Controlling Effective Introns for Multi-Agent Learning by Genetic Programming

Hitoshi Iba
Dept. of Frontier Informatics,
School of Frontier Science,
The University of Tokyo
iba@miv.t.u-tokyo.ac.jp

Makoto Terao
Dept. of Inf. and Comm. Eng.,
School of Engineering,
The University of Tokyo
terao@miv.t.u-tokyo.ac.jp

Abstract

This paper presents the emergence of the cooperative behavior for multiple agents by means of Genetic Programming (GP). For the purpose of evolving the effective cooperative behavior, we propose a controlling strategy of introns, which are non-executed code segments dependent upon the situation. The traditional approach to removing introns was able to cope with only a part of syntactically defined introns, which excluded other frequent types of introns. The validness of our approach is discussed with comparative experiments with robot simulation tasks, i.e., a navigation problem and an escape problem.

1 Introduction

Recently intelligent agents and multi-agent systems have attracted much interest in Distributed Artificial Intelligence (DAI). GP and its variants have been applied to the multi-agent learning (see [Haynes *et al.*95], [Luke *et al.*96], [Iba96], [Hara *et al.*99] for example). However, in the multi-agent application of GP, the computational burden is often problematic. This is because the number of GP trees required for the multi-agent task becomes larger with the number of agents. For instance, in the heterogeneous breeding strategy (see Section 2 for details), each agent uses a distinct GP program so that the total number of GP trees is N times as great as that of a single-agent task, where N is the number of agents.

Programs generated by GP grow very quickly in size and include large amount of non-functional codes, i.e., introns. This "bloating" effect degrades the GP search, in the sense that (1) larger programs often require more time and more space to run, and (2) larger programs tend to show worse generalization performance than shorter ones. Thus, reducing introns is very important in the GP search. The bloating is supposed to have a much worse effect on the multi-agent application because of a greater number of GP tress required for the task. [Soule *et al.*96] showed that the exponential growth was mainly dominated by non-functional codes. He used a controlling method of the tree growth by removing non-functional code segments, and demonstrated its effectiveness in bounding the programs' size. However, the deletion of all non-functional code segments is known to be an insolvable problem. It is reducible to the program equivalence problem, which is non-recursive. Thus, he only removed a part of syntactic introns that are known to be non-executed before the execution. These introns exclude other frequent types of introns (see Section 3 for details).

This paper introduces the concept of "effective introns", i.e., non-functional code segments of a GP tree dependent upon the execution, and proposes a controlling scheme of the tree growth for multi-agent GP learning. Based on the empirical studies, we show the effectiveness of our approach in the following points:

1. The fitness transition is improved during the training phase.
2. The code growth is effectively controlled.
3. The robustness of acquired programs is increased.

The rest of this paper is structured as follows. Section 2 describes the experimental setting for GP learning. Section 3 introduces a basic idea of controlling effective introns. Section 4 shows the experimental results with robot tasks. The performance is compared with traditional GP strategies. Section 5 discusses our approach, followed by some conclusion in Section 6.

2 Experimental Domains and GP Setting

In this paper, we use the following tasks for autonomous robots (see Fig.1). The world is a continuous 2-dimensional area on which there exist agents (i.e., robots) and obstacles. An agent is represented by an alphabet which is able to move in any direction. In our simulation, robots are supposed to be equipped with different sensors and motors. Thus, the appropriate job separation is required to solve efficiently the tasks described below.

The first task we have chosen is the robot navigation. Some of the training maps are shown in Fig.1(a) and (b), in which four robot agents are represented as a, b, c, and d. There are obstacles such as circles (a) or walls (b). The surrounding circle around an agent represents its view area. The destination of an agent is illustrated as an arrow. For instance, in Fig.1(a), the goal position of agent a is the start position of agent b. The agents' goal is to find the optimal path from given starting locations to their respective goals, while avoiding the obstacles and other robots.

The second task is an "escape problem", in which robot agents are supposed to leave a room from a door (or hole) in case of emergency, such as a fire (Fig.1(c)). However, in order to open the door (shown as □ in the figure), they have to push all the buttons (shown as +). Thus, this task consists of (1) pushing buttons and (2) escaping from the room into the door.

