
Genetic Programming and Evolvable Machines, 2, 259–288, 2001

Reinforced Genetic Programming

KEITH L. DOWNING keithd@idi.ntnu.no
The Norwegian University of Science and Technology, Trondheim, Norway
Received December 4, 2000; Revised May 11, 2001

Abstract. This paper introduces the Reinforced Genetic Programming (RGP) system, which enhances standard tree-based genetic programming (GP) with reinforcement learning (RL). RGP adds a new element to the GP function set: monitored action-selection points that provide hooks to a reinforcement-learning system. Using strong typing, RGP can restrict these choice points to leaf nodes, thereby turning GP trees into classify-and-act procedures. Then, environmental reinforcements channeled back through the choice points provide the basis for both lifetime learning and general GP fitness assessment. This paves the way for evolutionary acceleration via both Baldwinian and Lamarckian mechanisms. In addition, the hybrid hints of potential improvements to RL by exploiting evolution to design proper abstraction spaces, via the problem-state classifications of the internal tree nodes. This paper details the basic mechanisms of RGP and demonstrates its application on a series of static and dynamic maze-search problems.

Keywords: genetic programming, reinforcement learning, the Baldwin Effect, Lamarckism

1. Introduction

The benefits of combining evolution and learning, while largely theoretical in the biological sciences, have found solid empirical verification in the field of evolutionary computation (EC). When evolutionary algorithms (EAs) are supplemented with learning techniques, general adaptivity improves such that the learning EA finds solutions faster than the standard EA [4, 23]. These enhancements can stem from biologically plausible mechanisms such as the Baldwin Effect [2, 21], or from factors such as Lamarckism [6, 10] which have long since been disproven scientifically but can still work wonders for an EA.

In most learning EAs, the data structure or program in which learning occurs is divorced from the structure that evolves. For example, a common learning EA is a hybrid genetic-algorithm (GA)-artificial neural network (ANN) system in which the GA encodes a basic ANN topology (plus possibly some initial connection weights), and the ANN then uses backpropagation or Hebbian learning to gradually modify those weights [8, 13, 25]. A Baldwin Effect is often evident in the fact that the GA-encoded weights improve over time, thus reducing the need for learning [1]. Lamarckism can be added by reversing the morphogenic process and back-encoding the ANN's learned weights into the GA chromosome prior to reproduction [18].

It is well known that the search spaces in genetic programming (GP) are vast and peppered with large regions of syntactically-correct but functionally-useless code. As a remedy, Teller [19] proposes the distribution of credit and blame to the internal components of a genetic program followed by pre-reproductive genomic changes

based on these evaluations. Hence, learning directly improves the evolving program, as opposed to improving a different representation which may (Lamarckism) or may not (Baldwin Effect) be reverse-engineered into the genome. Teller employs neural programs to achieve results that support his hypothesis that internal reinforcement should help the GP to steer clear of impoverished portions of the search space while enhancing exploitation. However, Teller's central research question is representation independent [19]:

> Can the evolution of algorithms be extended in a domain-independent way to incorporate accurate credit-blame assignment of each program's internal structure and behavior in such a way that focused, principled reinforcement information improves the evolutionary process? (p. 326)

Our research addresses the same issue but uses a more standard representation: basic GP trees [9]. The primary objective is to devise a mechanism by which the Baldwin Effect and Lamarckism can be realized within any standard genetic program, without the need for a complex morphogenic conversion to a separate learning structure. Hence, as the GP program runs, the tree nodes can adapt, thereby altering (and hopefully improving) subsequent runs of the same program. Thus, the typical problem domain is one in which each GP tree executes many times during fitness evaluation, for example, control tasks. Since the goals of this research are subsumed by Teller's general inquiry, our results lend additional support to the hypothesis that internal reinforcement can improve evolutionary search.

