On-Line Evolution of Robot Arm Control Programs for Visual-Reaching Tasks using Memoized Function

Worasait Suwannik and Prabhas Chongstitvatana
Department of Computer Engineering
Chulalongkorn University
Bangkok, 10330, Thailand

prabhas@chula.ac.th

Abstract

This work applies an evolutionary algorithm called Genetic Programming to generate robot arm control programs for a visual-reaching task. The evolutionary method employs a probabilistic search of control programs that can perform a specified task. Starting with an initial set of programs, Genetic Programming evolves these programs using genetic operators until satisfactory solutions are found. In this paper, the programs are evolved on-line on a real robot in its working environment. The resulted control programs work robustly in the real environment. The on-line evolution does not require the robot model. However due to the large number of trials, which required the physical motion of robots, on-line evolution is very time consuming. To accelerate the on-line evolution, a memoized function is proposed to store the effect of robot arm motions during the evolution. The result shows that the memoized function can speed up the on-line evolution by 23 times. In terms of robustness, the on-line evolution improves the robustness of control programs when compared to an off-line evolution using the simulated robot by 27%.

Keywords

Evolutionary Robotics; Genetic Programming; Robot program; Robustness; Visual-reaching task; On-line evolution.

1. Introduction

1.1 Introduction to Robotics

Robot path planning refers to finding trajectories for a robot to achieve its task in its workspace without collision with the obstacles. The robot R and a set of obstacles O move in a Euclidean space W, call the *workspace*. An articulated object R consists of several moving rigid parts R_1, R_2, ... , R_m, called links, connected by joints. Each joint constrains the relative movements of the two links it connects. The motion planning, or the Piano Mover's problem, is stated as follows: Given a robot R and a workspace W, find a path from an initial configuration I to a goal configuration G, such that R never collides with any object $O_i \in O$ in W along the path P. The path P is a continuous sequence of positions and orientations of R.

The *configuration space* or C-space [1, 2] represents the robot as a point in the robot's configuration space, and maps the obstacles in this space. This mapping transforms the motion planning problem into the problem of planning the motion of a point in a higher dimensional space. The *free configuration space F* is a subset of configuration space [3] at which the robot does not intersect any obstacle. Planning for a collision-free path is to search for connectivity between an initial configuration of robot to its final configuration in *F*. The dimension of configuration space depends on the degree of freedom of the robot. Computing high-dimension configuration space exactly is impractical [4]. There are many approaches to compute an approximate representation of *F*.

The global planning methods such as *Roadmap Algorithm* [5], *Probabilistic Road Map* (PRM) [6, 7] and other geometric methods [8] are guaranteed to find a complete path, if one exists. *Probabilistic Road Map* (PRM) employed Monte Carlo algorithms to construct a representation of *F* by 1-dimensional networks. These methods sample points in *F* and connect them by an edge if they can be connected by a simple path inside *F*. PRM can find a simple collision-free path between initial and final configurations in many situations. The local planning methods, such as artificial potential field methods [9] are usually fast, but are not guaranteed to find a path, even if one exists. Algorithms based on artificial potential field methods define a potential field on the configuration space using a heuristic function rewarding distance to the goal and penalizing closeness to obstacles, therefore steering the robot towards the goal. *Randomized potential fields* [4] employ randomization in the form of random walks to avoid local minima in the potential field.

The configuration space approach has been extended to handle continuous motion and system dynamics, generally called *kinodynamic* planners. LaValle [10] proposed *Rapidly-exploring Random Tree* (RRTs) that encompasses both the constraints from the system dynamics as well as global kinematic constraints due to obstacles in the environment. General kinodynamic planning, given the high-dimensional state space, has been proven to be at least PSPACE-hard [11], with exact, time-optimal trajectory planning in state space being NP-hard [12].

The traditional approaches such as the configuration space approach for robot path planning require the geometrical model of the robot and its workspace. The drawback of these methods is that they require an accurate geometrical model of the robot and the workspace. In the cases where proper models are difficult to obtain, or there are high degree of uncertainty, the traditional approaches are likely to fail.

Alternative approaches to traditional robotics avoid using robot models. These approaches include *Subsumption Architecture* [13] and robot learning algorithms [14, 15]. Subsumption architecture organized robot actions and sensing into several layers. Each layer is autonomous; it has its own necessary sensing and decision making. The higher layer can subsume the lower one by inhibiting its action. Subsumption robots can achieve various tasks with insect level intelligence. It is not clear how to increase the level of intelligence to the human level under this architecture [16]. Another important factor is that programming a robot with Subsumption Architecture requires human intuition and judgments. In contrast, the robot learning approach uses learning algorithms to automatically acquire a control strategy for a robot task.