In order to apply GP to evolving agents' programs for the above tasks, we use the terminal and nonterminal sets shown in Table 1. In the table, a symbol without any argument is a terminal symbol. We have chosen a vector operation for the GP tree representation. This is aimed at incorporating more precise directional information as to the environment surrounding the agents. The output vector of a GP tree tells the agent how to move next, i.e., the robot moves forward in the vector's direction with the speed proportional to the vector's length unless it bumps into some obstacle. The robot commands are taken from a real robot, or can be constructed easily with the primitive commands (see [Ito *et al.*96] for details). We assume agents, i.e., robots, have an eye sensor with some limited view area. This area is shown as a circle in Fig.1. If the nearest agent is out of its scope, then the Nearest_Agent terminal returns a zero vector. It is also the case with the if_obstacle function.

There have been different breeding strategies proposed for the multi-agent learning by GP (see [Luke *et al.*96], [Iba96] and [Hara *et al.*99] for details). This paper uses the co-evolutionary breeding strategy, in which GP individuals are divided into a set of agent-type subpopulations (see Fig.2). Breeding is performed in the same way as in a distributed GP. As generations proceed, some individuals are expected to perform specialized tasks for different agents. We evaluate the fitness of individuals in an agent-type subpopulation as follows: Initially, i.e., at the first generation, the other agents' programs are chosen randomly. At later generations, we choose as the other agents' program the best programs evolved so far in the other agent-type subpopulations. In our previous papers [Iba96],[Iba98], we have empirically shown the superiority of the co-evolutionary breeding over the traditional strategies, such as the homogeneous breeding[1] and the heterogeneous breeding[2] .

The primitive behaviors, such as avoiding obstacles or searching for the goal, are supposed to be common among different agents. These building blocks can be evolved jointly. Therefore, we allow the migration of elite individuals between the agent-type subpopulation. The migration is expected to promote the gene exchange and result in the further improvement of the performance, which will be seen in later experiments.

3 Controlling Effective Introns

Syntactic introns in a GP tree are program code segments that are not reached and non-executed. For instance, in the codes (if true A B) and (or true X), both B and X are not executed. [Soule *et al.*96] used a method of replacing a nested non-functional code by a non-operational code. Consider the following codes:

```
(if A B  (while A C D))
(if_lte A B (if_lte A B E F) G)
```

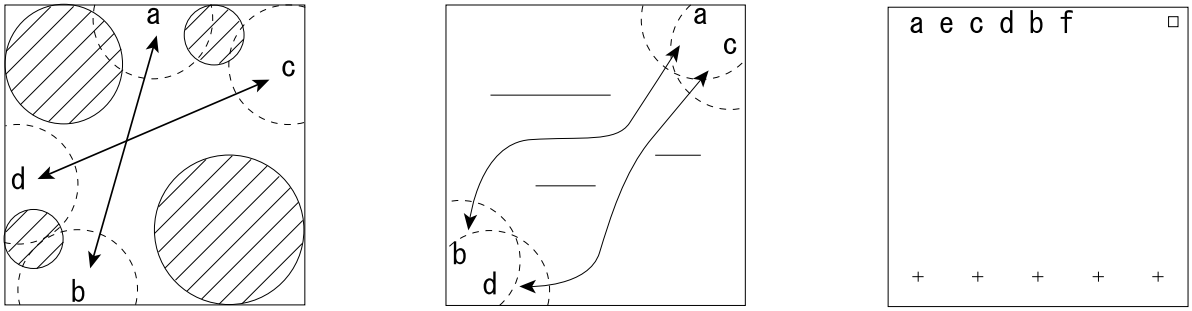
where A, B, C, etc. are any (list of) statements. In the first code, the second argument of the "if" statement, i.e., (while A C D), is never executed, because it is only executed if A is false. In the second code, the statement F is never executed because the "if_lte" condition is satisfied only when A is less than or equal to B. However, it is not easy to check these introns syntactically. It is known to be reducible to the problem equivalence problem and non-recursive.

There is another type of introns. Consider the following code:

```
(if_obstacle A B C)
```

[1] In the homogeneous breeding strategy, all agents use the same program evolved by GP.