2. Biological theories of evolution and learning

One of the first proposals that learning could accelerate evolution was Jean-Baptiste Lamarck's (1744–1829) *inheritance of acquired characteristics*, wherein physical and mental changes incurred during one's lifetime could be passed on directly to offspring. Contemporary knowledge of the germ-soma distinction permits a recasting of Lamarckism in modern Neo-Darwinian terms, as depicted in Figure 1. Thus, the theory entails a reverse transcription of the modified phenotype back into the genotype, a process that is fully realizable and often useful in evolutionary algorithms, but biologically unrealistic except in a few rare cases.

In 1896, James Baldwin postulated an indirect mechanism for the eventual inheritance of acquired characteristics [2]. This *Baldwin Effect* involves two stages. In phase I (Figure 2), assume a set of genotypes spread uniformly about a suboptimal region, D1, of the fitness landscape. If phenotypes have plasticity, then each can essentially perform local search in the fitness landscape (as shown by the circles with horizontal arrows), and a rough estimate of phenotypic fitness will be the time-averaged landscape locations of the phenotype. Clearly, those phenotypes lying near the base of the optimal peak will have better opportunities to learn their way to higher fitness. Hence, they will have a selective advantage, and the population distribution will move from D1 to D2. Basically, learning smooths the fitness

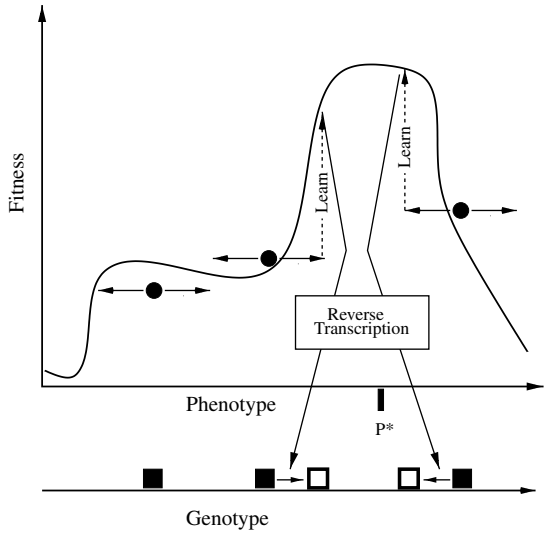

Figure 1. Neo-Darwinian interpretation of Lamarckian inheritance of acquired characteristics: during their lifetimes, phenotypes (circles) can improve and thus climb the fitness landscape away from their innate position and toward the optimal phenotype, P*. Any phenotypic improvements are then reverse transcribed into the genome (changing from filled to open square) prior to reproduction.

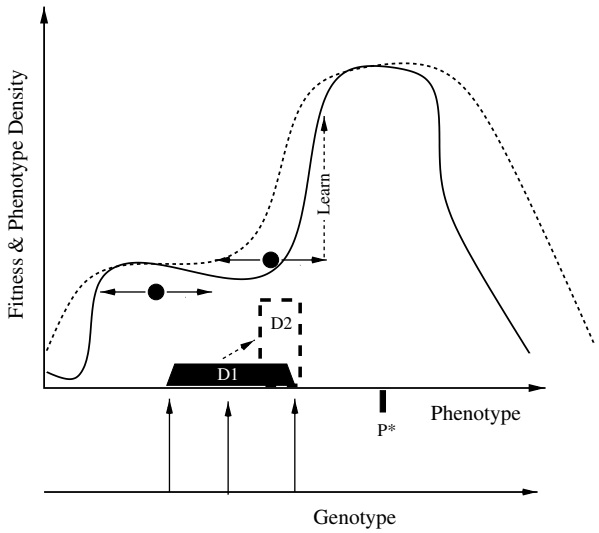

Figure 2. The Baldwin Effect phase I: All phenotypes have the ability to learn, but only those near the base of the peak can achieve fitness increases over the innate value. This selective advantage moves the genotype/phenotype distribution from D1 to D2, hence closer to the optimal phenotype, P*. As depicted by the dotted curve, learning effectively smoothes the fitness landscape.

