Feature Discovery in Reinforcement Learning using Genetic Programming

Paper ID

Author information is hidden for blind review

Abstract. The goal of reinforcement learning is to find a policy, directly or indirectly through a value function, that maximizes the expected reward accumulated by an agent over time based on its interactions with the environment; a function of the state has to be learned. In some problems, it may not be feasible, or even possible, to use the state variables as they are. Instead, a set of features are computed and used as input. However, finding a "good" set of features is generally a tedious task which requires a good domain knowledge. In this paper, we propose a genetic programming based approach for feature discovery in reinforcement learning. A population of individuals each representing possibly different number of candidate features is evolved, and feature sets are evaluated by their average performance on short learning trials. The results of experiments conducted on several benchmark problems demonstrate that the resulting features allow the agent to learn better policies.

1 Introduction

Reinforcement learning (RL) is the problem faced by an agent that is situated in an environment and must learn a particular behavior through repeated trial-and-error interactions with it [1]; the agent observes the state of the environment, chooses its actions based on these observations and in return receives some kind of "reward", in other words *reinforcement signal*, from the environment as feedback. Usually, it is assumed that the decision of the agent depends only on the current state but not the previous ones, i.e. has the Markovian property. The aim of the agent is to find a policy, a way of choosing actions, that maximizes its overall gain. Here, the gain is defined as a function of rewards, such as the (discounted) sum or average over a time period. Unlike supervised learning problem, in RL correct input/output pairs, i.e. optimal action at a given situation, are not presented to the agent, nor sub-optimal actions explicitly corrected. One key aspect of RL is that the rewards can be *delayed* in the sense that immediate rewards received by the agent may not be reflecting the true values of the chosen actions. For example, in the game of chess a move which causes your opponent to capture a piece of yours can be regarded as a "bad" move. However, a series of such moves on purpose may be essential to win the game and consequently receive a high reward. In a simplified setting, RL as defined above is closely related with the kind of learning and decision making problems that human beings face in their daily lives.

There are two main approaches for solving RL problems. In the first approach, the agent maintains a function $V^{\pi}(s)$, called *value function*, that estimates the expected return when starting in state s and following policy π thereafter, and tries to converge to the value function of the optimal policy. The policy is inferred from the value function. Alternatively, in *policy space* approaches, policy is represented as a parameterized function from states to actions and optimal policy is searched directly in the space of such functions. There also exist methods that combine both approaches. Note that, in any case, the functions that we are trying to learn (either value function, policy, or both) are naturally functions of the state (observation) variables. However, in a given problem (i) all these variables may not be relevant, which leads to *feature selection* problem, or worse (ii) in their raw form they may be inadequate for successful and/or efficient learning and it may be essential to use some kind of *feature extraction.* The most obvious case is when the number of states is large, or infinite, and each state variable by itself reflects limited and local information about the problem. Let us consider the popular game of Tetris. In Tetris, traditionally each state variable corresponds to the binary (occupied/empty) status of a particular square of the grid. Not only the number of possible states increases exponentially with respect to the size of the grid, but also each state variable tells very little about the overall situation. A human player (most successful computer players as well) instead takes into consideration more informative features that are computable from the state variables, such as the height of each column or the number of holes in the occupied regions of the grid, and decides on his actions accordingly. Similar reductions are also quite common in other domains, such as image processing applications where instead of the raw image various high level features derived from it are fed into learning algorithms. Furthermore, in some cases, additional features can be useful to improve the performance of learning. An example of this situation is presented in Figure 1 for the classical cart-pole balancing problem[1]. By adding sine and cosine of the pole's angle as new features to existing state variables, optimal policy can be attained much faster.

The set of features that are to be used instead of or together with state variables are usually defined by the user based on extensive domain knowledge. They can either be fixed, or one can start from an initial subset of possible features and iteratively introduce remaining features based on the performance of the current set, so called *feature iteration approach* [2]. However, as the complexity of the problem increases it also gets progressively more difficult to come up with a good set of features. Therefore, given a problem, it is highly desirable to find such features automatically in an unsupervised fashion solely based on the observations of the agent. In this paper, we explored using a genetic programming based approach for that purpose.