[2] Each agent uses a distinct program in the heterogeneous strategy.



(a) (b) (c)

Figure 1: Training Maps for Robot Tasks

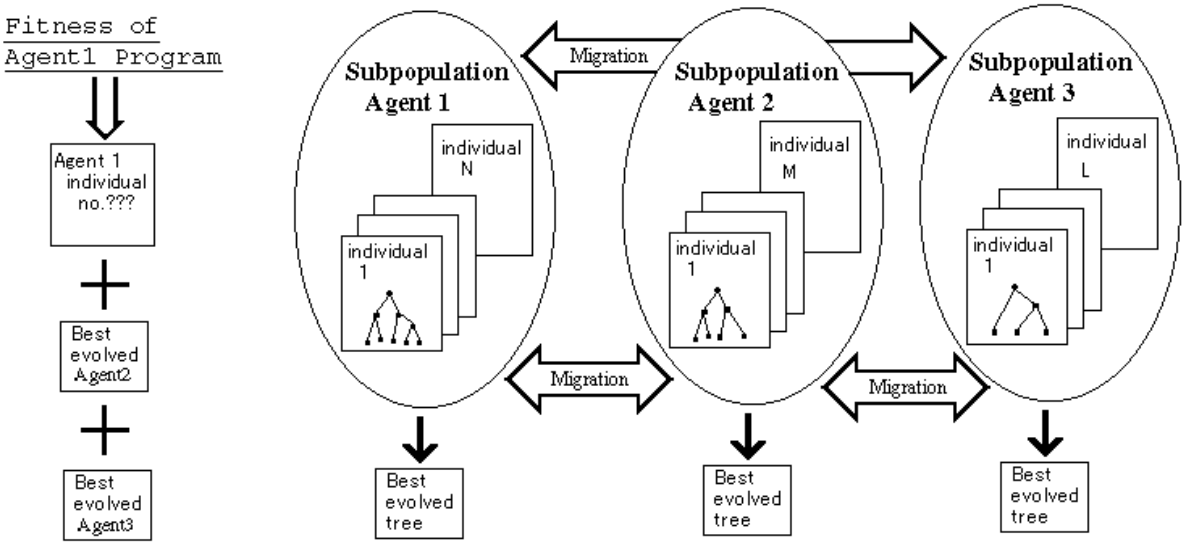


Figure 2: Co-evolutionary Breeding for Multi-Agent Learning

The statement A in the above code is never executed if the agent is in a room without any obstacles (see Table 1). Thus, the execution of these statements A, B and C is dependent upon the situation. We have often seen this type of introns appear in the evolved GP codes for the robot task.

We introduce the concept of "effective" introns, i.e., non-executed code segments upon the execution. For the sake of the identification, we attach an execution counter to each terminal or function symbol in a GP tree. This counter is set to be zero at the outset of the execution. During the evaluation of the tree, the counter is incremented when its symbol is evaluated. For instance, consider the above-mentioned code segments again:

```
(if A[0] B[0] (while A[0] C[0] D[0]))
(if_lte A[0] B[0]
    (if_lte(A[0] B[0] E[0] F[0]) G[0])
```

where the number in the bracket represents the correspondent counter. After the evaluation, suppose that we get the following code with counters incremented:

```
(if A[30] B[30] (while A[0] C[0] D[0]))
(if_lte A[34] B[34]
   (if_lte A[34] B[34] E[34] F[0]) G[0])
```

The symbol whose counter remains zero is regarded as an effective intron, which can be removed by an edit operator. For instance, the statement (while A C D) in the first code and the symbols F and G in the second code can be removed. Note that the concept of effective introns subsumes that of syntactic introns. Moreover, its definition is dependent upon the evaluating situation. For example, if the symbols A and B in the second code are evaluated dependently upon the situation, i.e., if in some cases A is greater than B,

Table 1: GP Terminals and Functions.