Various machine learning algorithms can be applied to the task of robot learning. *Evolutionary algorithms* [17] especially *Genetic Programming* has been used successfully to generate control programs for robots. Evolutionary algorithms are inspired by Darwinian principle of natural selection and random mutation. Evolutionary algorithms are a class of probabilistic search algorithms. The search involved a population of solutions as opposed to a single point used by several guided random search techniques such as Hill Climbing. The advantage of using evolutionary algorithms in solving a robotic problem is the ease of a behavioral design. An evolutionary robotics researcher simply specifies how to evaluate the robot performance instead of specifying how the task is done. It is the job of the algorithm to synthesize a robot controller. An introduction to evolutionary algorithms especially Genetic Programming will be given in the next section.

The task of robot arm control is interesting and quite different from controlling a mobile robot. Most experiments in Evolutionary Robotics have been conducted on mobile robots [18, 19, 20]. The dynamic nature of the mobile robot makes it difficult for the robot to be certain about its location. There are many uncertainties in the robot and its environment, for example, the wheels of the mobile robot can slip. Data read from sensors is noisy hence it is difficult to build a good representation of the real world. The problem of controlling a robot arm is quite opposite. It is easy to locate the position of the robot's end effector. The position of the end effector can be calculated from the joint angles given a complete geometrical model of the robot or it can be detected from the robot vision directly without using the model. The difficulty is in robot path planning for a collision free path. Therefore the problem of robot arm control is different from the problem of mobile robot control.

1.2 Introduction to Genetic Programming

Genetic Programming is a program induction approach based on the idea of natural evolution [21]. Genetic Programming is used to induce a robot program. A program in Genetic Programming has a tree structure as shown in Figure 1. Genetic Programming is a variation of *Genetic Algorithms*. Genetic Algorithm was introduced by Holland [22, 23]. Genetic Algorithm simulates the process of natural evolution. Each individual in a Genetic Algorithm population is a fixed-length binary string. A binary string can encode any object such as a number, a list of numbers, or even a computer program.

Unlike Genetic Algorithm, an individual program in Genetic Programming is a computer program whose size and shape are varied. A pseudo code for Genetic Programming is shown in Figure 2. The search is guided by the fitness (i.e., performance criteria) of a program in the population. The fitness value is evaluated by executing the program and observed the end result. Better programs are likely to be selected and are able to reproduce. The genetic operators perform the task of exploring the space of possible programs. The crossover operator recombines sub-trees from the selected programs to produce new programs. The mutation operator randomly changes sub-trees of the selected program to create a new program. The process of fitness evaluation, individual selection and genetic operation continues over generations until a solution program is found or a terminating criterion is met.

The repetitive process of an evolutionary algorithm can be completed in a short amount of time in a computer simulation. A computer simulates the evolution of the robot task using the model of a physical robot and its environment. Although evolving a robot program in simulation is fast, inaccurate models and uncertainty in the real world lead to a fragile behavior when the program is transferred to the real robot.

On-line evolution eliminates the problem of inaccurate robot model by executing a program on the real robot and evaluates its fitness in the real world. However, on-line evolution is very time consuming because the mechanical speed of a physical robot is an order of magnitude slower than a simulated robot. In order to achieve a practical on-line evolution, a *memoized function* is proposed to accelerate the on-line evolution. An introduction to memoized functions is the topic of the next section.

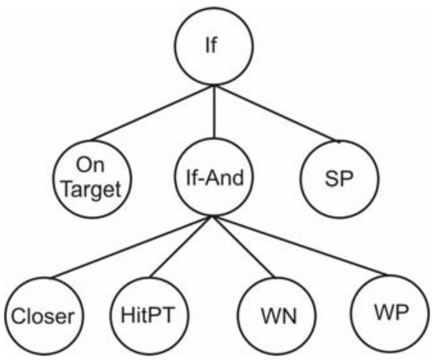

Figure 1. A program in Genetic Programming has a tree structure.

```
1. Randomly create initial population of programs
2. Evaluate each program in the population and assign a fitness value
3. Repeating the following steps until a new population is fully created
   3.1 Select programs using a selection algorithm
   3.2 Apply genetic operation: crossover and mutation on the selected
       program
   3.3 Add the result of genetic operations into the next generation.
4. If the termination criterion is not met, replace the old population with the
   new population and repeat steps 2-4. Otherwise, output the best
   individual program.
```

Figure 2. Genetic Programming pseudo code

1.3 Introduction to Memoized Functions

A memoized function or a memo function was invented by Donald Michie [24]. A pseudo code for the function is shown in Figure 3. A memoized function of a function *f* behaves the same as *f*. However, it remembers every answer of every call to *f*. Therefore, if *f* is called again with the same arguments, it can output the answer immediately without having to recompute *f*. A pseudo code for a memo function is listed below.

```
memo ($f$, $x$)
    if $table[x]$ has value
        return $table[x]$
    else
        $table[x] = f(x)$
where $f$ is a function and $x$ is an argument to the function, $table[.]$ is a data
structure large enough to hold all values of $f(x)$.
```

Figure 3. Memoized function pseudo code

Originally, a memoized function is used to speed up the execution of computer programs. The experiment in Section 5 used the memoized function to speed up the on-line evolution. To apply the concept of memoized function to the on-line evolution, a robot motor command is regarded as a function. Its argument is the current state of the robot. The output of this function is the effect of a motor command on the state of the environment. In the real world, to know the effect of executing a motor command, the actual robot motion is performed. As the motor command is treated as a memoized function, the effect of the motion to the state of the environment is remembered. If the same command is issued with the known state that has already been remembered, the robot arm does not have to move physically, instead, the effect of the command is recalled from the memory. The detailed discussion of the memoized function used in the experiment is exposed in Section 4.

