Feature Discovery in Reinforcement Learning Using Genetic Programming

Sertan Girgin[1] and Philippe Preux[1,2]

[1] Team-Project SequeL, INRIA Futurs Lille
[2] LIFL (UMR CNRS), Université de Lille
{sertan.girgin,philippe.preux}@inria.fr

Abstract. The goal of reinforcement learning is to find a policy that maximizes the expected reward accumulated by an agent over time based on its interactions with the environment; to this end, a function of the state of the agent has to be learned. It is often the case that states are better characterized by a set of features. 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 a set of features, is evolved, and individuals are evaluated by their average performance on short reinforcement learning trials. The results of experiments conducted on several benchmark problems demonstrate that the resulting features allow the agent to learn better policies in a reduced amount of episodes.

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]; at each time step, the agent observes the state of the environment, chooses its action based on these observations and in return receives some kind of "reward", in other words a *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 on 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 higher reward in the future.

M. O'Neill et al. (Eds.): EuroGP 2008, LNCS 4971, pp. 218–229, 2008.

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 *direct policy search* approaches, the policy is represented as a parameterized function from states to actions and an 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 learning (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, i.e. selecting a subset of useful state variables, 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 discovery*.

The most obvious situation where the second case emerges is when the number of states is large, or infinite, and each state variable 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 cell 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 bitmap various high level features derived from it are fed into learning algorithms. On the other end of the spectrum, in some cases, additional features can be useful to improve the performance of learning. An example of this situation is presented in Fig. 1 for the classical cart-pole balancing problem. By adding sine and cosine of the pole's angle as new features to existing state variables, optimal policy can be attained much faster. A related question is, of course, given a problem what these features are and how to find them. Note that feature discovery, which will also be our main objective, is a more general problem and includes feature selection as a special case.

Usually, the set of features that are to be used instead of or together with state variables are 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", this is the *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 solely based on the observations of the agent. In this paper, we report using a Genetic Programming (GP) [3] based approach for that purpose. Our aim is to find functions of state variables (and

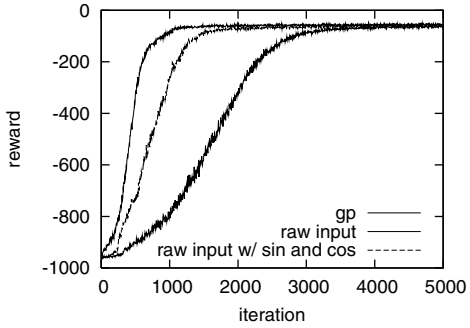

Fig. 1. In the cart-pole problem, the objective is to hold a pole, which is attached to a cart moving along a track, in upright position by applying force to the cart. The state variables are the pole's angle and angular velocity and the cart's position and velocity. Learning performance of policy gradient algorithm with Rprop update [4] when sine and cosine of the angle of the pole are added as new features, and using the features found by GP.

possibly other basis functions) that, when used as input, result in more efficient learning. Without any prior knowledge of the form of the functions, GP arises as a natural candidate for search and optimization within this context. Compared to other similar methods, such as neuro-evolutionary algorithms, it allows the user to easily incorporate domain knowledge into the search by specifying the set of program primitives. Furthermore, due to their particular representation, the resulting functions have the highly desirable property of being interpretable by humans and therefore can be further refined manually if necessary.

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

2 Related Work

Feature discovery is essentially an information transformation problem; the input data is converted into another form that "better" describes the underlying concept and relationships, and "easier" to process by the agent. As such, it can be applied as a preprocessing step to a wide range of problems and it has attracted attention from the data-mining community.

In [5], 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 are not involved in evolutionary search process. The idea behind this extension is to protect valuable features from possible harmful modifications so that the probability of accidentally abandoning promising search directions would be reduced. 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 [6] 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 data set 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 a 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 [7] recently introduced a technique for online feature discovery in relational reinforcement learning, in which 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 the *Apriori* algorithm to identify features that co-occur with a high frequency and creates a new joint feature as necessary.

