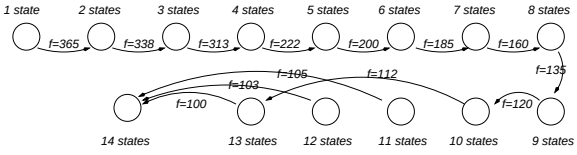


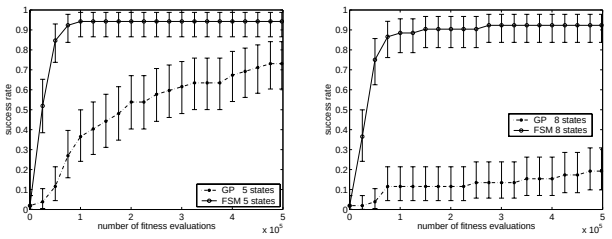
Figure 8: A partial order relation of S-expression trees with varying number of states in terms of behaviour performance (the arrow label indicates the threshold fitness for success/failure)

the FSM evolution only for a few sets of internal states. Evolving directly the FSM structure tends to produce more efficient controllers and its performance level is significantly better. The S-expression is a procedural program arranging a sequence of actions, while FSMs can not only encode a sequence of actions, but also they have more dynamic features in representation by allowing flexible transitions and actions from state to state.

By the fitness distribution of genetic programming controllers, we built a partial order relation among memory structure as shown in Figure 8. When the diagram is compared with the partial order graph by the FSM result (Figure 4), the threshold level for the relation is changed, but roughly keeps the relation structure. It seems that a large number of states have more variation on the relation result. As a matter of fact, the lattice diagram for memory analysis is extracted from empirical data which depends on the corresponding control structure. It only shows the partial order information among memory structure with a given confidence level, but it does not mean that the corresponding structure guarantees a given level of performance or it cannot achieve better performance than the threshold level. More trial runs will support more reliable information of the performance level or the partial order relation.

So far we showed the distribution of success rate or computational effort to measure the performance difference between a pair of sample groups. The significance test result of the success rate is equivalent to that of the effort cost if almost the same computing cost is spent for each evolutionary run. At this point the analysis of the effort cost is not more helpful than the analysis of success rate. However, the information of computational effort can be used to understand the evolutionary process for a given problem. To reach the fitness level 150, FSM controllers obtained 17 successes with 6 internal states and 10^5 evaluations (2,000 generations). More generations, for example, 5,000 generations and 10,000 generations produced more successes as expected. The unit cost C_2, C_5, C_{10} is the computing cost for $10^5, 2.5\times10^5, 5\times10^5$ evaluations and the costs can be approximated with $C_{10} = 2C_5 = 5C_2$. Then the confidence intervals for $10^5, 2.5\times10^5, 5\times10^5$ evaluations (2,000, 5,000 and 10,000 generations) become $[1.09C_2, 3.46C_2]$, $[1.69C_2, 5.09C_2]$, $[1.74C_2, 5.40C_2]$, respectively. Then the experiments with 2,000 generations have the confidence interval with the smallest effort level, whose range is also narrow. Thus, we can recommend 2,000 generations for the future experiments.

We can compare the computational efforts for different types of controllers or different algorithms. The analysis of the effort cost will be useful especially when the algorithms have different CPU run-time for an evolutionary run. For instance, the FSM controller with eight states and 10^5 evalu-



(a) (b)

Figure 9: Run-time distribution with genetic programming and FSM controllers (the average and 95% confidence range of success rate in the beta distribution are displayed) (a) evolving five internal states (threshold fitness: 222) (b) evolving eight internal states (threshold fitness: 150)

ations (2,000 generations) has a confidence interval $[0.05C_2, 0.27C_2]$ to obtain the fitness 150, while the genetic programming for eight states needs a computing cost $[3.67K_{0.4}, 21.51K_{0.4}]$ to obtain the first success with 95% chance. Indeed, the ratio of the average CPU run-time between the FSM and the genetic programming ($K_{0.4}/C_2$) was 1.23 in the experiments. The comparison result implies that the FSM controllers produce more efficient results than the genetic programming controllers. This was confirmed again in other pairwise tests for small fitness.

Hoos and Stützle [6] studied a run-time distribution to compare different algorithms or determine parameter settings. The distribution shows the empirical success rate depending on varying run time. In our experiments, the number of trial runs is relatively small and so the beta distribution of success rate was applied to the run-time distribution as shown in Figure 9, where the number of fitness evaluations was used instead of the actual CPU run-time. The FSMs with five and eight states showed significantly better performance in the run-time process. Generally, FSMs show more discriminative performance for a large number of states.