1.4 Organization of the Paper

The organization of this paper is as follows. Section 2 describes the robot configuration used in this work. Section 3 discusses Genetic Programming parameters used in the experiment. Section 4 explains the memoized function for on-line evolution. Section 5 elaborates the experiment on on-line evolution and its result. Section 6 concludes the paper.

2. The Robot Configuration

A simple robot arm with visual feedback is used as a test bed for exploring the evolutionary methods. The task is to plan a collision free path for the robot arm to reach a specified target in an area filled with obstacles. The robot arm used in the experiment is a three-degree of freedoms planar robot. The drawing of the robot arm used in the experiment is shown in Figure 4. Each joint is driven by a position controlled servo. A servo is programmed to be able to move to 60 distinct positions. The step size of each motion is 3 degrees.

A CCD camera located above the arm provides a robot vision. Figure 5 shows a robot arm seen from its vision system. The vision system detects the robot configuration from four black circular marks, each one placed on the pivot of each joint. A link is identified as a line between each pair of joints. The vision system measures the distance between the end effector and the target. It also detects whether the robot hits any obstacles.

There are large degrees of real world difficulties in this configuration. First, the visual distortion from the camera lens is clearly visible. Second, the position control of the joint servo is not very accurate. These factors are intentionally designed into the experiment to create a test bed for testing the robustness of the evolutionary methods. The aim is to subject evolutionary methods to these real world difficulties and studies how well they perform under these conditions.

An off-line evolution for this task is reported in [25]. The robot's forward kinematics model was constructed apriori and then was used during the evolution in simulation. The failure of some control program when used with the real robot has been observed. The main reason is the inaccuracy of the kinematics model and the world model used in simulation. This result illustrates the sensitivity of the off-line evolution to the accuracy of the models used during the evolution.

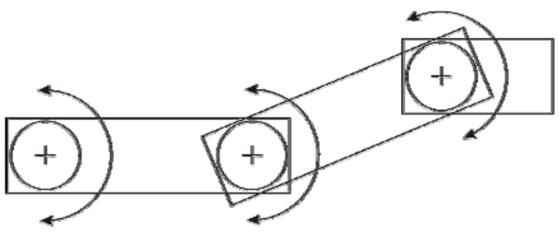
Figure 4. Drawing of the robot arm

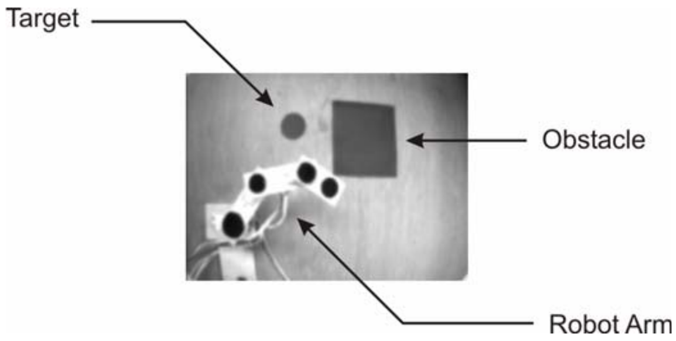

Figure 5. The robot arm seen from the CCD camera

3. Structure of Genetic Programming employed for this study

3.1 Terminal and Function Sets

An individual in Genetic Programming population is a program represented by a tree structure. A node in a program tree is an execution unit in a terminal set and a function set. A primitive in a terminal set is located at a leaf node in a program while a primitive in a function set is an internal node in a program.

Primitives in Genetic Programming can represent robot programs at any level of abstraction. Nordin et. al. [20] evolved a robot program at a machine language level. Suwannik and Chongstitvatana [26] evolved a robot program at a joint command level. Koza [27] evolved a robot program at a behavioral level. There are tradeoffs for choosing level of abstraction. A high level of abstraction reduces the search space but prohibits the discovery of a new set of primitives that may be derived from the lower level data.

The robot arm can perform six primitive motor commands. Each primitive moves a specified joint by one step in one particular direction (i.e., clockwise or counterclockwise.) The vision system can check if the end effector is closer to, farther from, or located on the target. It can also detect whether the robot hits any obstacles. The primitives for motor and sensing functions of the robot are summarized in the terminal set shown in Table 1.

The structure of a robot program consists of nested if-then rules. As shown in Table 2, the function set includes basic control flow primitives: If, If-And, If-Or, If-Not. These control flows are used to connect nodes together and control the flow of program execution.

Table 1. Terminal set for the visual-reaching task.