The more restricted feature selection problem can easily be formulated using a fixed-length binary encoding. It has been extensively studied within the soft computing community, and in particular, various genetic algorithm (GA) based methods have been proposed. One of the earliest works is by Siedlecki and Sklansky in which GA is used to find the smallest subset of features such that the performance of a classifier meets the specified criterion [8]. We refer the interested reader to [9] and [10] for reviews of related work.

3 Feature Discovery in RL Using GP

When GP is being applied to a particular problem, there are three main issues that need to be addressed: (i) structure and building blocks (i.e. primitive functions and terminals) of the individuals, (ii) set of genetic operators, and (iii) 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. Consequently, 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*; each S-expression corresponds to a unique feature represented as a function of various arithmetic and logical operators and terminals in parenthesized prefix notation. This particular representation of an S-expression lends itself naturally to a tree structure. Given the values of state variables, the values of features can be calculated by traversing and evaluating the corresponding S-expressions. In our implementation, instead of directly using the tree forms, we linearized each S-expression in prefix-order and then concatenated them 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.

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 the 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. The first i features of the first individual are merged with the last $m - j$ features of the second individual, and the first j features of the second individual are merged with the last $n - i$ features of the first individual to generate two off-springs (Fig. 2). The generated off-springs contain a mixture of features from both parents and may have different number of features compared to them.

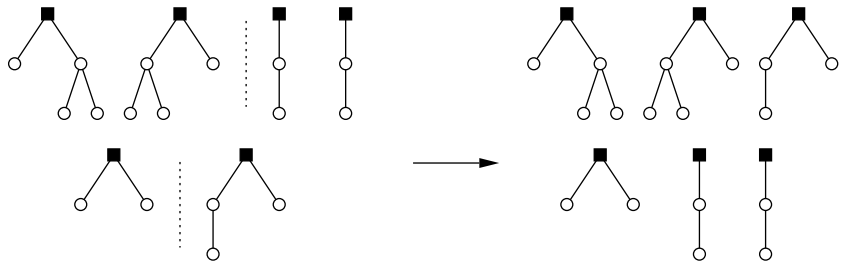

Fig. 2. Single point cross-over on feature lists of two individuals. Cross-over point is shown by vertical dashed line. The number of features represented by the off-springs differ from that of their parents.

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

Table 1. Genetic operators

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 Fig. 2.

by the agent when the corresponding feature functions are applied on a particular RL algorithm on a particular problem. In this work, we opted for two different algorithms, namely λ policy iteration and policy gradient method with RProp update; they are described in more detail in Sect. 4. 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 RL algorithm. In the experiments, we observed that both algorithms converge quickly towards an approximately optimal policy when the basis feature functions capture the important aspects of the complicated nonlinear mapping between states and actions. We also penalized feature functions according to their size to avoid very large programs, but in practice we observed that this penalization had very little effect as feature sets consisting of simpler functions tend to perform better and receive higher scores.

Accelerating the Computations

It is well known that GP is computationally demanding. In our case, which also applies in general, there are two main bottlenecks: (i) the time required to execute the program represented by an individual, and (ii) the need to evaluate many individuals in each generation.

During the evaluation of a single individual, feature functions are called repeatedly for different values of state variables in order to calculate the corresponding feature values. If at each call, the actual tree structure of each feature function (or its linear form) 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

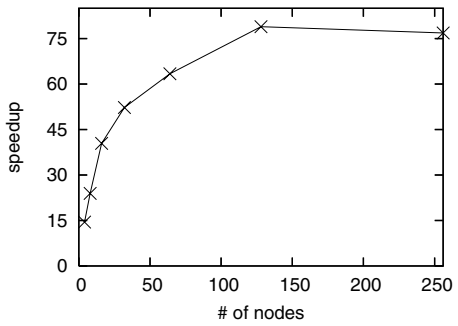

Fig. 3. Speedup for the evaluation of functions in the form of complete binary trees having depth 2-8. Each function is executed 10^6 times and the results are averaged over 10 independent runs.

