Genetic Programming for Reward Function Search

Scott Niekum, Andrew G. Barto, *Fellow, IEEE*, and Lee Spector

***Abstract*—Reward functions in reinforcement learning have largely been assumed given as part of the problem being solved by the agent. However, the psychological notion of *intrinsic motivation* has recently inspired inquiry into whether there exist alternate reward functions that enable an agent to learn a task more easily than the natural task-based reward function allows. This paper presents a genetic programming algorithm to search for alternate reward functions that improve agent learning performance. We present experiments that show the superiority of these reward functions, demonstrate the possible scalability of our method, and define three classes of problems where reward function search might be particularly useful: distributions of environments, nonstationary environments, and problems with short agent lifetimes.**

***Index Terms*—Genetic programming, intrinsic motivation, reinforcement learning.**

I. Introduction

THE reinforcement learning (RL) paradigm typically assumes a given reward function that specifies the learning problem being solved by the agent. Although RL theory and algorithms do not depend on the nature of the reward function, in practice the reward function is a critical parameter of the RL process that can significantly influence the rate of learning and, indeed, whether learning is successful or not in improving system performance. It is widely recognized that learning can be improved by manipulating the reward function. Ng, Harada, and Russell [1], for example, showed that the learning rate can be improved through a special class of "shaping" reward adjustments that do not disturb the optimal policy. However, they did not specify a mechanism for designing such beneficial adjustments. Even when guided by such a theoretical result, RL practitioners have found that defining good reward functions can be a very nontrivial task, and they often resort to searching "by hand" to find a reward function that produces acceptable results.

Singh, Lewis, and Barto [2], [3] recently advanced a general computational framework for reward that places it in an evolutionary context, formulating a notion of an *optimal reward function* given a fitness function and some distribution of environments. They demonstrated that an entire spectrum of reward functions can exist that allow a learning system to perform much better than it can when using what appear to be the most natural reward functions for a given problem. Even in simple problems, reward functions can exist that enable significantly faster learning than do natural task-specific reward functions. These alternate reward functions can be related to the problem in counterintuitive ways and can require precise tuning to achieve their benefits. Some components of these high-performing reward functions are analogous to what in psychology are called *intrinsic rewards*, meaning rewards that motivate activity "for its own sake" instead of for its problem-specific consequences. As more fully examined in the article by Singh, Lewis, and Barto [3] in this volume, such reward functions exploit complex causal chains involving recurring regularities that, in the natural world, can only be discovered by the evolutionary process.

Searching by hand is not likely to yield high-performing reward functions. To find good reward functions, Singh *et al.* [2] used automated quasiexhaustive searches over small, discretized sets of possible scalar reward functions defined on subsets of the state variables of the original problems. Although their objective was to examine questions about the nature and source of reward functions and not to advocate automated search for good reward functions as a practical methodology, their results suggest that such searches may indeed be beneficial if carried out efficiently.

In this paper, we employ a genetic programming algorithm to search for alternate reward functions with the aim of demonstrating that such a search may be worthwhile when an agent faces distributions of related environments, nonstationary environments, or problems with a limited agent lifetime. Reward functions discovered by this process are able to provide the agent with both extrinsic and intrinsic motivation, rewarding events that are causally both proximal and distal to agent success. Experiments validate that our search method finds superior reward functions and is minimally affected by increasing the dimensionality of the state space.

II. Background

A. Reinforcement Learning (RL)

The RL paradigm [4] typically models a finite-state problem faced by the agent as a finite Markov decision process (MDP), expressed as a tuple $M = \langle S, A, P, R \rangle$, where S is the set of environment states the agent can observe, A is the set of actions that the agent can execute, $P(s, a, s')$ is the probability that the environment transitions to $s' \in S$ when action $a \in A$ is taken in state $s \in S$, and $R(s, a, s')$ is the expected scalar reward given to the agent when the environment transitions to state s' from s after the agent takes action a. For simplicity and without significant loss of generality, in this paper we only examine reward functions of the form $R(s')$ that reward the agent solely based on the state it enters.

Manuscript received February 14, 2010; revised April 28, 2010; accepted May 20, 2010. Date of publication May 27, 2010; date of current version June 30, 2010. This work was supported in part by the National Science Foundation under NSF Grant #CNS-0619337. The work of A. G. Barto was supported by the Air Force Office of Scientific Research under Grant FA9550-08-1-0418. The work of L. Spector was supported by the Hampshire College's Institute for Computational Intelligence.

S. Niekum and A. G. Barto are with the Department of Computer Science, University of Massachusetts, Amherst, MA 01003 USA (e-mail: sniekum@cs.umass.edu; barto@cs.umass.edu).

L. Spector is with the School of Cognitive Science, Hampshire College, Amherst, MA 01002 USA (e-mail: lspector@hampshire.edu).

Digital Object Identifier 10.1109/TAMD.2010.2051436

B. Optimal Reward Functions

In the approach taken by Singh *et al.* [2], [3], learning agents, and therefore their reward functions, are evaluated according to their *expected fitness* given an explicit fitness function and some distribution of environments of interest. Specifically, they define an *optimal reward function* in terms of a distribution over MDP environments in some set $\mathcal{E}$, in which agents should perform well (in expectation), and a space $\mathcal{R}$ of reward functions feasible for these MDPs. A specific reward function $R \in \mathcal{R}$ and a sampled MDP $E \in \mathcal{E}$ produce a history h generated by an agent learning in environment E while using the reward function R. A given *fitness function* F produces a scalar evaluation $F(h)$ for all such histories h. An optimal reward function $R^* \in \mathcal{R}$ is a reward function that maximizes expected fitness over the distribution of environments. In this formulation, the problem to be solved is specified by the designer through the fitness function, leaving the reward function as an adjustable parameter of the agent.