[1] In the cart-pole problem, there is a pole attached from the bottom to a cart that travels along a track. The objective is to hold the pole in upright position by applying certain amount of force to the cart. The state variables are the pole's angle and angular velocity and the cart's horizontal position and velocity.

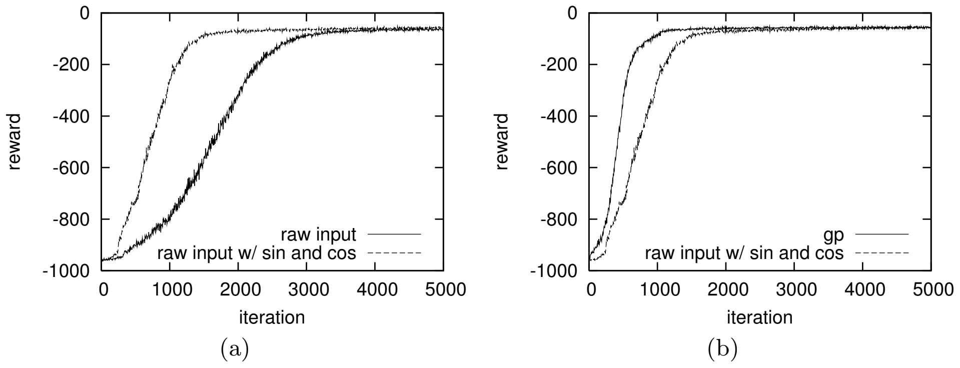


Fig. 1. (a) Policy gradient with Rprop update and optimal baseline on the cart-pole problem using a linear parameterized policy (see [3] for more detailed discussion). Original state variables vs. original state variables plus sine and cosine of the angle of the pole as additional features. Policy is updated every 5 episodes. (b) Performance of the features found by GP.

The rest of the paper is organized as follows: In Section 2, we review related work on feature discovery in RL and GP based approaches in other fields. Section 3 describes our method in detail. Section 4 presents some empirical evaluations of our approach on some benchmark problems. We conclude in Section 5 with a discussion of results and future work.

2 Related Work

Feature discovery, in essence, is an information transformation problem; the input data is converted into another form that "better" describes the underlying noumena and "easier" to process by the agent. As such, it can be applied as a preprocessing step to a wide range of problems and attracted attention from other fields particularly for classification. In [4], Krawiec studies the change of representation of input data for machine learners and genetic programming based construction of features within the scope of classification. Each individual encodes a *fixed* number of new feature definitions expressed as s-expressions. In order to determine the fitness of an individual, first a new data set is generated by computing feature values for all training examples and then a classifier (decision tree learner) is trained on this data set. The resulting average accuracy of classification becomes the evaluation of the individual. He also proposes an extended method in which each feature of an individual is assigned a utility that measures how valuable that feature is and the most valuable features do not involve in evolutionary search process. This protects them from possible harmful modifications and reduces the probability of accidentally abandoning promising search directions. It is possible to view this extension as an elitist scheme at the level of an individual. While Krawiec's approach has some similarities with our approach presented in this paper, they differ in their focus of attention and furthermore we consider the case in which the number of features is not fixed but also determined by GP.

Smith and Bull [5] have also used GP for feature construction in classification. However, they follow a layered approach: in the first stage, a fixed number of new features (equal to the number of attributes in the dataset subject to a minimum 7) is generated as in Krawiec (again using a decision tree classifier and without the protection extension), and then a genetic algorithm is used to select the most predictive ones from the union of new features and the original ones by trying different combinations. Although their method can automatically determine the number of features after the second stage, in problems with large number of attributes the first stage is likely to suffer as it will try to find a large number of features as well (which consequently affects the second stage).