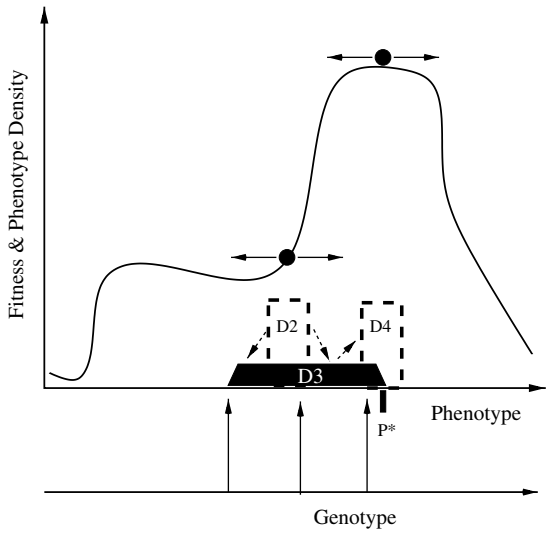

Figure 3. The Baldwin Effect phase II: Genetic operations spread the population distribution from D2 to D3, producing individuals with hard-wired optimal phenotypes, P*. Due to the cost of learning, these *natural-born P*s* have a selective advantage over the *learned P*s*, and the distribution moves from D3 to D4.

landscape and enhances selective pressure such that the population moves toward the optimal phenotype, denoted by P* on the phenotype axis.

To move, and not merely redistribute, the genotype pool, evolution relies on genetic operators (mutation, crossover, inversion, etc.). If the genotype and phenotype space are well correlated [12], then genetics can initiate the emergence of innately optimal phenotypes, *natural born P*s*, and, in general, lead to a flattening of distribution D2 into D3 (Figure 3). Additionally, if learning has a cost, as it normally does [12], then the *P** *learners* will pay it but the natural-born P*s will not, thus giving the latter a selective advantage and moving the population distribution from D3 to D4, where the learned phenotype, P*, becomes fully innate. Thus, in the Baldwin Effect, learning accelerates evolution; and then, if the fitness landscape is static, evolution obviates learning via genetic assimilation.

3. Overview of the RGP approach

Reinforced Genetic Programming combines reinforcement learning [17] with conventional tree-based genetic programming [9]. This produces GP programs with some nodes whose actions are reinforced such that successive runs of the same tree exhibit improved performance on the fitness task. These improvements may or may not be reverse-encoded into the genomic form of the tree, thus facilitating tests of both Lamarckian and Baldwinian enhancements to GP.

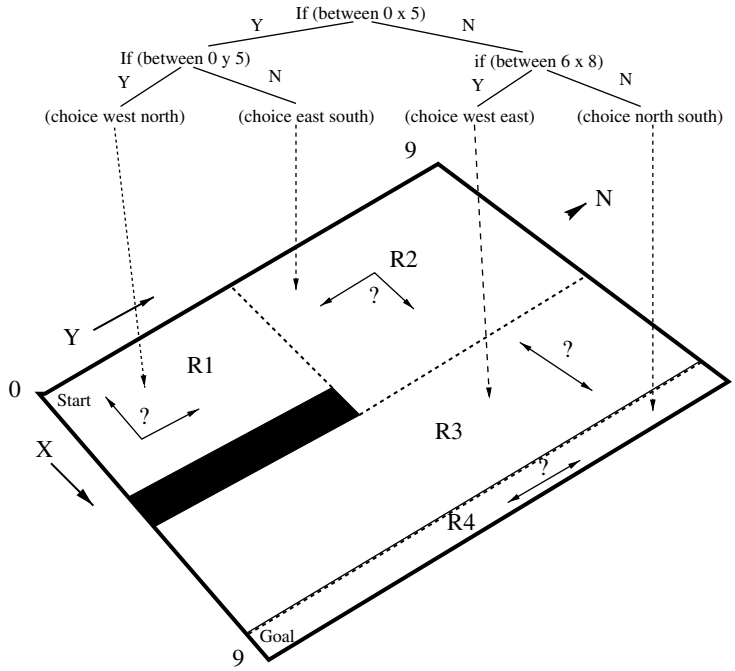

Figure 4. The genetic program determines a partitioning of the reinforcement-learning problem space.