A *fitness-based reward function* assigns a positive reward to states that directly increase fitness, negative reward to states that directly decrease fitness, and uniform reward elsewhere. Such reward functions are the traditional "obvious" representation of goals by means of a reward function. Note that not all fitness functions admit such simple fitness-based reward functions since they evaluate entire histories. (However, for simplicity, all the fitness functions we use in this work do admit simple fitness-based reward functions).

At first glance, it is not clear what is gained by this formulation. How is a fitness function different from a reward function? It may seem no easier to design a good fitness function than a good reward function. We can appeal to the analogy with biology, in which the evolution of animal reward systems is guided by the reproductive fitness to which these systems contribute. An animal learning under the influence of its reward system can exploit regularities in its local environment, but only the evolutionary process can exploit regularities that have appeared across ensembles of ancestral environments. This wider reach of evolution—the outer-loop search process—creates reward functions that balance multiple subtle factors to enable learning and behavior that is robust with respect to environmental nonstationarity. In some cases, these evolved reward functions include rewards that we would call intrinsic rewards. Furthermore, since a fitness function evaluates entire life histories without necessarily being decomposable into separate influences of states and actions, it can often represent the true task objective more directly than can a reward function (although our experiments do not illustrate this advantage). For more discussion of these central issues, see [2], [3], and [5].

C. Q-Learning

Given a reward function, the agent's objective is to maximize a measure of the cumulative reward it receives. Commonly, this measure is the *discounted cumulative reward* , defined as $\sum_{i=0}^{\infty} \gamma^i r_{t+i+1}$, where r_t is the reward received at time t and $0 \leq \gamma < 1$ is a discount factor that specifies to what degree the agent prefers immediate rewards to future ones. Q-learning [6] is an RL algorithm that iteratively computes "Q-values" intended to closely approximate optimal state–action values, $Q^*(s,a)$, that give for each state-action pair (s,a) the expected discounted cumulative reward to be received if the agent takes action a in state s, and follows an optimal policy thereafter. Depending on the accuracy of the approximation, choosing for each state the action having the largest Q-value can yield high cumulative reward over time. These values are determined incrementally using the following update rule at each time t:

$$\hat{Q}(s_t,a_t) \leftarrow \hat{Q}(s_t,a_t) + \alpha \left[R(s_t,a_t,s_{t+1}) + \gamma \max_{a \in A} \hat{Q}(s_{t+1},a) - \hat{Q}(s_t,a_t) \right]$$

where $\hat{Q}$ is the current approximation of the optimal state-action value function Q^*. To balance exploration and exploitation, our experiments use ϵ-greedy action selection, meaning that at each time step, the optimal action is selected with probability $(1-\epsilon)$ and a random action is selected with probability ϵ.

D. Genetic Programming

Genetic programming is a technique for searching a space of computer programs using genetic algorithms [7] in which the "genotypes" are executable programs. In the most standard version of the technique, one begins with a fixed-sized population of randomly generated programs, constructed from functions and values chosen for the problem domain, and then iteratively evaluates the quality of each program and produces a new population by randomly varying and recombining the better programs in the current population [8]. Each step of this iterative process, in which an entire population is evaluated and a new population is created, is called a "generation." Because the genotypes are executable programs, they are generally evaluated by executing them, often on a fixed set of "fitness case" inputs, while taking any necessary measures to ensure execution safety (e.g., by defining a nonerror return value for division by zero).

In the most standard version of the technique, the programs in the population are expressed as Lisp-like symbolic expressions, although we use a variant of this by operating on symbolic expressions for a stack-based virtual machine (see below). Variation by "mutation" is implemented as replacement of random subexpressions by newly generated random subexpressions, while recombination ("crossover") is implemented as the swapping of subexpressions between two programs. Further detail on genetic programming techniques can be found in survey texts [9] and [10].

III. Evolving Reward Functions

Task-based reward functions usually used in RL are generally straightforward, taking the form of simple state-to-reward mappings. However, to facilitate reward function search, a representation is needed that is compact, expressive, and naturally capable of accommodating search operations. Even over a small state set, the space of all possible reward functions is infinite, and we can only speculate about what characteristics such a space may have. To be effective in the general case, a reward function representation and search algorithm are required that allow for efficient search over large spaces, that avoid local minima, and that are compatible with both discrete and continuous state spaces. For similar representation and search tasks, others have used evolutionary methods on neural networks [11] or searched over linear combinations of features, but these methods are difficult to analyze in the former case and are relatively inflexible in the latter.

We propose applying genetic programming methods to search for improved reward functions. These reward functions operate over the entire state space of an RL problem and are evolved to quickly and automatically identify relevant variables and features of the problem. This may allow an agent using such a reward function to outperform an agent that uses a straightforward fitness-based reward function. The use of genetic programming methods may alleviate the difficulty of scaling reward function search and provide a natural way to search through a very expressive space of functions. Furthermore, genetic programming is a global search method that can perform well in the presence of local minima and when little is known about the characteristics of the search space. For this task, we selected PushGP, a stack-based genetic programming system built on the Push programming language [12], [13].