Terminal Name	Operation
Sp, Ep, Wp	Rotate a shoulder, elbow, or wrist clockwise. Return true if the operation is successful. Otherwise, return false.
Sm,Em,Wm	Rotate a shoulder, elbow, or wrist counterclockwise. Return true if the operation is successful. Otherwise, return false.
Closer	Return true if the last servo command causes the end effector of the robot to move closer to the target. Otherwise, return false.
Farther	Return true if the last servo command causes the end effector to move away from the target. Otherwise, return false.
HitPT	Return true if the last clockwise move results in hitting an obstacle. Otherwise, return false.
HitNT	Return true if the last counterclockwise move results in hitting an obstacle. Otherwise, return false.
OnTarget	Return true if the end effector is on the target (i.e., the distance between their centroid is less than 3 pixels). Otherwise, return false.
See	Return true if the end effector can see the target (i.e., there are no obstacles between the end effector and the target). Otherwise, return false.

Table 2. Function set for the visual-reaching task.

Function Name	Operation
If	Evaluate the first child. If the evaluation result is true, evaluate the second child. Otherwise, evaluate the third child.
If-And	Evaluate the third child if the evaluation result first and the second children are both true. Otherwise, evaluate the fourth child.
If-Or	Evaluate the third child if one of the evaluation result first and the second children is true. Otherwise, evaluate the fourth child.
If-Not	Evaluate the first child. If the evaluation result is false, evaluate the second child. Otherwise, evaluate the third child.

3.2 Fitness Function

Genetic Programming can be regarded as a program searching method guided by a fitness function. For each generation, every robot program in the population is evaluated on how well it can perform the task. This evaluation is called a fitness evaluation. To evaluate the performance of a robot program, the program is executed for a specific number of steps, counting each robot movement as one step. Hence all programs will be allocated the same amount of time for their evaluation. After the program was terminated, the fitness value is calculated. The fitness function follows directly from the objective of the reaching task. The further the distance from the end effector to the target, the lower the fitness. A fitness function is defined as follows.

$$fitness = \begin{cases} 1000 & ; d \le 2 \\ 1000 - d & ; otherwise \end{cases} \qquad (1)$$

where d is the distance between the target and the end effector after the program terminated.

3.3 Genetic Parameters

Genetic Programming was run with the parameters shown in Table 3. The population size is an important factor especially in on-line evolution. To determine an appropriate population size, the total running time of the on-line evolution is estimated in simulation. The evolution time was estimated by counting the number of moves multiplied by the duration for each move, which is one second. Among the candidate population size, it was found that the population size of 200 programs gave the best result in the simulation.

Normally, Genetic Programming will be terminated when the solution is found or the maximum number of generation is reached. However, in this experiment, there is no limit of the number of generations. If evolution progresses (i.e., the best fitness value is increasing), it would go on. The reason for using this terminating condition is to avoid restarting the evolution process. Restarting the on-line evolution from the first generation is costly. Starting a new run means randomly create a new population of robot programs. This new population of programs tends to move the robot to different configurations. Therefore, the memoized function would not be effective. The robot has to make a large number of moves physically instead of using the results stored in the memory and thus slow down the evolution.

Table 3. Genetic Programming parameters for the on-line evolution.

Name	**Value**
Population	200 programs
The method used for generating the first generation	Grow method with depth limit 4
Crossover rate	80%
Reproduction rate	10%
Mutation rate	10%
Selection method	Tournament selection with tournament size 7
Maximum generations	No limits on maximum generation. Evolution stops when no progress is made during 10 consecutive generations

4. Memoized Function

4.1 Motivation

The on-line evolution is very time consuming. A robot program is evolved entirely on a real robot in a physical environment. The mechanical speed of the robot is several magnitudes slower than the speed of executing a computer instruction. It is prohibitive to evaluate a large number of programs on-line. In [28], it is reported that evolving an obstacle avoiding mobile robot would take about 30 hours. In [25], it is estimated that evolving a target-reaching behavior for a robot arm might take 130 hours.

To accelerate the robot program evaluation time during evolution, the effect of the robot motions in the real world should be stored and reused. The effect includes the position of each joint after the motion is completed and whether the arm hits any obstacle. A robot equipped with this knowledge can move *virtually* (i.e., without the actual motion in the real world) thus it is very fast. Initially, there is no knowledge about the robot's kinematics and the model of the environment since the model is discarded in on-line evolution. A robot incrementally acquired the knowledge during the execution of a robot program during on-line evolution. The implementation of the storage of this knowledge is in the form of a function indexed by robot joint angles. Replacement of an actual motion with a virtual one can be realized by a memoized function.

4.2 Structure and Implementation in this Study

The memo version of a function *f* (i.e., memo *f*) is a version of *f* that caches the result of the previous calls to *f*. Caching the result is implemented using a hash table. The arguments to a function are a key of the hash table.

A robot movement is regarded as a function that takes three joint angles. It returns a structure that contains a coordinate of every joint position and a flag that tells whether the arm hits an obstacle at that configuration. Since there are three joints, the memo table is implemented using a 3-dimensional array. The array can hold every possible combination in the joint space. Each entry is indexed by three joint servo angles. Since each joint has 60 discrete steps, there are a 60^3 or 216,000 entries.