individuals (i.e the number of nodes) and the number of calls (in the order of thousands or more) increase. Several approaches have been proposed to overcome this problem, such as directly manipulating machine language instructions as opposed to higher level expressions [11,12] or compiling tree representation into machine code [13]. Following the work of Fukunaga et.al. [13], we dynamically generate machine code at run-time 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 [14]; it abstracts the user from the target CPU by defining a standardized RISC instruction set with general-purpose integer and floating point registers. As code is directly translated 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[1]. Furthermore, problem specific native operators or functions (mathematical functions etc.) can be easily called from within compiled code. Figure 3 shows the speedup of an optimized implementation of standard approach compared to the 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 about 75 fold improvement which is substantial.

In GP, at each generation the individuals are evaluated independently of each other, that is the evaluation process is highly parallelizable. As such, it can be implemented efficiently on parallel computers or distributed computing systems. By taking advantage of this important property, we developed a parallel GP system using MPICH2 library [15], an implementation of the *Message Passing Interface*, and run the experiments on a Grid platform. This also had a significant impact on the total execution time.

[1] Time to compile a function of 64 nodes is around 100 microseconds on a 2.2Ghz PC.

4 Experiments

We evaluate the proposed GP based feature discovery method on three different benchmark problems: Acrobot [16], multi-segment swimmer [17] and Tetris [18]. The first two problems, Acrobot and multi-segment swimmer, are dynamical systems where the state is defined by the position and velocity of the elements of the system, and action being an acceleration which, according to Newton's law, defines the next state. These are non-linear control tasks with continuous state and action spaces; the number of state variables are respectively 4 and $2n+2$ where n is the number of segments of the swimmer. Despite their seemingly easiness, these two tasks are difficult to learn (to say the least, far from obvious). 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 and 7 different shapes that consist of 4 pieces, thus the number of features was 18.

For evaluating the individuals and testing the performance of the discovered features, we employed two different RL algorithms: λ policy iteration for the Tetris problem, and policy gradient method with RProp update for the Acrobot and swimmer problems. λ policy iteration is a family of algorithms introduced by Ioffe and Bertsekas which generalizes standard value iteration and policy iteration algorithms [2]. 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 [19]. We used the approximate version of λ policy iteration as defined in Sect. 8.3 of [18]. 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 [20]. 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

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

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 the experiments, 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 two times higher probability than the others. There is a certain level of elitism, 10% of best performing individuals of each generation are directly transferred to the next generation. The set of operators is $\{+, -, *, /, \sin, \cos \sqrt{.}\}$ for the Acrobot and swimmer problems and $\{+, -, *, /, \min, \max, \|-\|_1\}$ for the Tetris problem where $\|-\|_1$ denotes the absolute difference between two values. The terminals consist of the set of original state variables as given above and $\{1, 2, e\}$ where e denotes an ephemeral random constant $\in [0, 1]$. In the policy gradient method (Acrobot and swimmer problems), 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 (Tetris), 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 (50000 iterations for policy gradient, and 30 iterations for λ policy iteration) to verify how well they perform. We averaged over 20 such test trainings.

Figure 4a and Fig. 5[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 larger 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 evolutionary search drives towards better solutions and further improvement may be possible with longer GP runs especially in swimmer problem (Fig. 4b and Fig. 5[b,d]). Note that, the learning curves are not strictly increasing due to stochasticity in the simulations. We obtained inferior results with smaller population sizes (Fig. 5b). 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 (Fig. 6).

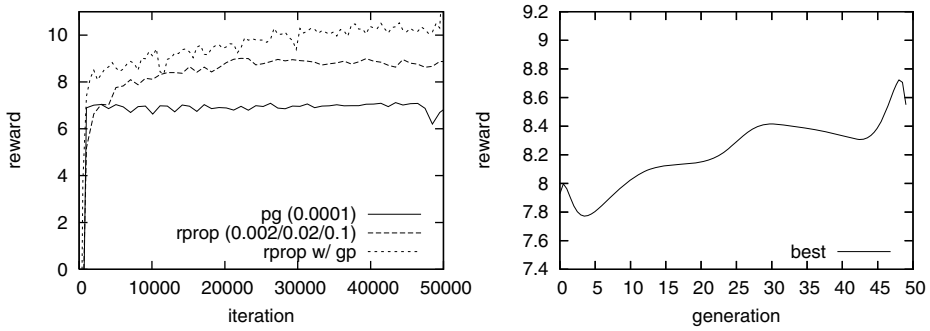

Fig. 4. Results for Acrobot. (a) Performance of discovered features, and (b) fitness values of best individuals in each generation.