PushGP has a number of features that make it appealing for representing and evolving reward functions. Its stack-based architecture allows for the use of multiple data types without concern for the type-correctness of code following evolutionary operations. Push programs can be analyzed relatively easily, as PushGP supports an automatic code simplification algorithm, a full set of familiar logical and programmatic constructs, and a built-in system that allows for the natural emergence of code modularity. In principle, a Push program can effectively represent any computable function, providing us with unrestricted power to solve arbitrarily difficult problems. Empirically, PushGP has proven to be effective in producing human-competitive results in multiple domains, and in one case, defeated other search methods by several orders of magnitude in a mathematical domain dealing with finite algebras [14].

However, our aim in this article is not to find the *best* search method, but to propose a *good* search method that is efficient enough to allow us to search for and explore the properties of alternate reward functions. PushGP reasonably meets our criteria of expressibility, efficiency, and local-minima avoidance.

IV. Experiments

Our methodology is evaluated using the Hungry–Thirsty domain [2], shown in Fig. 1. At each time step, the agent can move one square north, south, east, or west, or choose from one of two special actions, "eat" or "drink." In each instance of this domain, inexhaustible food and water locations are chosen from two of the four corners and held fixed for the agent's lifetime (the food and water cannot be colocated, resulting in 12 possible environmental configurations). The agent's objective is to eat as much food as possible by using the "eat" action at the food location, but the agent's eat action fails when it is thirsty (or if the agent is not at the food location). Drinking at the water location makes the agent not-thirsty, but at each time step thereafter, the agent becomes thirsty again with probability 0.1. When the agent successfully eats, it becomes not-hungry for one time step, and is hungry again thereafter. For each time step in the agent's lifetime that it is not-hungry, its fitness score increases by one.

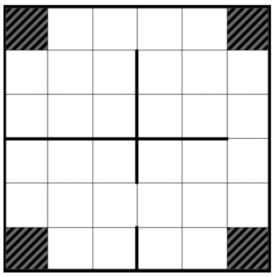
Fig. 1. Hungry–Thirsty domain. Thick lines are walls, striped squares denote possible food or water sites.

Therefore, $F(h)$ gives the total number of time steps in which the agent was not-hungry throughout its history, or lifetime, h, which we limit to a fixed finite number of time steps. In what follows, we call this score *cumulative fitness*. As a corresponding fitness-based reward function, we provide a positive reward of 100 when the agent is not-hungry and -1 otherwise, to encourage exploration. We intentionally chose this small domain to simplify the analysis of our methodology and to allow us to relate our results to those of Singh *et al.* [2].

In our experiments, generations of agents are evaluated based on their fitness score across a distribution of environments within this domain. Thus, a reward function may emerge from the evolutionary process that maximizes an agent's fitness across this distribution, capturing the most salient common features of these environments.

To accelerate the evolutionary process, we evolved reward function adjustments rather than reward functions directly. An agent's final reward function is created by adding the evolved reward function to a given fitness-based reward function. Thus, we are evolving reward functions from a reasonable starting point so that PushGP does not have a harder job than necessary. Note that evolving reward functions "directly" is a special case of evolving reward function adjustments where the starting point is the zero reward function. Also, hand-designing a starting point is the same as creating a simple fitness-based reward function, which researchers already do when formulating an RL problem. Therefore, we are not requiring knowledge or effort that goes beyond what is customarily brought to the task of formulating an RL problem.

For each run of these experiments, a starting population of 1000 random reward functions is initialized. Each reward function in the population is evaluated by assigning it to an agent and evaluating the agent's fitness over 75 000 time steps in the Hungry–Thirsty domain. This is repeated for 120 agent lifetimes (over a distribution of environments that vary per experiment, resetting learning after each), from which an average fitness score for the reward function is computed and used to determine which reward functions are selected for mutation, crossover, and duplication to produce the next generation. This process is repeated for 50 generations and the best reward function is recorded for later use. Finally, this entire experimental process is repeated 50 times and the results are averaged to make all our final graphs. The statistical significance of the comparisons of the various reward functions is determined through paired t-tests that assume unequal variances.

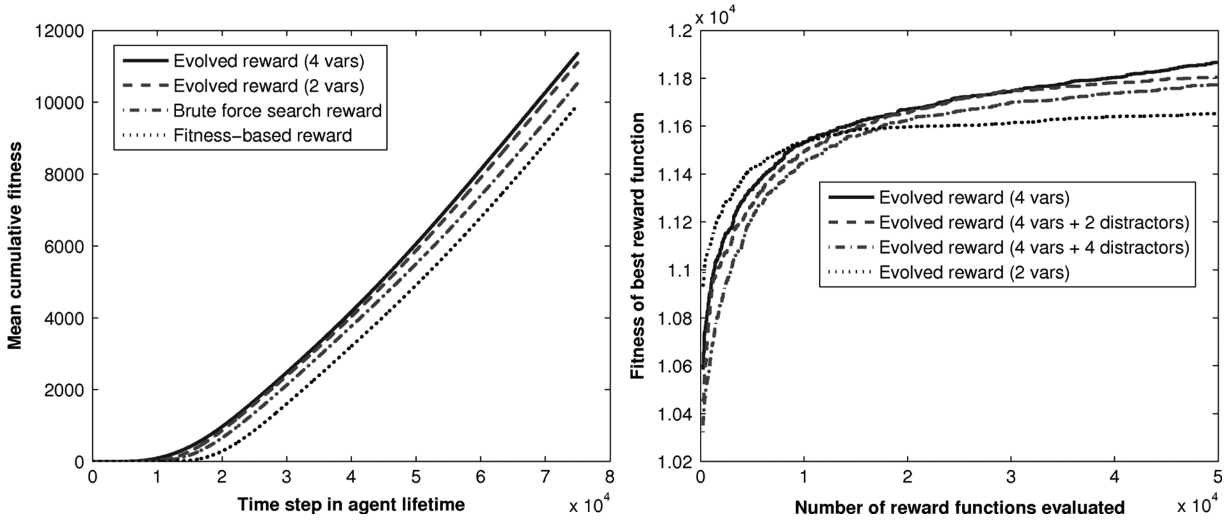

Fig. 2. Agent fitness (left) and evolutionary progress (right) over a distribution of environments.