In RL, Sanner [6] recently introduced a technique for online feature discovery in relational reinforcement learning, in which the the value function is represented as a ground relational naive Bayes net and structure learning is focused on frequently visited portions of the state space. The features are relations built from the problem attributes and the method uses a variant of *Apriori* data mining algorithm to identify features that co-occur with a high frequency and creates a new joint feature as necessary.

3 Feature Discovery in RL using GP

Genetic programming (GP) is an evolutionary algorithm based methodology in which each individual represents a computer program and a population of individuals is progressively evolved under the influence of several genetic operators (such as crossover and mutation) and evaluated over a series of generations [7]. Evolution and "survival of the fittest" dynamics results in individuals having higher fitness scores, i.e. programs that better solve the posed problem. GP is shown to be quite effective on various domains and in some cases produced results that are competitive with or surpass human performance [8].

When GP is being applied to a particular problem, there are three main issues that need to be addressed:

1. structure and building blocks (i.e. primitive functions and terminals) of the individuals,
2. set of genetic operators, and
3. fitness function.

In our case, due to the fact that we are interested in identifying useful features for a given RL problem, each individual must essentially be a program that generates a set of features based on the state variables. Therefore, state variables are the *independent variables* of the problem and are included in the set of terminals together with (ephemeral) constants and possible problem specific zero argument functions. An individual consists of a list of s-expressions, called *feature-functions*, such that each s-expression corresponds to a unique feature represented as a function of various arithmetic and logical operators and terminals. Given the values of state variables, the agent can calculate the value of features by evaluating the s-expressions. In our implementation, we linearized each s-expression in prefix-order and then concatenated together to obtain the final encoding of the individual. This compact form helps to reduce memory requirements and also simplifies operations. As we will describe later, each individual is dynamically compiled into executable binary form for evaluation and consequently this encoding is accessed/modified only when genetic operators are applied to the individual.

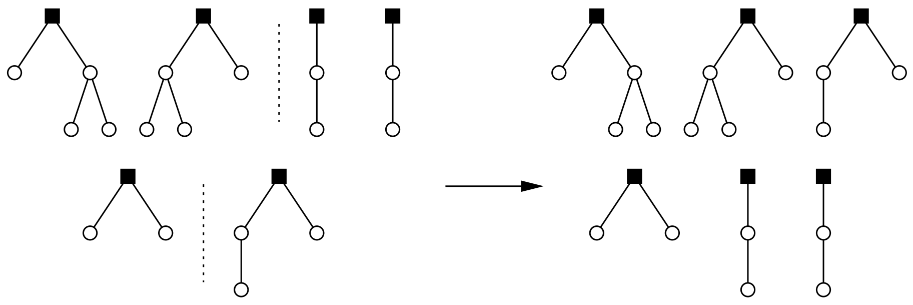


Fig. 2. Single point cross-over on feature lists of two individuals. Cross-over point is shown by vertical dashed line. The number of features represented by the off-springs differ from that of their parents.

Note that, the number of useful features is not known a priori (we are indeed searching for them) and has to be determined. Instead of fixing this number to an arbitrary value, we allowed the individuals to accommodate varying number of feature functions (s-expressions) within a range, typically less than a multiple of the number of raw state variables, and let evolutionary mechanism search for an optimal value. To facilitate the search, in addition to regular genetic operators presented in Table 1 we also defined a single-point crossover operator over the feature function lists of two individuals. Let n and m be the number of features of two individuals selected for cross-over, and $0 < i < n$ and $0 < j < m$ be two random numbers. First i features of the first individual is merged with the last $m - j$ features of the second individual, and first j features of the second individual is merged with the last $n - i$ features of the first individual to generate two offsprings (Figure 2). The generated offsprings contain a mixture of features from both traits and may have different number of features than their parents.

Since our overall goal is to improve the performance of learning, the obvious choice for the fitness of an individual is the expected performance level achieved by the agent when the corresponding feature functions are applied on a particular RL algorithm. In this work, we opted two different algorithms for

Cross-over One of the nodes in any feature function of an individual and the whole branch under it is switched with another node from another individual in the population.