Name	#Args.	Description
Destination	0	The directional vector by which to move the agent toward its goal.
Last	0	The last vector of the GP output for the agent. If this is the first move, then returns a zero vector.
Nearest_Agent	0	The directional vector by which to move the agent toward the nearest agent.
Rand	0	A random vector.
+	2	Add two vectors.
–	2	Subtract two vectors.
∗2	1	Multiply the magnitude of a vector by 2.
/2	1	Divide a vector by 2.
<-45	1	Rotate a vector counterclockwise 45 degrees.
->45	1	Rotate a vector clockwise 45 degrees.
inv	1	Invert a vector, i.e., if the input is v, then return $-v$.
if_dot	4	If their dot product is greater than 0, then evaluate and return the third argument, else evaluate and return the fourth argument.
if_lte	4	If the magnitude of the first argument is greater than the magnitude of the second argument, then evaluate and return the third argument, else evaluate and return the fourth argument.
if_right	5	If the first argument is in the right side to the second argument, then evaluate the third argument. Else if the first is in the left side to the second, then evaluate the fourth. Else evaluate the fifth argument.
if_crash_wall	2	If the agent bumped into the wall in the last motion, then evaluate the first argument, else evaluate the second argument.
if_crash_agent	2	If the agent bumped into another agent in the last motion, then evaluate the first argument, else evaluate the second argument.
if_obstacle	3	If there exists a wall between the agent and the destination, then evaluate the first argument, Else if there is another agent between them, then evaluate the second. Else evaluate the third argument.

and in other cases, A is smaller, then we will have the following results:

```
(if_lte A[34] B[34]
   (if_lte A[24] B[24] E[24] F[0]) G[10])
```

where A is greater than B in 24 cases out of 34 cases. In this situation, G is not an intron.

[Smith *et al.*96] discussed the analysis of introns and gave a useful taxonomy of them. Effective introns in our paper correspond to their types 1, 2 and 4. We try to extend their previous researches and establish a controlling method of the above-mentioned effective introns. The next section describes experimental results to show the effectiveness of our approach.

4 Experimental Results

This section explains the experimental results with the robot tasks described in Section 2. GP parameters we have chosen are as follows: Population size = 512, Maximum generations = 50, and Tournament selection method.

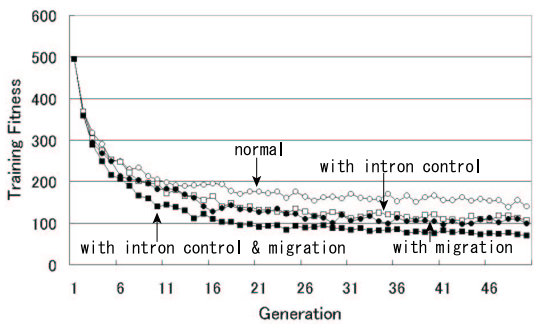
4.1 Navigation Problem

We used six different maps for the training. Some of the training maps are shown in Fig.1(a) and (b). The initial positions of agents are changed at every generation for the training data. The fitness (F) is defined in the following way. In general, the faster the task is finished, the better, i.e., the smaller, the fitness is.

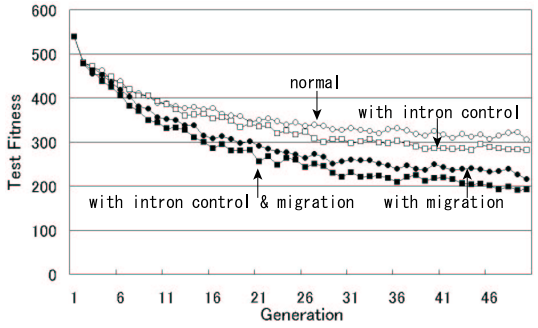
$$F := \begin{cases} 100.0 - 3 \times (\text{Remaining Times}) & \text{Success} \\ 300.0 + \sum_{i=1}^{\#\text{ agents}} \mathrm{D}(Loc_i, Dest_i) & \text{Failure} \end{cases} \quad (1)$$

The GP program of a robot agent is evaluated for a limited number of time steps, i.e., 40 to 50 time steps. If all agents have reached their destination during the evaluation, i.e., the task is completed, then the fitness is reduced with a bonus proportional to the remaining time steps. When the task has not been finished, the fitness is added with a penalty. The penalty is dependent upon the distance between the current agent location and its destination, i.e., $\mathrm{D}(Loc_i, Dest_i)$. The actual fitness of a GP program is the averaged value of the above fitness over all the training maps.