We used a Java implementation of PushGP called Psh, written by Jon Klein.[1] For these experiments, we chose a very limited subset of Push's full instruction set to simplify our experiments and analysis. We used only arithmetic floating point operators, random numbers, swapping, and duplication. We also added one special instruction per input variable that pushes the variable directly onto the float stack, so that PushGP can access each state variable on demand. More specifically, FLOAT.+, FLOAT.-, FLOAT.*, and FLOAT.% pop the top two elements from the float stack and perform addition, subtraction, multiplication, and modulo, respectively. (When order matters, Push executes <second item on the stack> *operator* <top item on the stack>). FLOAT.DUP duplicates the top item on the float stack and pushes it. FLOAT.SWAP swaps the top two items on the float stack. STATE n pushes the $n\mathrm{th}$ state variable onto the float stack. If there are not enough items on a stack to complete an operation, it simply does nothing.

Future work may use PushGP's full power by including logical constructs, looping, and modularity operators. For our purposes, however, this simple set of operators allowed us to find improved reward functions and demonstrated the power of evolutionary search over a large space of programs. In this configuration PushGP functions much like the simplest, original genetic programming systems [8].

Each RL agent learned using ϵ-greedy Q-learning ($\alpha = 0.1$, $\gamma = 0.99$, $\epsilon = 0.1$). The initial Q-values were initialized randomly to small values in the range of $[-0.001,\ 0.001]$. We represented the Q-value function as a simple lookup table, as to not confound our results with the representational difficulties sometimes encountered with function approximation.

Three experiments are presented that each define a class of problems for which intrinsic rewards may be particularly useful, or even necessary. We examine cases in which the agent faces distributions of related environments, nonstationary environments, and problems with a significantly shortened agent lifetime.

[1]Source code available at http://github.com/jonklein/Psh

V. Results

A. Experiment 1: Distribution of Environments

In solving real-world problems, an agent may face problem instances selected from a distribution of environments that share common features, but where the specifics of any particular environment are not known *a priori*. It is desirable for an agent to be able to learn quickly in any of these environments by taking advantage of known information about the environment distribution. When populations of agents are exposed to such a distribution of environments, an evolved reward function may emerge that allows an agent to learn quickly on any given problem within the distribution. To test the ability of PushGP to evolve such functions, we evaluated reward functions over 120 samples from a uniform distribution over the 12 possible food and water arrangements (resetting learning after each trial) in the Hungry–Thirsty domain. We compared our results to those of Singh *et al.* [2] on this same distribution.

The left panel of Fig. 2 compares the average cumulative fitness over an agent's lifetime when using the fitness-based reward functions, the best brute-force search reward function from Singh *et al.* [2], and reward functions evolved by PushGP. In the two-variable case, the evolved reward function was given as input the hunger and thirst status of the agent (the same two state variables used in [2]). Here, the evolved function performs somewhat better; the evolutionary search did not have to search over a discretized space, and therefore found a more refined solution. In the four variable case, we provided the reward function with two additional state variables, the x and y coordinates of the agent's location, and the evolutionary process produced a slightly better reward function. This shows that an evolutionary search algorithm can extract salient features of a problem, even when searching over a large space of functions, affording better performance than the simpler brute-force method. All comparisons on this graph are highly statistically significant after time step 15 000 ($p < 4 \times 10^{-4}$).

The right panel of Fig. 2 shows the fitness of the best reward function evaluated so far as a function of time during the

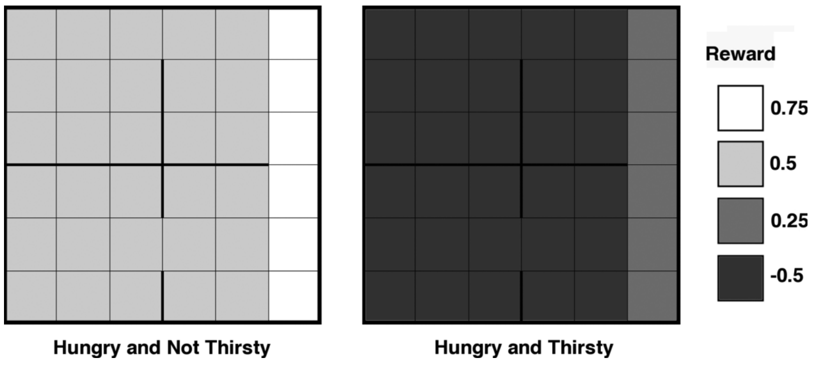

Fig. 3. Evolved reward function from the Hungry–Thirsty domain.

evolutionary process (where each generation is 1000 evaluations on the x-axis, with the initial 250 evaluations removed to ease scaling and readability). Despite the fact that adding additional relevant variables significantly increases the size of the search space, we see that the four-variable curve only lags behind the two-variable curve until around the tenth generation, and then shows better performance thereafter (after the 25th generation, $p < 10^{-5}$). In the other two cases, we added additional "distractor" variables (two and four, respectively) that were uniformly distributed integers between 0 and 4. It appears that evolutionary search quickly learned to ignore these irrelevant variables, as evolutionary progress slowed relatively little with their addition. In fact, after the 15th generation, the difference between the standard four-variable case and the two-distractor case is no longer statistically significant ($p > 0.08$). Furthermore, by the end of evolution, there is no statistically significant difference between the two-distractor and four-distractor cases ($p > 0.27$). It should be noted that PushGP is not sensitive to the possible domains of any of the input variables; each variable is just seen by PushGP as a floating point number that can take on any possible value. Thus, each time a variable was added, an entire dimension was added to the problem, yet the evolutionary process was largely unaffected. This suggests that evolutionary methods such as PushGP may generalize well to larger, high-dimensional problems.