4.3 Limitations

When dealing with higher degree of freedom robots, it is impossible to allocate memory large enough to hold every possible value. However, as it shall be seen in the next section, while evolving a program, a very small percentage of the table is occupied. There are many suitable techniques to implement a sparse table.

5. On-line Evolution

5.1 Motivation

Evolving a robot program from inaccurate simulation models is a reason for a fragile behavior of a physical robot when transferred to the real world [29]. Several researchers also reported a reality-simulation gap problem [30, 31, 19]. A program evolved from simulation is not robust when transfer to the real world. Several approaches have been proposed to improve the robustness of robot controller evolved from simulation. A realistic simulation can improve the robustness of the control program.

A realistic simulation can be built by capturing sensor and motor data from the real robot. This approach can be used with a mobile robot as in [32, 33]. It can also be used with a robot arm as in [34]. However, capturing all possible configurations of the robot is a time consuming task.

Noise can be added to simulation in order to capture the effect of uncertainty. The resulting programs evolved from the simulation with noise show the ability to work more robustly in the real robot. However, this approach requires somewhat exact noise model. In other words, noise also has to be modeled. The robot is most robust when noise is modeled with the same level of noise found in reality [31]. Too little noise or too much noise in simulation causes the robot to perform erratically in the real environment.

Adaptive control architecture can improve the robustness of a robot controller. An adaptive controller changes its strategy depending on situation. An example of adaptive controller is an artificial neuro modulator. In nature, a neuro modulator is a biological system found in an animal neuron system. It can rearrange the neuron network structure as well as the neuron network weight. By using this method, a peg-pushing robot was evolved in a simulation and was able to transfer the controller to the real robot [18]. In [28], a neuro modulator with Hebb rules was evolved. In both cases, the network weight is updated during the lifetime of the robot. The adaptation process continues after the evolution is completed.

Another method to improve the robustness is by evolving a program with multiple trials. A program that is evolved with multiple trials can handle changes better than a program that is evolved with only one trial. In [35], the author reported that this approach has two benefits. First, the robot evolved with multiple trials is capable of handling uncertainty in the real world. Second, the simulation can be created more easily. The robot can accomplish the task without relying on some aspects that are difficult to simulate. Instead of using multiple trials, multiple environments can also be used to the same benefit. The robustness of a simulated mobile robot is improved by evolving it with several environments [36]. The programs were more robust as more trials were used during training.

Although simulation is a useful tool to study some fundamental problems in evolutionary robotics, Brooks [30] mentioned that when using simulation, the effort might have been directed to solve a problem that does not exist in the real world. Thus, the solution of a problem using simulation may not work in the real world. This is because the world is usually simplified in simulation. Many aspects of the real world such as noise and uncertainty are ignored.

5.2 Approach

In principle, an artificial evolution can be performed without using any simulation model. This type of evolution is called the on-line evolution. The on-line artificial evolution occurs in the physical world. It occurs entirely on a real robot in its environment. However, the on-line evolution is very time consuming. The mechanical speed of the real robot is an order of magnitude slower than the simulated one.

Several researchers proposed methods for reducing the on-line evolution time. Miglino et. al. [37] pre-evolved a robot behavior in simulation and continues the evolution process in the real world. Olmer et. al. [38] decomposed task into subtasks and evolved subtask. After that, a strategy for selecting subtasks was evolved. This technique can accelerate the evolution time because a subtask is easier to evolve. Nordin et. al. [20] evolved an obstacles-avoidance behavior of a mobile robot. A minimal world model from the past experience was stored in a memory buffer.

This paper proposed a method to speed up the on-line Genetic Programming of robot programs using a memoized function. The memoized function remembers the result of the physical robot movement. If the robot receives the same command, it will use the memorized result.

5.3 Experiment

The aim of the experiment is to evolve a robot program on-line using Genetic Programming. Genetic Programming is run on a Microsoft Windows based personal computer that connects to a real robot arm and its vision system. We implemented Genetic Programming using C++ language.

In the off-line experiment, a robot learnt its task in a simulator. Although the robot is not complex, building an accurate simulation model is difficult because of the following reasons.

- The lens distortion is not modeled. The vision system used in the experiment has high distortion but there is no such effect in simulation. It can be seen from Figure 5 that a black square is distorted. The lens distortion causes the length of each link to vary between configurations when measurement is made through the camera.
- The noise in locating the position of each joint due to the vision system is not modeled. The vision system locates each joint by recognizing its circular marker and calculates its centroid.
- The light caused uneven reflection in the scene as can be seen from Figure 5.
- The position of each joint is not calibrated with the real robot.

A program is successful when it can move the end effector to the center of the target within the range 2 pixels, as seen from the vision system. A program that can do so will receive the fitness value of 1000.

The robustness of a robot program is the percentage of times the robot can successfully perform the task. The robustness of the evolved program is important. Even with the on-line evolution, the resulting program can fail to work. To measure the robustness, the result from on-line evolution was tested for 10 times on the same environment they were evolved. Robot programs were evolved in five different environments as shown in Figure 6.