Mutation (node) One of the nodes in any feature function of an individual is substituted with another compatible one – a terminal or a zero argument function is replaced with a terminal or a zero argument function, and an n-ary operator is replaced with an n-ary operator. Note that the branch under the mutated node, if any, is not affected.

Mutation (tree) One of the nodes in any feature function of an individual and the whole branch under it is substituted with a new randomly generated sub-tree having a depth of 3 or less.

Shrinkage One of the operator nodes in any feature function of an individual is substituted with one of its children.

Feature-list cross-over See text and Figure 2.

Table 1. Genetic operators.

this purpose: λ policy iteration and policy gradient method with RProp update. λ policy iteration is a family of algorithms introduced by Ioffe and Bertsekas which generalizes standard value iteration and policy iteration algorithms [9]. Value iteration starts with an arbitrary value function and at each step updates it using the Bellman optimality equation (with one step backup); the resulting optimal policy is greedy with respect to the value function[2]. On the other hand, policy iteration starts with an initial policy and generates a sequence of improving policies such that each policy is greedy with respect to the estimated value (calculated by policy evaluation) of its predecessor. λ policy iteration fuses both algorithms together with a parameter $\lambda \in (0, 1)$ by taking a λ-adjustable step toward the value of next greedy policy in the sequence [10]. Policy gradient method also works on the policy space, but approximates a parameterized (stochastic) policy directly. Starting from an initial policy, the policy parameters are updated by taking small steps in the direction of the gradient of its performance and under certain conditions converge to a local optima in the performance measure [11]. The gradient is usually estimated using Monte Carlo roll-outs. With RProp update, instead of directly relying on the magnitude of the gradient for the updates (which may lead to slow convergence or oscillations depending on the learning rate), each parameter is updated in the direction of the corresponding partial derivative with an individual time-varying value. The update values are determined using an adaptive process that depends on the change in the sign of the partial derivatives. In both RL algorithms, we represented the value function and the policy as a linear combination of feature functions, hence the parameters correspond to the coefficients of each feature function. The fitness scores of individuals are calculated by taking their average performance over a small number (around 4-10) of short learning trials using the corresponding

[2] For example, as the policy that selects in each state the action with highest estimated value.

RL algorithm. In the experiments, we observed that both algorithms converges quickly an approximately optimal policy when the basis feature functions capture the important aspects of the complicated nonlinear mapping between states and the actions.

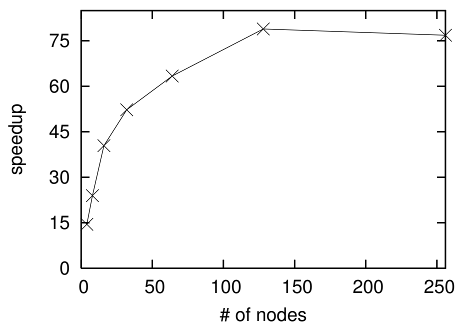


Fig. 3. Relative execution time for functions in the form of complete binary trees having depth 2-8 (averaged over 10 independent runs).

During the evaluation of a single individual, feature functions are called repeatedly, in the order of thousands, for different values of state variables to calculate the corresponding feature values. If at each call, the actual tree structure (or the linear form) of each feature function is interpreted directly by traversing the s-expression, much time is spent in auxiliary operations such as following nodes, pushing/popping values onto the stack, parsing node types etc. This overhead can easily, and in fact eventually, become a bottleneck as the size of the individuals (i.e the number of nodes) increase. Several approaches has been proposed to overcome this problem, such as directly manipulating machine language instructions as opposed to higher level expressions [12, 13] or compiling tree representation into machine code [14]. Following the work of Fukunaga et.al. [14], we dynamically generated machine code at runtime for each feature function using GNU Lightning library, and execute the compiled code at each call. GNU Lightning is a portable, fast and easily retargetable dynamic code generation library [15]. It defines a standardized RISC instruction set with general-purpose integer and floating point registers, and abstracts the user from the target CPU. As it translates code directly from a machine independent interface to that of the underlying architecture without creating intermediate data structures the compilation process is very efficient and requires only a single pass over the tree representation or linearized form of an individual. Furthermore, problem specific native operators or functions (mathematical functions etc.) can be easily called from within compiled code. Figure 3 shows relative execution time of an optimized implementation of standard approach vs. dynamically compiled code on randomly generated functions that have a complete binary tree form (i.e.