Fig. 3 shows the best evolved reward function from one of our runs (in the four-variable case) at each (x, y) location when the agent is hungry and not-thirsty (left) and hungry and thirsty (right). The first noticeable feature is that there is a penalty for being thirsty, ranging from 0.5 to 1, depending on the agent's location. This is similar in nature to the thirst penalty that was discovered by the brute-force method in Singh *et al.* [2] and by our method in the two-variable case. However, in our reward function, there is also a corridor on the eastern side of the domain that is rewarded more highly than the rest of the locations. This corridor is significant because in eight of the twelve possible food and water configurations, the agent must pass through this corridor to move between the food locations. Here, evolutionary search finds a common feature of the environments that may not be immediately apparent to a human designer. Furthermore, the evolutionary process carefully tunes the magnitude of the weights so that they assist learning, rather than distract the agent, something a human designer cannot do easily. Thus, the evolved reward function provides the agent with a carefully balanced *intrinsic motivation* to drink water and spend time in the eastern corridor, even though there is no direct fitness increment associated with doing so, thereby improving learning speed and cumulative fitness of the agent.

Examining the actual Push program that produced this reward function yields some interesting insights about PushGP. The program is reproduced here without parentheses, which have no effect on program execution in the present context[2]:

FLOAT.- FLOAT.* FLOAT.SWAP STATE2 FLOAT.+ FLOAT.* FLOAT.SWAP FLOAT.+ FLOAT.* FLOAT.* STATE0 FLOAT.* FLOAT.% STATE2 4.5 FLOAT.% STATE2 FLOAT.% FLOAT.DUP STATE1 STATE2 FLOAT.% FLOAT.+ FLOAT.* STATE1 FLOAT.- 0.5 FLOAT.+ FLOAT.DUP

Here, STATE0 is hunger status, STATE1 is thirst status, STATE2 is x-position, and STATE3 is y-position. The following is a simplified version of the program produced by removing the operators that do not do anything or that do not affect the final reward value:

STATE2 4.5 FLOAT.% STATE2 FLOAT.% FLOAT.DUP STATE1 STATE2 FLOAT.% FLOAT.+ FLOAT.* STATE1 FLOAT.- 0.5 FLOAT.+

While we can do this simplification for analysis, it should be noted that the "garbage" instructions may be important for the evolutionary process, as they can be activated by a small change in the program.

When executed, this program leaves the result of the following expression (where thirst status is denoted by t and x-location coordinate by x) on top of the float stack as the reward (more correctly, this is the reward modifier that is later added to the fitness-based reward)

$$[[(t\%x) + (x\%4.5\%x)] \times (x\%4.5\%x)] + 0.5 - t.$$

The most obvious feature of this expression is the thirst penalty of $-t$ and the base reward of 0.5. We also see that the phrase $(x\%4.5\%x)$ is repeated twice, via the FLOAT.DUP instruction. Since we did not provide PushGP with an "if" instruction, it finds a creative way to check if the agent is in the eastern corridor: $(x\%4.5\%x)$ evaluates to 0.5 when $x = 5$ and evaluates to

[2]Parentheses are important in PushGP because they delimit the subprograms that are targets of mutation and crossover operations. They can also affect execution when programs include code-manipulation instructions, but no such instructions were used in the experiments described here.

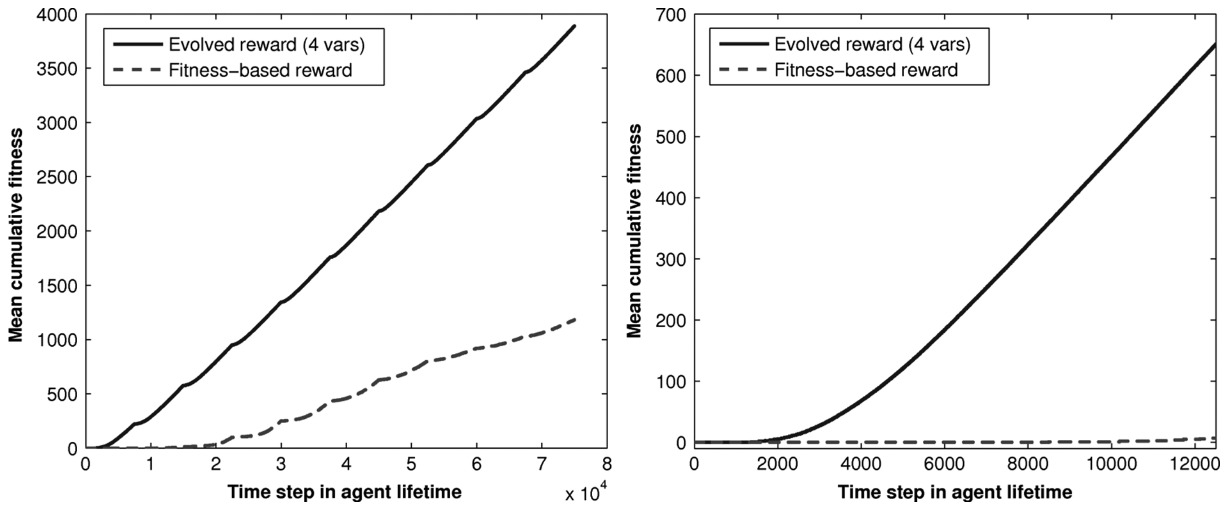

Fig. 4. Agent fitness on nonstationary (left) and short lifetime (right) problems.