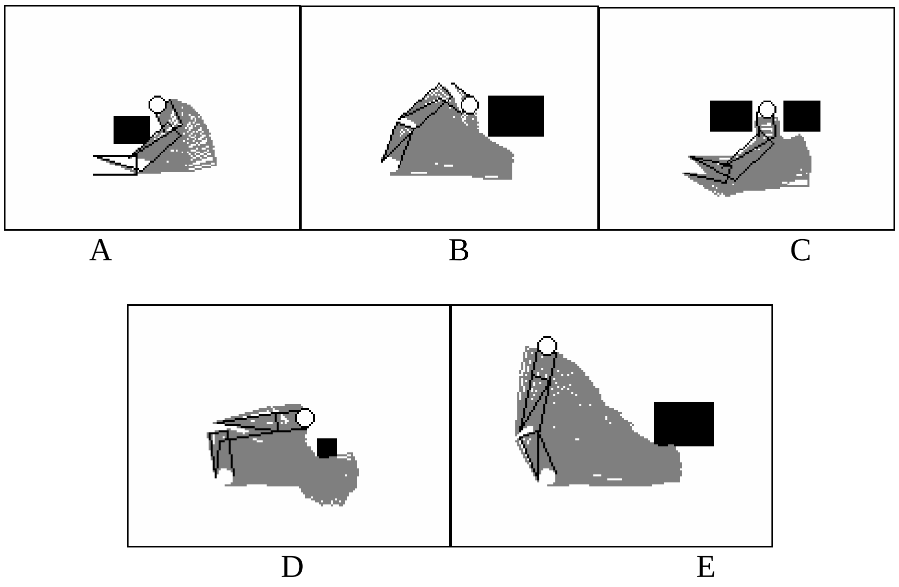

Figure 6. Five environments for testing the visual-reaching task. In each sub figure, a white circle is a target. Black rectangles are obstacles.

5.4 Result

The fitness curve of the best individual from a particular run in on-line evolution is shown in Figure 7. The task is successfully achieved in the generation 11 with the fitness 1000. To measure the performance of the memoized function, the fitness evaluation time is recorded. In Table 4, the column labeled *Actual time* is the on-line evolution time with memoized function. The column labeled *Estimated time* is an estimation of the on-line evolution time without using a memoized function. *Speed up* is the ratio between the estimated and the actual time. Using the memoized function can speed up the on-line evolution by 23 times on average.

The memory usage is the percentage of entries in the memo table that were used during the evolution. In analyzing the memoized function's memory usage, less than 3% of the memory allocated for the memoized function was used. This suggested that it is possible to evolve a program for a robot with larger degree of freedom.

The recall rate is the percentage of requested data that is found in the memory. As show in Figure 8, the percentage of recall and speed up are related. Each recall means that the robot does not have to move in the physical world. When there is no recall, the speed up is equal to one (i.e., no speed up.) The speed up grows much larger when the recall rate is above 95%. If the recall rate is high, then it is possible to evolve a control program for a more complex task in a reasonable amount of time.

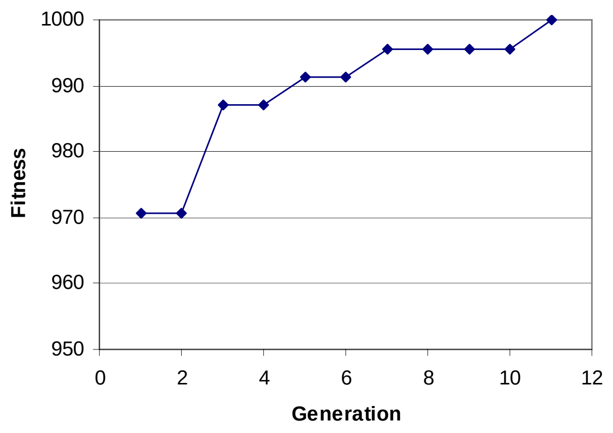

Figure 7. Fitness curve of the best individual for each generation.

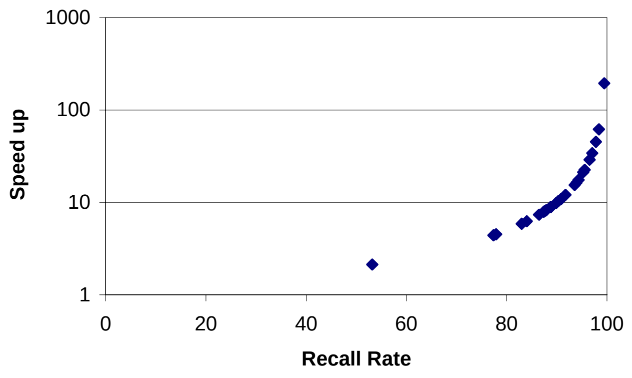

Figure 8. Recall Rate and Speed up of a memoized function
(the speed up is in log scale)

The robustness of the robot program is a percentage of time a robot can successfully perform the task. We compared the robustness of programs evolved on-line and off-line. In the off-line evolution, a robot program is evolved in simulation. After the evolution is completed, the result program is tested on the real robot. As shown in Table 5. The average robustness of a program evolved on-line is 27% higher than the one evolved off-line. The robustness depends on the similarity between the training environment and the testing environment. The on-line evolution occurred in the actual working environment as opposed to the off-line evolution. Therefore, a program evolved on-line evolution tends to have higher robustness.