contains $2^d - 1$ nodes where d is the depth of the tree). The speed-up increases with the size of the functions, reaching a high of about 75 fold performance improvement which is substantial.

4 Experiments

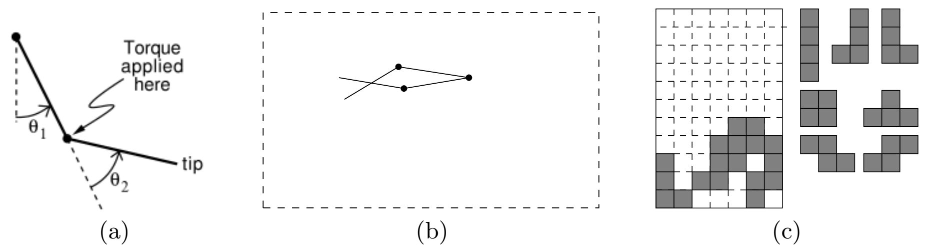


Fig. 4. (a) Acrobot, (b) multi-segment swimmer, and (c) Tetris with seven shapes that consist of 4 pieces used in the experiments.

We evaluated the proposed GP based feature discovery method on three different benchmark problems: Acrobot [16], multi-segment swimmer [17] and Tetris [2] (Figure 4). The first two problems, Acrobot and multi-segment swimmer, are non-linear control tasks with continuous state and action spaces; the number of state variables are 4 and $2n + 2$ where n is the number of segments of the swimmer, respectively. In Acrobot, there is only a single control variable, whereas in swimmer the agent has to decide on torques applied to each of $n - 1$ joints. Although it is similar in nature to Acrobot, swimmer problem has significantly more complex state and control spaces that can be varied by changing the number of segments. As the number of segments increase the problem also becomes harder. Our third benchmark problem, the game of Tetris, has discrete state and action spaces. The state variables are the following: (i) the heights of each column, (ii) the absolute difference between the heights of consecutive columns, (iii) the maximum wall height, (iv) the number of holes in the wall (i.e. the number of empty cells that have an occupied cell above them), and (iv) the shape of the current object. We used a 12×8 grid, thus in our case the number of features is 17. More detailed information about the problems is provided in the Appendix section.

In the experiments, unless stated otherwise a population consisting of 100 individuals evolved for 50 generations. We set crossover probability to 0.7 (with a ratio of 1/6 for the feature-list crossover), and the remaining 0.3 is distributed among mutation and shrinkage operators. Node mutation is given about two times higher probability than the others. There was a certain level of elitisism, 10% of best performing individuals of each generation are directly transferred to the next generation. The set of operators was $\{+, -, *, /, \sin, \cos \sqrt{\cdot}\}$ for the

Acrobot and swimmer problems and $\{+, -, *, /, \min, \max, \|-\|\}$ for the Tetris problem where $\|-\|$ denotes the absolute difference between two values. The terminals consists of the set of original state variables as given above and $\{1, 2, e\}$ where e denotes an ephemeral random constant $\in [0, 1]$. We applied policy gradient method with Rprop update for the Acrobot and swimmer problems, and λ policy iteration for the Tetris problem, respectively. In policy gradient method, an optimal baseline is calculated to minimize the variance of the gradient estimate and the policy is updated every 10 episodes. We tested with two different sets of parameters: $(\triangle_{min} = 0.01, \triangle_{ini} = 0.1, \triangle_{max} = 0.5)$ and $(\triangle_{min} = 0.002, \triangle_{ini} = 0.02, \triangle_{max} = 0.1)$. In λ policy iteration, we run 30 iterations and sampled 100 trajectories per iteration using the greedy policy at that iteration. λ is taken as 0.6. The results presented here are obtained using a single GP run for each problem. The discovered features are then tested on the same RL algorithms but with a longer training period to verify how well they perform. We averaged over 20 such test trainings.