0 otherwise (FLOAT.% is implemented to allow fractional return values). Thus, an additional reward is given to the agent if $x = 5$, and the thirst penalty is slightly adjusted as well. We see that PushGP finds a sensible program and represents it efficiently by reusing a useful calculation via FLOAT.DUP. However, this should not be confused with the built-in support Push has for code modularity that we did not enable for these experiments.

To verify that our results were not highly dependent on the subset of instructions we selected to use, we also ran this experiment with 18 instructions rather than seven, including if-statements, boolean stack operators, and simple code manipulation. We hypothesized that the extra instructions would increase the search space and slow down the evolutionary process (although this larger subset might allow a better solution to be found in the limit). However, by the 30th generation, the evolutionary process produced a solution using this larger subset that is not significantly different from our original result ($p > 0.19$). This suggests that there may be only a small initial time penalty for using a larger set of instructions. Thus, it may be worthwhile to do so since these additional instructions may provide additional descriptive power and solution quality.

B. *Experiment 2: Nonstationary Environment*

In this experiment, we examined the class of *nonstationary* problems in which the state–transition function, P (i.e., the environmental dynamics) changes during an agent's lifetime. This experiment modifies the Hungry–Thirsty domain such that the locations of the food and water change every 7500 steps, for a total of 10 subproblems over a lifetime. In a manner similar to when the agent faces problems sampled from a distribution, we expect the evolved reward function to identify salient common features between the possible environmental configurations and adapt to the additional demands of nonstationarity. In a nonstationary environment, the agent must be able to learn the correct policy quickly, which requires unlearning an incorrect policy quickly when the environment changes. It should be emphasized that shaping rewards [1] (or equivalently, initial value functions [15]) are of little use on nonstationary problems since they are quickly "learned away," that is, are reduced to zero in the course of learning. Evolved reward functions, on the other hand, are permanent and provide a consistent advantage across environmental changes.

The left panel of Fig. 4 compares the average cumulative fitness over an agent's lifetime when using the fitness-based reward function and functions evolved using PushGP. We see that an agent using an evolved reward function greatly outperformed an agent using the fitness-based reward function ($p < 10^{-30}$ after step 15 000). The nonstationarity of the problem magnified this improvement as compared to the improvement shown over a distribution of environments in the first experiment. The fitness-based curve shows almost no fitness gains for the first two subproblems, but thereafter shows some degree of learning on each subproblem, followed by a performance hit after each switch. This suggests that some of the Q-values learned on the early subproblems are reusable in the later problems; otherwise the agent would never gain any fitness, given its performance on the early problems. However, as time goes on, the maximum slope of the curve during a subproblem (the rate of fitness gain) decreases, and the curve becomes more linear. It appears that the agent does not have enough time to completely learn each subproblem, so as time goes on, the Q-values tend toward the mean of the values appropriate for the various subproblems, reducing performance on any one of them. By contrast, we see that the agent using the evolved reward function takes a small, consistent performance hit every time the problem changes but otherwise quickly gains fitness at a rapid rate. This suggests that the reward function is tuned to help the agent both learn and unlearn quickly in the presence of a nonstationary environment.

Upon examination, the evolved reward functions for nonstationary environments look somewhat different from those in the first experiment. The reward functions leading to the highest performance exhibit a strong gradient of negative reward that encourages the agent to always move east when possible. This suggests that the evolutionary process is optimizing for particular problems (when food and water are both on the eastern side) at the expense of others because the average payoff for doing so is greater than the losses incurred. This is one possible danger when designing a fitness function—you get exactly what you ask for. There is no penalty for the agent failing completely at one of the subtasks if it can make up for it later, thereby increasing the mean fitness. An alternate fitness function may give the agent a fitness of zero if it does not perform with some threshold level of competence on every task, depending on what the designer desires.

C. Experiment 3: Short Agent Lifetime

Finally, we examine the properties of evolved reward functions on a problem with a short agent lifetime. In such a scenario, an agent might never reach a fitness-incrementing state or be able to learn a good policy in the allotted time. We modified the original experiment such that the agent only lived for 12 500 time steps rather than 75 000, significantly reducing the amount of time available to learn the task. The right panel of Fig. 4 compares the average cumulative fitness over an agent's lifetime when using the fitness-based reward function and functions evolved using PushGP. Again, we see a magnified improvement ($p < 10^{-25}$ after time 5000). The agent using the fitness-based reward function managed to eat a few times, but never learned a good policy due to its short life. The evolved reward functions allowed the agent to learn a better policy within the allotted lifetime, despite not knowing all the details of the environment that it will face. These reward functions look very similar to those that evolved under nonstationary conditions, but they generally have a weaker gradient toward the east.