5. CONCLUSION

In this paper, we applied two types of memory-encoding controllers, genetic programming controllers (tree structure) and finite state machines to the artificial ant task which has a perceptual aliasing problem. We explored a systematic analysis over the usage of internal memory, based on statistical significance test, and built a partial order relation of memory states needed to reach each level of performance. The significance test with success rate or computational effort shows that FSMs have more powerful representation to encode internal memory and produce more efficient controllers than the tree structure, while the genetic programming code is easy to understand. Using the suggested approach, we can identify the role of internal states or observe the relevance of memory to agent behaviours.

6. ACKNOWLEDGMENT

The author would like to thank Dr. John Hallam for introducing the effort test based on beta distribution.

7. REFERENCES

[1] P. Angeline. Multiple interacting programs: A representaiom for evolving complex behaviors. *Cybernetics and Systems*, 29(8):779–806, 1998.

[2] P. Angeline, G. Saunders, and J. Pollack. An evolutionary algorithm that constructs recurrent neural networks. *IEEE Trans. on Neural Networks*, 5(1):54–65, 1994.

[3] B. Bakker and M. de Jong. The epsilon state count. In *From Animals to Animats 6: Proc. of Conf. on Simulation of Adaptive Behaviour*, pages 51–60. MIT Press, 2000.

[4] D. Braziunas and C. Boutilier. Stochastic local search for POMDP controllers. In *AAAI*, pages 690–696, 2004.

[5] L. Fogel, A. Owens, and M. Walsh. *Artificial intelligence through simulated evolution*. Wiley, New York, 1966.

[6] H. Hoos and T. Stützle. Evaluating Las Vegas algorithms - pitfalls and remedies. In *Proc. of Conf. on UAI*, pages 238–245. Morgan Kaufmann, 1998.

[7] D. Jefferson, R. Collins, C. Cooper, M. Dyer, M. Flowers, R. Korf, C. Taylor, and A. Wang. Evolution as a theme in artificial life. In C. Langton, editor, *Artificial Life II*. Addison Wesley, 1991.

[8] D. Kim. Analyzing sensor states and internal states in the Tartarus problem with tree state machines. In *Parellel Problem Solving From Nature 8, Lecture Notes on Computer Science vol. 3242*, pages 551–560, 2004.

[9] D. Kim. Evolving internal memory for T-maze tasks in noisy environments. *Connection Science*, 16(3):183–210, 2004.

[10] D. Kim and J. Hallam. Mobile robot control based on Boolean logic with internal memory. In *Advances in Artificial Life, Lecture Notes in Computer Science vol. 2159*, pages 529–538, 2001.

[11] D. Kim and J. Hallam. An evolutionary approach to quantify internal states needed for the woods problem. In *From Animals to Animats 7, Proc. of Int. Conf. on the Simulation of Adaptive Behavior*, pages 312–322. MIT Press, 2002.

[12] Z. Kohavi. *Switching and Finite Automata Theory*. McGraw-Hill, New York, London, 1970.

[13] J. R. Koza. *Genetic Programming*. MIT Press, Cambridge, MA, 1992.

[14] W. Langdon and R. Poli. Why ants are hard. In *Proceedings of Genetic Programming*, 1998.

[15] P. Lanzi. An analysis of the memory mechanism of XCSM. In *Genetic Programming 98*, pages 643–651. Morgan Kauffman, 1998.

[16] P. Lanzi. Adaptive agents with reinforcement learning and internal memory. In *From Animals to Animats 6: Proc. of Conf. on Simulation of Adaptive Behaviour*, pages 333–342. MIT Press, 2000.

[17] W.-P. Lee. *Applying Genetic Programming to Evolve Behavior Primitives and Arbitrators for Mobile Robots*. Ph. D. dissertation, University of Edinburgh, 1998.

[18] L. Lin and T. M. Mitchell. Reinforcement learning with hidden states. In *From Animals to Animats 2: Proc. of Conf. on Simulation of Adaptive Behaviour*, pages 271–280. MIT Press, 1992.

[19] A. McCallum. *Reinforcemnet Learning with Selective Perception and Hidden State*. Ph. D. dissertation, University of Rochester, 1996.

[20] N. Meuleau, L. Peshkin, K.-E. Kim, and L. P. Kaelbling. Learning finite-state controllers for partially observable environments. In *Proc. of the Conf. on UAI*, pages 427–436, 1999.

[21] S. Nolfi and D. Floreano. *Evolutionary Robotics*. MIT Press, Cambridge, MA, 2000.

[22] L. Peshkin and N. M. ad L. P. Kaebling. Learning policies with external memory. In *Proc. of Int Conf. on Machine Learning*, pages 307–314, 1999.

[23] A. Silva, A. Neves, and E. Costa. Genetically programming networks to evolve memory mechanism. In *Proceedings of Genetic and Evolutionary Computation Conference*, 1999.

[24] A. Teller. The evolution of mental models. In *Advances in Genetic Programming*. MIT Press, 1994.

[25] S. W. Wilson. ZCS: A zeroth level classifier system. *Evolutionary Computation*, 2(1):1–18, 1994.