When tested on the real robot, it has been observed that a failure program moved the robot arm to configurations that were not found in the memory three times on average more than the successful program did as shown in Table 6. In other words, the failed program moved to the robot configurations that were not visited during the evolution. It is hypothesized that this is due to noise in the system that leads the robot to move to unseen configurations.

Table 4. Memoized function: speed up and memory usage.

Environment	Actual time (hours)	Estimate time (hours)	Speed up (times)	Recall (%)	Memory used (%)
A	0.51	5.00	9.64	89.52	0.90
B	0.64	6.33	7.80	85.65	1.13
C	1.19	25.32	20.93	94.82	2.20
D	1.28	132.68	59.09	93.45	2.31
E	0.93	34.81	17.18	81.91	1.79
Average	0.91	40.83	22.93	89.07	1.67

Table 5. Robustness of an evolved program.

Instance	Robustness of a program evolved off-line	Robustness of a program evolved on-line with memoized function
A	82	98
B	26	76
C	82	82
D	56	60
E	28	92
Average	55	82

Table 6. Average number of times the robot moves to a configuration that is not found in the memory.

Instance	Average number of times the robot move to a configuration that is not found in the memory	
	In success cases	In failure cases
A	15	140
B	81	112
C	63	214
D	27	83
E	33	102
Average	44	130

7. Conclusion and Further work

Robot arm control programs for a visual-reaching task can be evolved off-line and on-line. The off-line evolution requires the model of a robot and its working environment. An inaccurate model leads to fragile behavior of a real robot. As opposed to the off-line evolution, the on-line evolution of a robot program does not require any model. Therefore, it is suitable for a robot or an environment which models are difficult to obtain such as a robot created from servos connected randomly or *Random Morphology* robots [39]. However, the on-line evolution is very time consuming. By using memoized function, the on-line evolution of a visual-reaching task can be sped up by 23 times. It took less than one hour to evolve a control program on-line. Moreover, the resulting program is 27% more robust when tested on the real robot.

The on-line evolution method proposed in this work also has a more general implication. The on-line evolution of a robot controller can be viewed as two separated parts. The first part is to learn the model of the robot and its environment. The sensory-action learner can be implemented as a function. The simplest model learner is a memoized function. The second part is to learn how to perform the task. In this paper, we used Genetic Programming to learn the reaching task. Two learners are independent of each other and can be chosen in various combinations. One of the examples is an obstacle-avoiding target-reaching controller reported in [40]. A neural network can be used to learn the model of a robot arm from visual feedback. A search algorithm is used to plan sequences of motion to perform the target reaching task.

References

[1] T. Lozano-Perez, J. Jones, E. Mazer, HANDEY: A Robot Task Planner, MIT Press, 1992.

[2] T. Lozano-Perez, and R. Wilson, "*Assembly sequencing for arbitrary motions*," Proc. IEEE Int. Conf. on Robotics and Automation, 1993.

[3] J. Latombe, Robot motion planning, Kluwer academic publishers, 1991.

[4] J. Barraquand and J. Latombe, "*Robot motion planning: a distributed representation approach*," Int. J. of Robotic Research, 10(6):628-649, 1991.

[5] J. Canny, The complexity of robot motion planning, ACM doctoral dissertation award, MIT Press, 1988.

[6] L. Kavraki, J. Latombe, R. Motwani, and P. Raghavan, "*Randomized query processing in robot path planning*," Proc. of the 27th Annual ACM Symposium on Theory of Computing, 1995, pp.353-362.

[7] L. Kavraki, P. Svestka, J. Latombe, and M. Overmars, "*Probabilistic roadmaps for path planning in high dimensional configuration spaces*," IEEE Trans. Robot. Autom. 12:566-580, 1996.

[8] M. Foskey, M. Garber, M. Lin, and D. Manocha, "*A voronoi-based hybrid planner*," Proc. of IEEE/RSJ Int. Conf. on Intelligent Robots and Systems, 2001.

[9] O. Khatib, "*Real-time obstacle avoidance for manipulators and mobile robots*," Int. J. of Robotic Research, 5(1):90-98, 1996.

[10] S. LaValle and J. Kuffner, "*Randomized kinodynamic planning*," Int. J. of Robotics Research, 20(50):378-400, 2001.

[11] J. Reif, "*Complexity of the mover's problem and generalizations*," Proc. 20th IEEE Sym. on Foundations of Computer Science, 1979, pp.421-427.

[12] B. Donald, P. Xavier, J. Canny, and J. Reif, “*Kinodynamics motion planning*,” Journal of the ACM, 40(5):1048-1066, 1993.