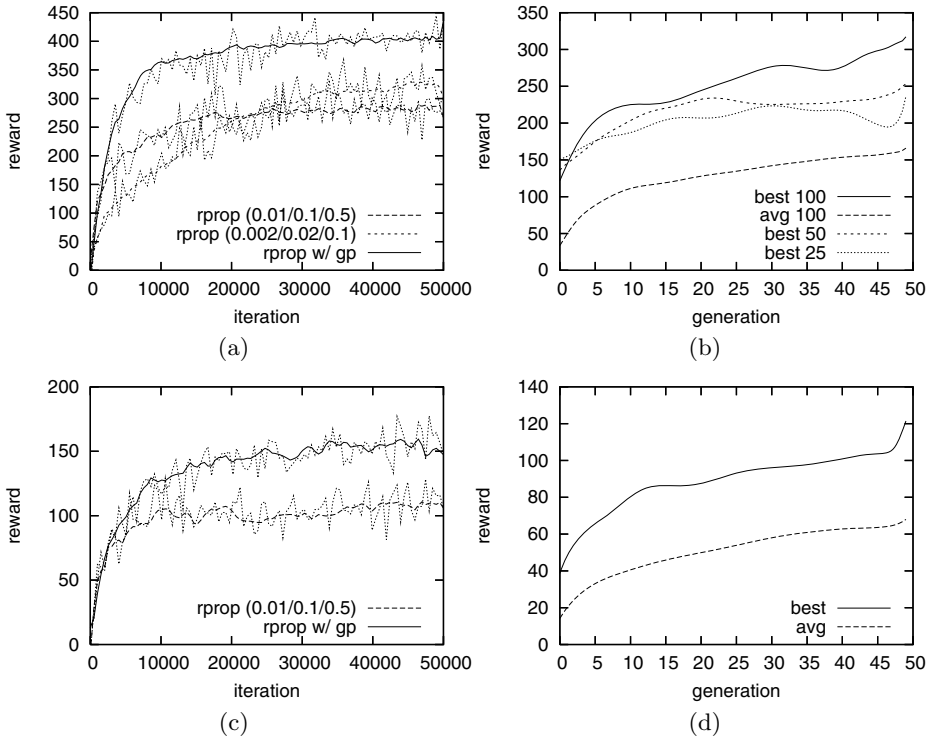

Fig. 5. Results for 3 (Fig. a and b) and 5 (Fig. c and d) segment swimmers. (a, c) Performance of discovered features, and (b, d) fitness values of best individuals and average fitness of the population in each generation. Figure b also shows the fitness values of best individuals for the population sizes of 25 and 50.

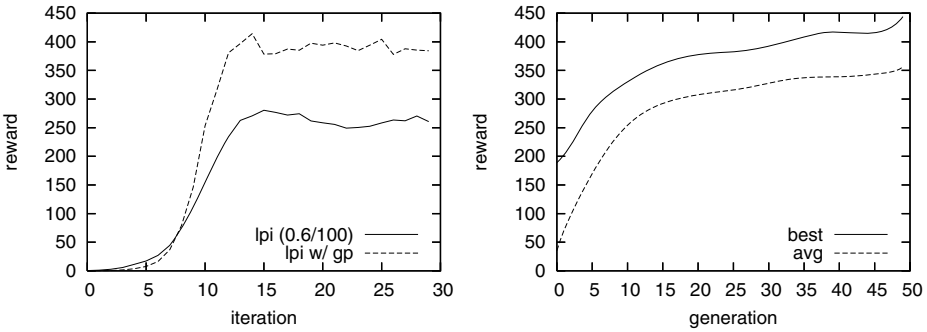

Fig. 6. Results for Tetris. (a) Performance of discovered features, and (b) fitness values of best individuals and average fitness of the population in each generation.