Fig.3(a) shows the experimental result, which plots the fitness values with generations. We compare four different types of GP runs, i.e., GP with/without controlling effective introns, and with/without the migration. The data are averaged over 50 runs. Note that the fitness value of 100.0 is considered as the task complete level. For standard GP runs, the final fitness value was about 140, which is well over the task complete level



(a) Training Cases.



(b) Test Cases.

Figure 3: Experimental Results (Generation vs. Fitness)

		Controlling Effective Introns	
		×	◯
Migration	×	10	23
	◯	26	47

Table 2: Numbers of Successful Runs

(see the line labeled as "normal"). In fact, in only ten cases out of 50 runs, all four robots were evolved to reach their respective goals. On the other hand, if we controlled the effective introns, the better performance was obtained. When we used the migration as well, the performance was further improved. Table.2 compares the numbers of successful cases out of 50 runs.

Fig.4 shows the example behavior of an evolved robot. As can be seen in the figure, some robots behaved in a different way from others. For instance, the robot agent marked as an arrow dashed to its goal, i.e., it never gave way even when it came across other agents. Another agent always gave way to other agents. Thus, we can observe that the appropriate job separation has been established for this navigation task.

Fig.5(a) shows the averaged sizes of the best programs with generations. Fig.5(b) plots the intron ratios. We can confirm that the effective introns were successfully removed by the proposed method, which we believe leads to the above-mentioned improvement.

The robustness is an important feature of a program evolved by GP [Ito *et al.*96]. It is defined as the ability to cope with noisy or unknown situations. In the robot navigation, the robustness could be examined by testing an evolved program for another navigation task. In pursuit of the robustness, we verified the validness of an evolved program for testing data, which were different from the training data. We used three testing maps. The initial 100 positions were randomly generated every time for the sake of testing the generalization performance. Thus, 300 cases in total were tested for the validation. The result is shown in Fig.3(b). As can be seen from the figure, we can confirm the effectiveness of the removing method of effective introns, in terms of the robustness.

4.2 Escape Problem

We used four different maps for the training. One of the training maps is shown in Fig.1(c), in which six robot agents are represented as a,b,c etc. The maximum speed of robots are ranked as $a = b < c = d < e = f$. That is, robots e and f move faster than the others. For this task, we introduced a special terminal for identifying the button, i.e., Nearest_Button. We have again confirmed the effectiveness of controlling effective introns with this more complicated task (see Fig.6). Fig.7 shows the experimental results, i.e., the example behavior of acquired robot programs. We can observe that faster agents, i.e., e and f in Fig.7(a), bothered to push buttons in spite of being late. In another case, the nearest robots carried out the duty to push buttons (Fig.7(c)). In this way, robot agents have established an effective job separation dependent upon the situation, and seem to achieve the greatest happiness of the greatest number.

5 Discussion

Angeline noted that the intron emerged spontaneously from the process of GP evolution and that this emer-