A simple example conveys the basic mechanism. Consider the small control program (a Lisp s-expression) for a maze-wandering agent below:

```
(if (between 0 x 5)
  (if (between 0 y 5)
    (choice (move-west) (move-north))      R1
    (choice (move-east) (move-south)))     R2
  (if (between 6 x 8)
    (choice (move-west) (move-east))       R3
    (choice (move-north) (move-south))))   R4
```

Figure 4 illustrates the relationship between this program and the maze. Variables x and y specify the agents' current maze coordinates, while the *choice* nodes are monitored action decisions. The *between* predicate simply tests if the middle argument is within the closed range specified by the first (upper bound) and third (lower bound) arguments, while the *move* functions are self-explanatory. So if the agent's current location falls within the southwest region, R1, specified by the (between 0 x 5) and (between 0 y 5) predicates of the decision tree, then the agent can choose between a westward and a northward move. Conversely, the eastern edge gives a north-south move option. The agent runs this tree at each timestep as part of a particular task, such as finding the goal location in the southeast corner.

In this example, assume that the GP's strong typing insures that a) the only legal arguments to a choice node are the zero-argument move functions, and

b) s-expressions form decision trees with all and only actions at the bottom. The actions may be monitored (via choices) or simple stand-alone moves. These assumptions are valid for most of the examples presented in this paper but are not general requirements for using RGP, which can also handle internal choice points.

During fitness testing, the agent will execute its tree code on each timestep and take the appropriate action in the maze, which then returns a reinforcement signal. For example, hitting a wall may invoke a small negative signal, while reaching a goal state would garner a large positive payback. The sum of all reinforcements over all timesteps plays a pivotal role in the fitness function, but the individual reinforcements also drive the learning process.

When the action function is wrapped within a choice (see implementation details in the Appendix), the individual reinforcement is processed by the reinforcement learner, which outputs a temporal difference (explained below), which is then sent back to the *state-action pair* (SAP) associated with the performance of the particular action when at that choice node, as depicted in Figure 5. The SAP uses this feedback to update its own evaluation and then propagates a decayed version of the reinforcement to the SAP that was invoked prior to itself (on a previous timestep),