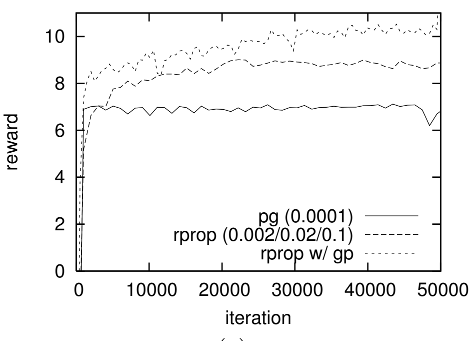
Figure 5a and Figure 6[a,c] show the testing results for the Acrobot and multiple-segment swimmer problems, respectively. In both cases, features found by GP show an improvement over original features and allow the agent to learn policies with higher return. The improvement is more evident in swimmer problem, where the agents utilizing the discovered features can learn policies that perform on average 50% better. Since candidate feature-functions are evaluated based on their average performances on short learning trials, learning speed is also faster in the initial stages as expected. The fitness values of best individuals and average fitness of the population in each generation during the feature discovery process indicate that the evolutionay search drives towards better solutions and further improvement may be possible with longer GP runs (Figure 5b and Figure 6[b,d]). We obtained inferior results with smaller population sizes (Figure 6b).

Although we used a different RL algorithm, the results for Tetris are also similar to those of Acrobot and swimmer, and show considerable improvement in terms of the performance of the resulting policies (Figure 7).

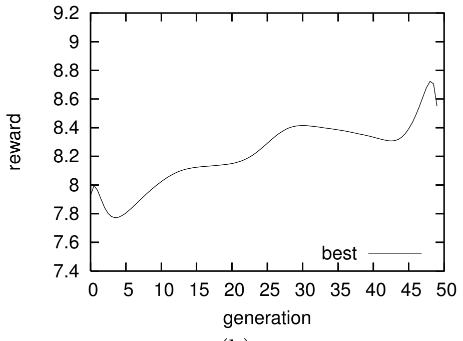
5 Conclusion

In this paper, we explored a novel genetic programming based approach for discovering useful features in reinforcement learning problems. Empirical results show that evolutionary search is effective in generating functions of state variables that when fed into RL algorithms allow the agent to learn better policies.

One important feature of proposed method is that it allows the user to guide the search and if needed incorporate domain knowledge simply by specifying the set of program primitives (i.e. ingredients of the feature functions). Furthermore, resulting feature functions are human parseable (and not hard to comprehend) which makes it possible to fine-tune and also transfer knowledge to (feature extraction process of) similar problems. This can be done either manually, or by converting them into meta functions, as in *automatically defined functions* [18],



(a)



(b)

Fig. 5. Results for Acrobot. (a) Performance of discovered features, and (b) fitness values of best individuals in each generation.

leading to a hierarchical decomposition. We pursue future research in this direction.