[13] R. Brooks, “*Intelligence Without Reason*,” Proc. International Joint Conference on Artificial Intelligence, 1991, pp.569-595.

[14] I. Harvey, P. Husbands, D. Cliff, A. Thompson, N. Jakobi, Evolutionary Robotics: the Sussex Approach, Robotics and Autonomous Systems, Elsevier, 1997, pp.205-224.

[15] S. Mahadevan, J. Connell, “*Automatic Programming of Behavior-based Robots using Reinforcement Learning*,” Proc. National Conference on Artificial Intelligence, 1991, pp.768-773.

[16] R. Brooks, “*From Earwigs to Humans*,” IEEE Robotics and Autonomous Systems, 20:291-304, 1997

[17] D. Fogel, Evolutionary Computation: Toward a New Philosophy of Machine Intelligence, Wiley-IEEE Press, 1999.

[18] A. Ishiguro, S. Tokura, T. Kondo, Y. Uchikawa, P. Eggenberger, “*Reduction of the Gap between Simulated and Real Environments in Evolutionary Robotics: a Dynamically-Rearranging Neural Network Approach,*” Proc. IEEE Systems, Man, and Cybernetics, 1999, pp.239-244.

[19] M. Mataric, D. Cliff, “*Challenges in Evolving Controllers for Physical Robots*,” Robotics and Autonomous Systems, 19(1):67-83, 1996.

[20] P. Nordin, W. Banzhaf, “*Real Time Control of a Khepera Robot using Genetic Programming*,” Cybernetics and Control, 26:553-561, 1997.

[21] J. Koza, Genetic Programming, Vol. 1, MIT Press, 1992.

[22] J. Holland, Adaptation in Natural and Artificial Systems, University of Michigan Press, 1975.

[23] D. Goldberg, Genetic Algorithms in Search, Optimization and Machine Learning, Addison-Wesley, 1989.

[24] P. Norvig, “*Techniques for Automatic Memoization with Application to Context-Free Parsing*,” Computational Linguistics, 1991, pp.991-998.

[25] J. Polvichai, P. Chongstitvatana, “*Visually-Guided Reaching by Genetic Programming*,” Proc. Asian Conference on Computer Vision, 1995, pp.329-333.

[26] W. Suwannik, P. Chongstitvatana, “*Improving the Robustness of Evolved Robot Arm Control Programs Generated by Genetic Programming*,” Proc. Int. Conf. on Intelligent Technologies, 2000, pp.149-153.

[27] J. Koza, “*Evolution of Subsumption using Genetic Programming*,” Proc. European Conference on Artificial Life, 1992, pp.110-119.

[28] D. Floreano, J. Urzelai, “*Evolution of Plastic Control Networks*,” Autonomous Robots, 11:311-317, 2001.

[29] W. Suwannik, P. Chongstitvatana, “*Improving the Robustness of Evolved Robot Arm Control Programs with Multiple Configuration*,” Proc. Asian Symposium on Industrial Automation and Robotics, 2001, pp.87-90.

[30] R. Brooks, “*Artificial Life and Real Robots*,” European Conference on Artificial Life, 1991, pp.3-10.

[31] N. Jakobi, P. Husbands, I. Harvey, “*Noise and the Reality Gap: The Use of Simulation in Evolutionary Robotics*,” Proc. European Conference on Artificial Life, 1995, pp.704-720.

[32] W. Lee, J. Hallam, H. Lund, “*Applying Genetic Programming to Evolve Behavior Primitives and Arbitrators for Mobile Robots*,”: Proc. IEEE Int. Conf. on Evolutionary Computation, 1997, pp.501-506.

[33] H. Lund, J. Hallam, “*Evolving Sufficient Robot Controllers*,” Proc. IEEE 4th Int. Conf. on Evolutionary Computation, 1997, pp.495-499.

[34] P. Chongstitvatana, J. Polvichai, “*Learning a Visual Task by Genetic Programming*,” Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems, 1996, pp.534-540.

[35] N. Jakobi, “*The Minimal Simulation Approach to Evolutionary Robotics*,” Proc. Evolutionary Robotics, 1998.

[36] P. Chongstitvatana, “*Using Perturbation to Improve Robustness of Solutions Generated by Genetic Programming for Robot Learning*,” Journal of Circuits, Systems and Computer, 9(1&2):133-143, 1999.

[37] O. Miglino, H. Lund, S. Nolfi, “*Evolving Mobile Robots in Simulated and Real Environments*,” Artificial Life, 1995, pp.417-434.

[38] M. Olmer, P. Nordin, W. Banzhaf, “*Evolving real-time behavioral modules for a robot with GP*,” Proc. 6th Int. Symposium on Robotics and Manufacturing, 1996, pp.675-680.

[39] P. Dittrich, R. Burgel, W. Banzhaf, “*Learning to Move a Robot with Random Morphology*,” Proc. European Workshop on Evolutionary Robotics, 1998, pp.165-178.

[40] B. Mel, Connectionist Robot Motion Planning, Academic Press, 1990.