VI. Related Research

Other work has explored alternate reward functions in various contexts. Singh *et al.* [2], [3] demonstrated possible benefits of reward function search while illuminating issues surrounding the origins of reward and the distinction between intrinsic and extrinsic rewards. Sorg *et al.* [5] emphasize the benefits of reward function search in counteracting limitations of learning systems, benefits that are exhibited in our results as well. Ng, Harada, and Russell [1] explored shaping adjustments to the task-based reward function that do not disturb the optimal policy. In contrast to their approach, ours essentially defines a new problem (because there is a different reward function) that is easier to learn to solve than the original problem but whose solution entails a solution of the original problem. Shaping rewards do not redefine the problem and can be "learned away," an approach later shown to be equivalent to specifying an initial value function [15]. Reward function search has been explored by others [11], [16]–[20], but to the best of our knowledge, none have used genetic programming methods.

Our work shares some characteristics with existing approaches to transfer learning (e.g., [21]). In this paper, we only experiment over distributions of problems involving the same domain, but one can use our approach to find reward functions that exploit similarities across domains. Future research will explore the utility of reward function search for transfer learning.

VII. Discussion and Conclusion

We demonstrate a genetic programming method to search for alternate reward functions for RL problems and describe classes of problems where it might be particularly useful or necessary to do so. Genetic programming can discover common features across distributions of environments to improve an agent's expected fitness across this distribution. This property may increase performance in a number of real-world problems where examples of the domain are known, but the details of what an agent may face in any particular instance is unknown. We demonstrated a magnified performance boost on a nonstationary version of the Hungry–Thirsty problem [2], suggesting that reward function search may be valuable for dealing with constantly changing environments in realistic problems. In general, reward function search is most beneficial when the up-front computational expense of the search can be offset by a large improvement over a series of problems to be faced over time.

Furthermore, we show how an agent with a properly tuned reward function can perform well on a task that an agent using the fitness-based reward function simply cannot learn in a short lifetime. Practically, a short agent lifetime can be interpreted in many ways: a high-cost for an agent training episode, a hard time constraint on solving an instance of a problem, or an upper-bound on computational time that is reasonable to spend on a task. Consider the case of a physical robot that learns how to balance itself. Each training episode is expensive; an engineer may have to pick the robot up and reinitialize it each time that it falls while learning. The number of real-world training episodes required might be dramatically decreased if a reward function were evolved in simulation first and then transferred to the physical robot to guide learning. Elfwing, Uchibe, and Doya [22] demonstrate the utility of this strategy on a small foraging robot using evolved shaping rewards. Thus, even if the up-front cost is high, reward function search can be beneficial if it helps conserve other, more valuable resources. Sorg *et al.* [5] explore additional ways that a reward function can compensate for limitations of the agent and its learning opportunities.

A simple domain was used for demonstration purposes in this article, but our method can be applied to more difficult RL problems. Most, if not all, of the PushGP parameters could stay the same, although the instruction set would likely have to be changed. A different subset of instructions would be chosen based on the data types required and the complexity of the problem. Although we used a small subset of the total instruction set in our examples, there is evidence that using a larger subset does not significantly slow the rate of evolution. Furthermore, given our results with distractor variables, it seems reasonable to provide all the available state variables as input to all the reward functions in the search space and let the evolutionary process decide which of them are useful. In this manner, a reward function can be evolved for larger problems.

Evolved reward functions can improve the learning rate and fitness of agents in a domain, but they may also have more powerful uses in the future. As mentioned earlier, a properly evolved reward function captures the common features across environments that an agent may face. Thus, such a reward function may be able to aid an agent in autonomous skill identification and learning, allowing the agent to hierarchically decompose and provide good solutions to problems that cannot be readily handled by current methods. In the *options* framework [23], temporally extended actions called options are added to an agent's set of primitive actions. If the options are created wisely and represent useful, reusable skills in a domain, this technique can greatly facilitate learning and planning. The benefit of using options has been shown in domains that have hierarchical structure [24]. However, options are generally either given to an agent by the designer, created when an event in a prespecified class of events happens, or automatically created by a generic (i.e., problem nonspecific) method such as graph partitioning [25]. Evolved reward functions appear to identify key, problem-specific features of environments and therefore may be useful for improving the identification of useful skills for autonomous option creation and effective hierarchical RL.

Acknowledgment

The authors would like to thank G. Konidaris, P. Thomas, W. Dabney, S. Singh, and R. Lewis, and J. Sorg for the many valuable discussions that made this research possible, and J. Klein for his "Psh" source code.

References

[1] A. Ng, D. Harada, and S. Russell, "Policy invariance under reward transformations: Theory and application to reward shaping," in *Proc. 16th Int. Conf. Mach. Learn.*, Bled, Slovenia, 1999, pp. 278–287.

[2] S. Singh, R. Lewis, and A. Barto, "Where do rewards come from?," in *Proc. 31st Annu. Conf. Cogn. Sci. Soc.*, Amsterdam, The Netherlands, 2009, pp. 2601–2606.

[3] S. Singh, R. L. Lewis, A. G. Barto, and J. Sorg, "Intrinsically motivated reinforcement learning: An evolutionary perspective," *IEEE Trans. Autonom. Mental Develop.*, vol. 2, no. 2, pp. 70–82, Jun. 2010.

[4] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA: MIT Press, 1998.

[5] J. Sorg, S. Singh, and R. L. Lewis, "Internal rewards mitigate agent boundedness," in *Proc. 27th Int. Conf. Mach. Learn.*, Haifa, Israel, 2010.

[6] C. Watkins, "Learning From Delayed Rewards," Ph.D. dissertation, Univ. Cambridge, Cambridge, U.K., 1989.

[7] J. H. Holland, *Adaptation in Natural and Artificial Systems: An Introductory Analysis With Applications to Biology, Control, and Artificial Intelligence*, 1st ed. Cambridge, MA: MIT Press, 1992.

[8] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA: MIT Press, 1992.