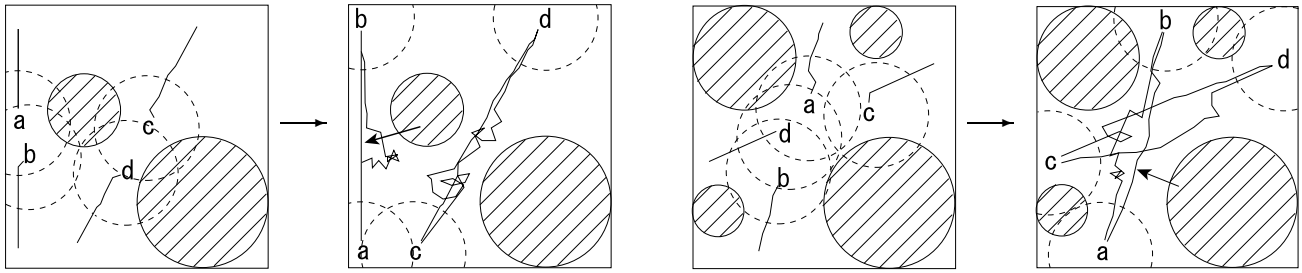
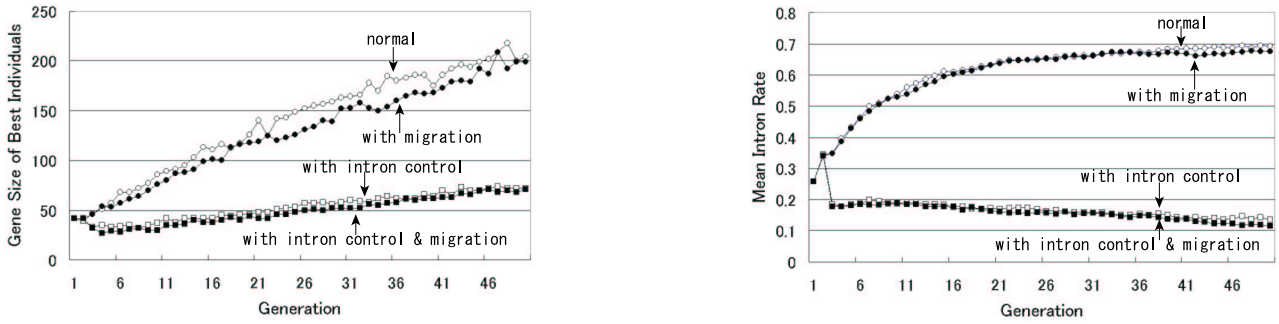


Figure 4: Acquired Behaviors (Robot Navigation)



(a) Program Sizes.

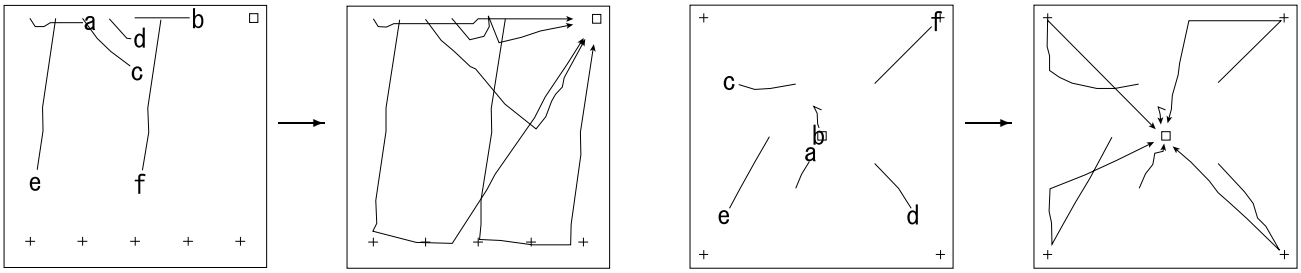
(b) Intron Ratios.

Figure 5: Program Sizes and Introns (Robot Navigation)

gent property was important to successful evolution [Angeline98]. There are pros and cons to the emergence of GP introns (see [EC98] and [Banzhaf *et al.*98] for details). During the early and middle part of a run, introns may have beneficial effects so that good building blocks will be able to protect themselves against the damaging effects of crossover. On the other hand, the exponential growth of introns (i.e., bloat) at the end of the run is probably deleterious.

Our experimental results have shown that removing the introns even at the earlier stage is beneficial. There was no significant performance difference between (a) the runs with removing introns at each generation and (b) those with removing introns every several generations. Moreover, when the introns were removed only at later generations, it gave poorer performance than any other runs. Therefore, we think that the removal of effective introns as often as possible will lead to the success of the multi-agent learning.

The common method for controlling the tree growth in GP is to use the parsimony pressure, i.e., GP individuals are subjected to the selective pressure against length [Soule *et al.*96]. We have conducted comparative experiments with this method for the above navigation task. The used parsimony factor is 0.5, i.e., the fitness is added by the penalty of 0.5 $\times$ Tree Size. The experimental results showed that the fitness improvement by the selective pressure was satisfactory at earlier generations, but that the raw fitness value was about 130 at the final generation, which is better than normal GP, i.e., 140 but worse than GP with controlling effective introns, i.e., 107 (see Fig.8(a)). The number of successful runs was 9 out of 50 cases. The robustness for testing data was just as good as the normal GP. The averaged size of acquired trees was significantly smaller by the selective pressure (see Figs.8(b) and (c)). We have obtained the similar results with other parsimony factors. In summary, although the parsimony pressure contributes to generating smaller trees, it does not necessary evolve a better solution. This is partly because the parsimony pres-



(a) (b) (c) (d)

Figure 7: Experimental Results (Escape Problem)

sure sometimes control the functional codes, i.e., non-intron parts of programs.