References

1. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press, Cambridge, MA (1998) A Bradford Book.
2. Bertsekas, D.P., Tsitsiklis, J.N.: Neuro-Dynamic Programming. Athena Scientific, Belmont, MA (1996)
3. Riedmiller, M., Peters, J., Schaal, S.: Evaluation of policy gradient methods and variants on the cart-pole benchmark. (2007) 254–261
4. Krawiec, K.: Genetic programming-based construction of features for machine learning and knowledge discovery tasks. Genetic Programming and Evolvable Machines **3**(4) (2002) 329–343
5. Smith, M.G., Bull, L.: Genetic programming with a genetic algorithm for feature construction and selection. Genetic Programming and Evolvable Machines **6**(3) (2005) 265–281 Published online: 17 August 2005.
6. Sanner, S.: Online feature discovery in relational reinforcement learning. In: Open Problems in Statistical Relational Learning Workshop (SRL-06). (2006)
7. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge, MA, USA (1992)
8. Koza, J.R., Keane, M.A., Streeter, M.J., Mydlowec, W., Yu, J., Lanza, G.: Genetic Programming IV: Routine Human-Competitive Machine Intelligence. Kluwer Academic Publishers, Norwell, MA, USA (2003)
9. Bertsekas, D., Ioffe, S.: Temporal differences-based policy iteration and applications in neuro-dynamic programming. Technical Report LIDS-P-2349, MIT (1996)
10. Scherrer, B.: Performance bounds for lambda policy iteration. (2007)
11. Sutton, R.S., McAllester, D.A., Singh, S.P., Mansour, Y.: Policy gradient methods for reinforcement learning with function approximation. In: NIPS. (1999) 1057–1063

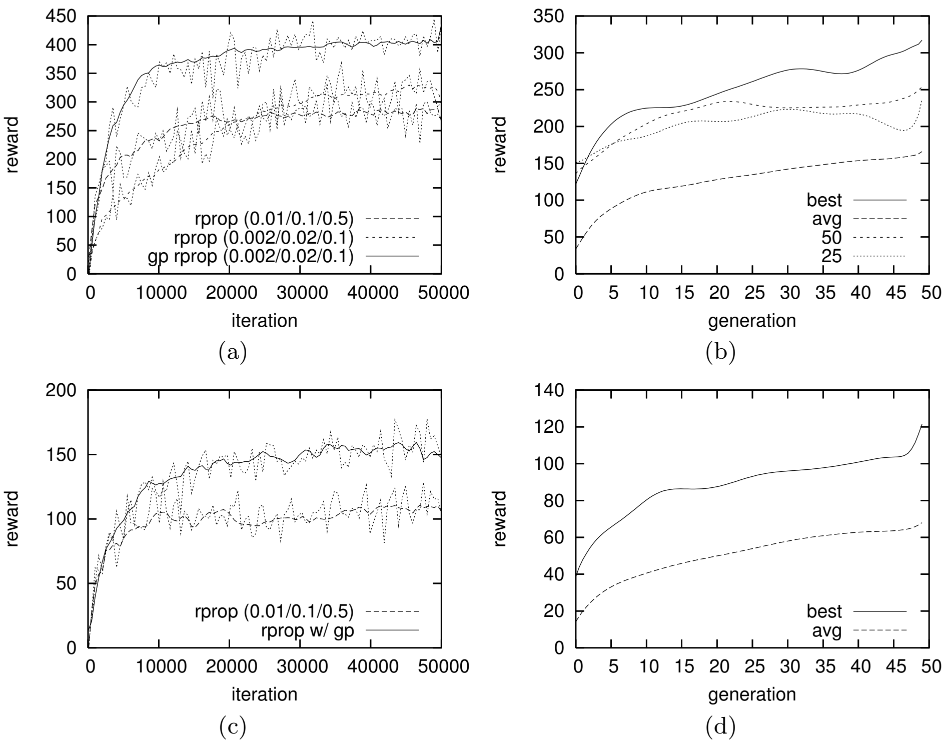


Fig. 6. Results for 3 (figures a and b) and 5 (figures c and d) segment swimmers. (a, c) Performance of discovered features, and (b, d) fitness values of best individuals and average fitness of the population in each generation. Figure b also shows the results for different population sizes.