[9] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone, *Genetic Programming—An Introduction; On the Automatic Evolution of Computer Programs and its Applications*. San Francisco, CA: Kaufmann, 1998.

[10] R. Poli, W. B. Langdon, and N. F. McPhee, A Field Guide to Genetic Programming 2008 [Online]. Available: http://lulu.com

[11] D. Ackley and M. Littman, "Interactions between learning and evolution," in *Artificial Life II*. New York: Westview, 1991.

[12] L. Spector, J. Klein, and M. Keijzer, "The push3 execution stack and the evolution of control," in *Proc. 2005 Conf. Genet. Evol. Comput.*, Washington DC, 2005, vol. 2, pp. 1689–1696.

[13] L. Spector and A. Robinson, "Genetic programming and autoconstructive evolution with the push programming language," *Genet. Program. Evolvable Mach.*, vol. 3, no. 1, pp. 7–40, Mar. 2002.

[14] L. Spector, D. M. Clark, I. Lindsay, B. Barr, and J. Klein, M. Keijzer, G. Antoniol, C. B. Congdon, K. Deb, B. Doerr, N. Hansen, J. H. Holmes, G. S. Hornby, D. Howard, J. Kennedy, S. Kumar, F. G. Lobo, J. F. Miller, J. Moore, F. Neumann, M. Pelikan, J. Pollack, K. Sastry, K. Stanley, A. Stoica, E. Talbi, and I. Wegener, Eds., "Genetic programming for finite algebras," in *Proc. 10th Annu. Conf. Genet. Evol. Comput.*, Atlanta, GA, Jul. 12–16, 2008, pp. 1291–1298.

[15] E. Wiewiora, "Potential-based shaping and Q-value initialization are equivalent," *J. Artif. Intell. Res.*, vol. 19, pp. 205–208, 2003.

[16] T. Damoulas, I. Cos-Aguilera, G. M. Hayes, and T. Taylor, M. S. Capcarrere, A. A. Freitas, P. J. Bentley, C. G. Johnson, and J. Timmis, Eds., "Valency for adaptive homeostatic agents: Relating evolution and learning," in *Adv. Artif. Life: 8th Eur. Conf.*, Berlin, Germany, 2005, vol. 3636, pp. 936–945.

[17] M. L. Littman and D. H. Ackley, "Adaptation in constant utility nonstationary environments," in *Proc. 4th Int. Conf. Genet. Algorithms*, 1991, pp. 136–142.

[18] M. Schembri, M. Mirolli, and G. Baldassarre, Y. Demiris, D. Mareschal, B. Scassellati, and J. Weng, Eds., "Evolving internal reinforcers for an intrinsically motivated reinforcement-learning robot," in *Proc. 6th Int. Conf. Develop. Learn.*, London, U.K., 2007.

[19] M. Snel and G. M. Hayes, "Evolution of valence systems in an unstable environment," in *Proc. 10th Int. Conf. Simulation Adapt. Behav.: From Animals to Animats*, Osaka, Japan, 2008, pp. 12–21.

[20] E. Uchibe and K. Doya, "Finding intrinsic rewards by embodied evolution and constrained reinforcement learning," *Neural Netw.*, vol. 21, no. 10, pp. 1447–1455, 2008.

[21] G. Konidaris and A. Barto, "Autonomous shaping: Knowledge transfer in reinforcement learning," in *Proc. 23rd Int. Conf. Mach. Learn.*, New York, 2006, pp. 489–496.

[22] S. Elfwing, E. Uchibe, K. Doya, and H. I. Christensen, "Co-evolution of shaping rewards and meta-parameters in reinforcement learning," *Adapt. Behav.—Animals, Animats, Software Agents, Robot., Adapt. Syst.*, vol. 16, no. 6, pp. 400–412, 2008.

[23] R. Sutton, D. Precup, and S. Singh, "Between MDPS and semi-MDPS: A framework for temporal abstraction in reinforcement learning," *Artif. Intell.*, vol. 112, pp. 181–211, 1999.

[24] A. Barto, S. Singh, and N. Chentanez, "Intrinsically motivated learning of hierarchical collections of skills," in *Proc. Int. Conf. Develop. Learn.*, La Jolla, CA, 2004.

[25] Ö Simsek and A. G. Barto, "Skill characterization based on betweenness," in *Proc. 22nd Annu. Conf. Neural Inform. Process. Syst.*, Vancouver, BC, Canada, 2008, pp. 1497–1504.

Scott Niekum received the B.Sc. degree in computer science and cognitive science from Carnegie Mellon University, Pittsburgh, PA, in 2005. He is currently pursuing the Ph.D. degree in computer science at the University of Massachusetts, Amherst, under the advisorship of Andrew G. Barto.

His research interests include intrinsic motivation, hierarchical reinforcement learning, and genetic programming.

Andrew G. Barto (M'83–SM'91–F'06) received the B.Sc. degree in mathematics in 1970, and the Ph.D. degree in computer science in 1975, both from the University of Michigan, Ann Arbor.

He is currently a Professor of Computer Science at the University of Massachusetts, Amherst. He codirects the Autonomous Learning Laboratory and is a core faculty member of the Neuroscience and Behavior Program of the University of Massachusetts.

Lee Spector received the B.A. degree in philosophy from Oberlin College, Oberlin, OH, in 1984, and the Ph.D. degree in computer science from the University of Maryland, College Park, in 1992.

He is currently a Professor of Computer Science in the School of Cognitive Science at Hampshire College, Amherst, MA, and an Adjunct Professor in the Department of Computer Science at the University of Massachusetts, Amherst.