In this paper, we mainly describe the method of removing syntactic introns. There is another type of introns, i.e., semantic introns [Angeline98], which are code segments that are executed but have no effect on the overall result. For instance, the codes (+ 0 a) and (not (not x)) include the semantic introns. These introns can be edited and replaced using a predefined template, such as "double not's" or "zero plus" [Koza 92]. Preliminary experiments have shown that editing semantic introns did not any harm or good to our multi-agent GP learning. This is partly due to the fact that these introns seldom occur in our task. We will be in pursuit of the role of these introns in our future research.

6 Conclusion

This paper proposed a controlling strategy of effective introns for multi-agent GP learning. Experimental results showed the effectiveness of our approach in the following points: (1) the fitness transition was improved for training, (2) the code growth was effectively reduced, and (3) the robustness of an acquired program was improved.

Our future research concern is to study this problem on a real robot in the future. We also plan to conduct an experiment with a more difficult problem when the workspace is gradually changed with generations.

References

[EC98] Special Issue: Variable-Length Representation and Noncoding Segments for Evolutionary Algorithms, Evolutionary Computation, vol.6, no.4, MIT Press, 1998

[Angeline98] Angeline,P.J., Subtree Crossover Causes Bloat in Proc. of Genetic Programming Conference 1998 (GP98), 1998

[Banzhaf *et al.*98] Banzhaf,W., Nordin,P., Keller,R.E., and Francone,F.D., Genetic Programming, An Introduction, Morgan Kaufmann, 1998

[Hara *et al.*99] Hara,A., and Nagao,T., Emergence of Cooperative Behavior using ADG; Automatically Defined Groups, in *Proc. of the Genetic and Evolutionary Computation Conference (GECCO99)*, Morgan Kaufmann, 1999

[Haynes *et al.*95] Haynes, T., Wainwright,R., and Sen,S., Evolving a Team, in *Working Notes of the AAAI-95 Fall Symposium on Genetic Programming*, AAAI Press, 1995

[Iba96] Iba,H., Emergent Cooperation for Multiple Agents using Genetic Programming, in *Parallel Problem Solving form Nature IV (PPSN96)*, 1996

[Iba98] Iba,H., Evolutionary Learning of Communicating Agents, *Information Sciences*, 108(1-4), 1998

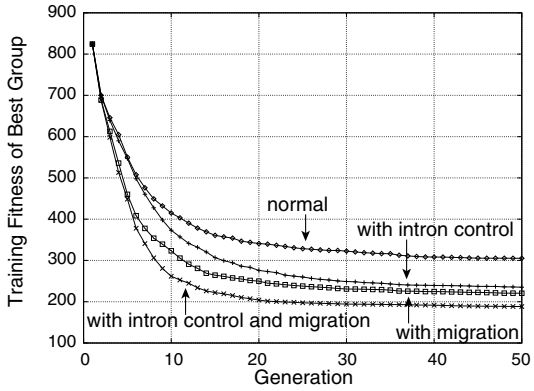
[Ito *et al.*96] Ito,T., Iba,H. and Kimura,M., Robot Programs Generated by Genetic Programming, Japan Advanced Institute of Science and Technology, IS-RR-96-0001I, in *Genetic Programming 96*, 1996

[Koza 92] Koza, J., Genetic Programming, On the Programming of Computers by means of Natural Selection, MIT Press, 1992