12. Nordin, P.: A compiling genetic programming system that directly manipulates the machine code. In Kinnear, Jr., K.E., ed.: Advances in Genetic Programming. MIT Press (1994) 311–331
13. Banzhaf, W., Francone, F.D., Keller, R.E., Nordin, P.: Genetic programming: an introduction: on the automatic evolution of computer programs and its applications. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1998)
14. Fukunaga, A., Stechert, A., Mutz, D.: A genome compiler for high performance genetic programming. In Koza, J.R., Banzhaf, W., Chellapilla, K., Deb, K., Dorigo, M., Fogel, D.B., Garzon, M.H., Goldberg, D.E., Iba, H., Riolo, R., eds.: Genetic Programming 1998: Proceedings of the Third Annual Conference, University of Wisconsin, Madison, Wisconsin, USA, Morgan Kaufmann (1998) 86–94
15. GNU: Lightning (2007) Available from http://www.gnu.org/software/lightning/.
16. Spong, M.W.: Swing up control of the acrobot. In: ICRA. (1994) 2356–2361
17. Coulom, R.: Reinforcement Learning Using Neural Networks, with Applications to Motor Control. PhD thesis, Institut National Polytechnique de Grenoble (2002)
18. Koza, J.R.: Genetic programming II: automatic discovery of reusable programs. MIT Press, Cambridge, MA, USA (1994)

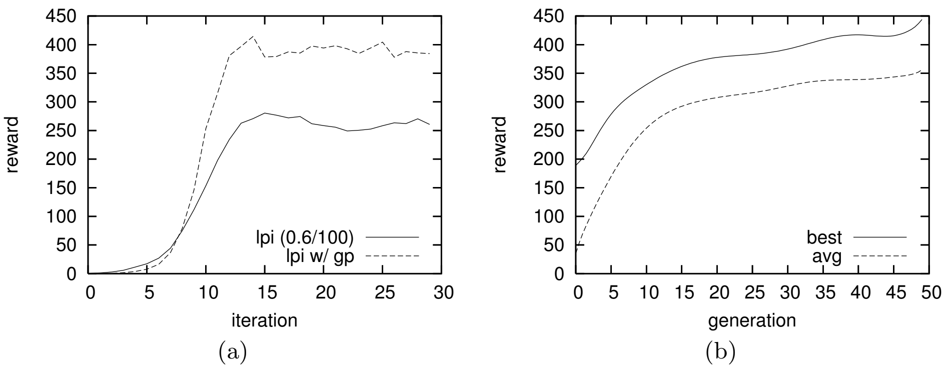


Fig. 7. Results for Tetris. (a) Performance of discovered features, and (b) fitness values of best individuals and average fitness of the population in each generation.

Appendix

Acrobot, is a well-known non-linear robot control task with dynamics similar to a swinging gymnast on a high bar [16]. The robot has two segments, *body* and *legs*, connected by a joint and the other end of the body is fixed at origin (corresponding to the hands of the gymnast) (Figure 4a). The goal is to reach the upside vertical position (which is an unstable equilibrium point) by applying torque to the joint connecting the body and the legs. The state variables are the angles of links with respect to the vertical axis and their derivatives with respect to time.

In *Swimmer*, a swimmer moving in a two dimensional pool is being simulated. The swimmer is made of n ($n \geq 3$) segments connected to each other with $n-1$ joints (Figure 4b). The goal is to swim as fast as possible to the right by applying torques to these joints and using the friction of the water. There are $2n+2$ state variables consisting of (i) horizontal and vertical velocities of the center of mass of the swimmer, (ii) n angles of its segments with respect to vertical axis and (iii) their derivatives with respect to time; the actions are the $n-1$ torques applied at segment joints. The system dynamics and more detailed information about Swimmer problem can be found in [17].

Tetris is played on a two dimensional grid which is initially empty. At each step, a random object from a set of predefined shapes appears at the top of the grid. It can be moved horizontally and rotated (in multiples of 90^o) subject to the constraints imposed by the sides of the grid. Then, it is allowed to fall until it hits the bottom of the grid or any occupied cell. This eventually leads to a wall of bricks possibly with some holes (empty cells) in it (Figure 4c). When a row is completely filled, it is removed from the grid, all rows above this row are shifted down and the player scores a point. The game ends when the top of a placed object exceeds the top of the grid. The player's objective is to maximize the score (i.e. total number of rows removed) up to the termination of the game.