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. As supported by previous results in classification tasks, the approach may also be applicable in supervised settings by changing the learning algorithm used for evaluating the individuals. However, care must be taken to choose algorithms that converge quickly when supplied with a "good" state representation.

One important point of the 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 readable by humans (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* [21], leading to a hierarchical decomposition. We pursue future research in this direction.

Acknowledgment

Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, an initiative from the French Ministry of Research through the ACI GRID incentive action, INRIA, CNRS and RENATER and other contributing partners (see https://www.grid5000.fr).

References

1. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press, Cambridge (1998) (A Bradford Book)
2. Bertsekas, D., Ioffe, S.: Temporal differences-based policy iteration and applications in neuro-dynamic programming. Technical Report LIDS-P-2349, MIT (1996)
3. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge (1992)
4. Riedmiller, M., Peters, J., Schaal, S.: Evaluation of policy gradient methods and variants on the cart-pole benchmark, pp. 254–261 (2007)
5. Krawiec, K.: Genetic programming-based construction of features for machine learning and knowledge discovery tasks. Genetic Programming and Evolvable Machines 3(4), 329–343 (2002)
6. Smith, M.G., Bull, L.: Genetic programming with a genetic algorithm for feature construction and selection. Genetic Programming and Evolvable Machines 6(3), 265–281 (2005)
7. Sanner, S.: Online feature discovery in relational reinforcement learning. In: Open Problems in Statistical Relational Learning Workshop (SRL-2006) (2006)
8. Siedlecki, W., Sklansky, J.: A note on genetic algorithms for large-scale feature selection. Pattern Recogn. Lett. 10(5), 335–347 (1989)
9. Martin-Bautista, M.J., Vila, M.A.: A survey of genetic feature selection in mining issues. In: Proceedings of the 1999 Congress on Evolutionary Computation CEC 1999, vol. 2, p. 1321 (1999)
10. Hussein, F.: Genetic algorithms for feature selection and weighting, a review and study. In: Proceedings of the Sixth International Conference on Document Analysis and Recognition, Washington, DC, USA, p. 1240. IEEE Computer Society, Los Alamitos (2001)
11. Nordin, P.: A compiling genetic programming system that directly manipulates the machine code. In: Kinnear Jr, K.E. (ed.) Advances in Genetic Programming, pp. 311–331. MIT Press, Cambridge (1994)
12. 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 (1998)
13. Fukunaga, A., Stechert, A., Mutz, D.: A genome compiler for high performance genetic programming. In: Genetic Programming 1998: Proceedings of the Third Annual Conference, University of Wisconsin, Madison, Wisconsin, USA, pp. 86–94. Morgan Kaufmann, San Francisco (1998)
14. G.N.U.: Lightning (2007), `http://www.gnu.org/software/lightning/`
15. Laboratory, A.N.: Mpich2 (2007), `http://www-unix.mcs.anl.gov/mpi/mpich2/`
16. Spong, M.W.: Swing up control of the acrobot. In: ICRA, pp. 2356–2361 (1994)
17. Coulom, R.: Reinforcement Learning Using Neural Networks, with Applications to Motor Control. PhD thesis, Institut National Polytechnique de Grenoble (2002)
18. Bertsekas, D.P., Tsitsiklis, J.N.: Neuro-Dynamic Programming. Athena Scientific, Belmont, MA (1996)
19. Scherrer, B.: Performance bounds for lambda policy iteration (2007)
20. Sutton, R.S., McAllester, D.A., Singh, S.P., Mansour, Y.: Policy gradient methods for RL with function approximation. In: NIPS, pp. 1057–1063 (1999)
21. Koza, J.R.: Genetic programming II: automatic discovery of reusable programs. MIT Press, Cambridge (1994)