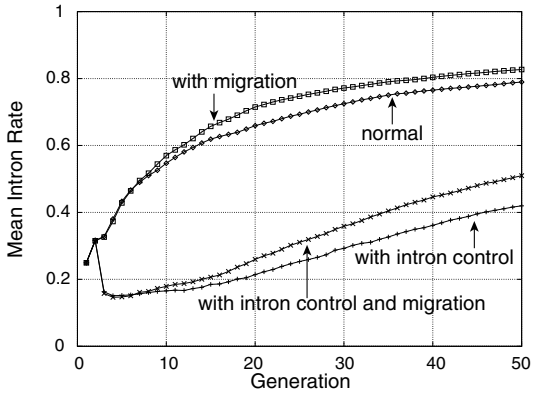
[Luke *et al.*96] Luke,S. and Spector,L., Evolving Teamwork and Coordination with Genetic Programming, in Genetic Programming 96, MIT Press, 1996

[Smith *et al.*96] Smith,P.W.H, and Harries,K., Code Growth, Explicitly Defined Introns, and Alternative Selection Schemes, in *Evolutionary Computation*, vol.6, no,4, MIT Press, 1999

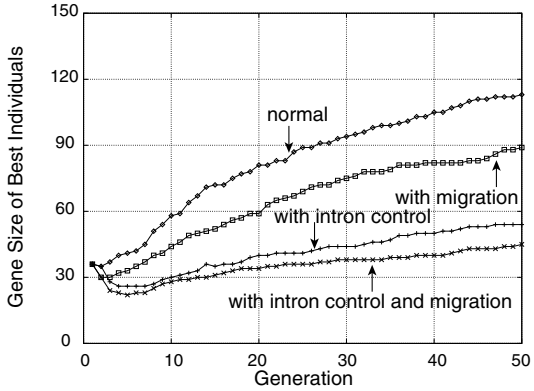
[Soule *et al.*96] Soule,T., Foster,J.A., and Dickinson,J., Code Growth in Genetic Programming, in *Genetic Programming 96*, 1996



(a) Fitness vs. Generations

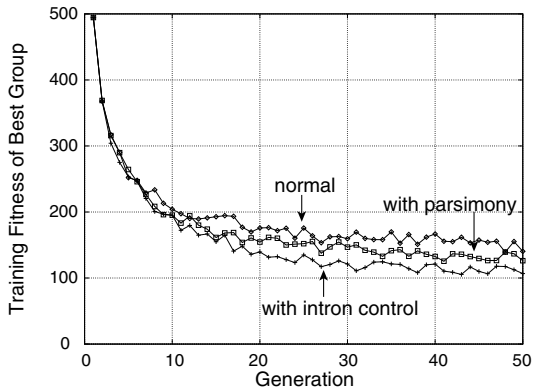


(b) Intron Rate vs. Generations

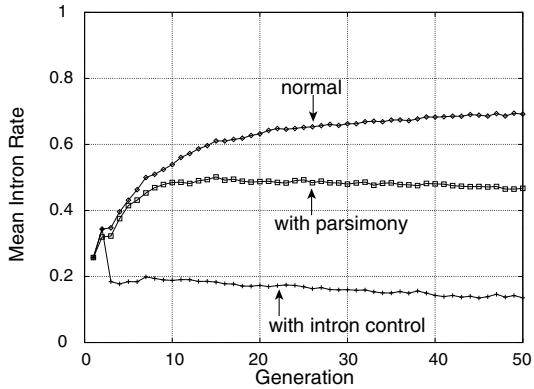


(c) Program Size vs. Generations

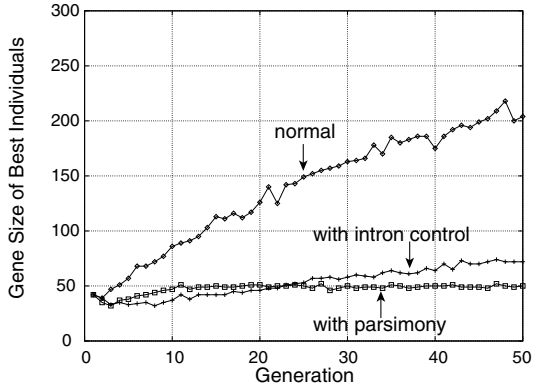
Figure 6: Experimental Results (Escape Problem)



(a) Fitness vs. Generations



(b) Intron Rate vs. Generations



(c) Program Size vs. Generations

Figure 8: Comparison with Penalty Pressure