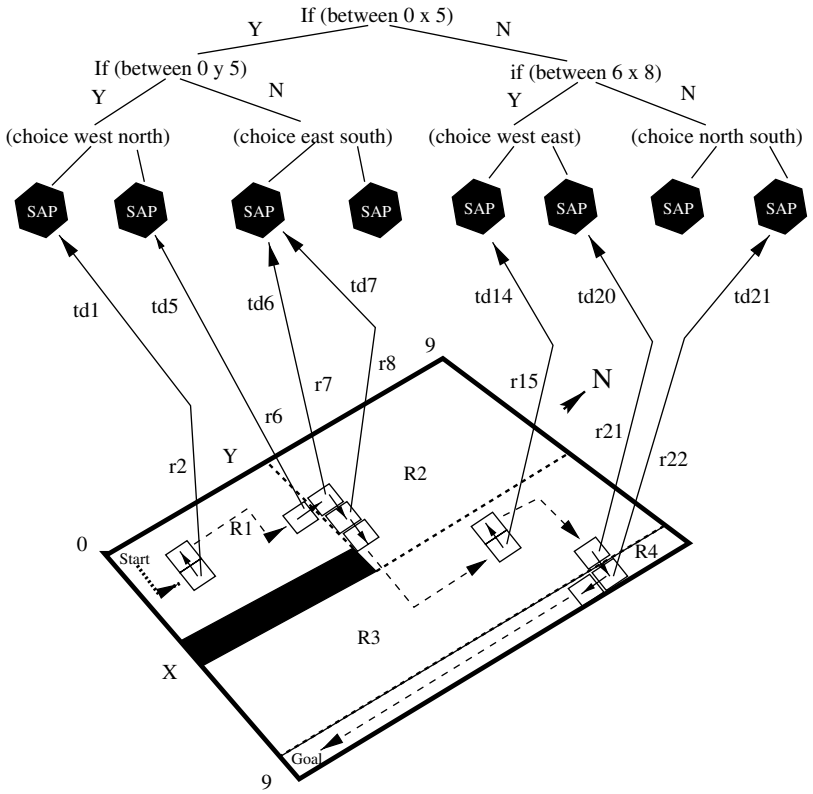

Figure 5. Reinforcements from problem-solving/search are indirectly passed back to the state-action pairs (SAPs) that are associated with the choice nodes of the GP decision tree.

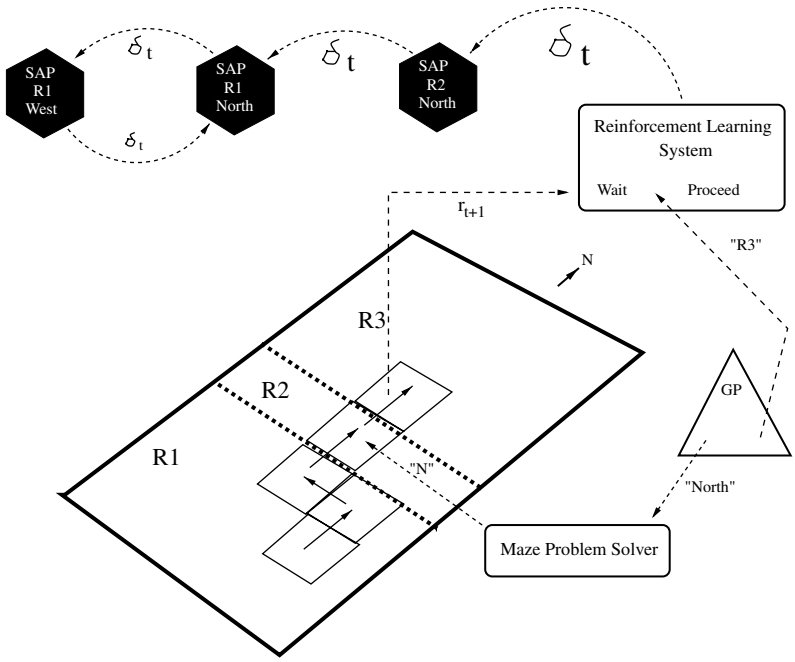

Figure 6. The basic control flow in RGP: The GP tree sends a movement command to the problem solver, which carries it out and returns the reinforcement to the RLS. After waiting to receive the next state from the GP, the RLS computes the temporal difference, δ_t and passes it down the chain of recently-active SAPs.

which updates its evaluation and sends a (further decayed) version to its predecessor, etc., as shown in Figure 6.

Initially, the choice nodes select randomly among their possible actions, but as the fitness test proceeds, each node accumulates reinforcement statistics as to the relative utility of each action (in the context of the particular location of the choice node in the decision tree, which reflects the location of the agent in the maze). After a fixed number of random *free trials*, which is a standard parameter in reinforcement-learning systems (RLSs), the node starts making choices that are biased by these statistics, so the most successful actions are most often chosen. Hence, the node's initial exploration gives way to exploitation. The evolving genome sets the range of this exploration by specifying the possible actions to the choice node, and the RLS fine-tunes the search.

Over the lifespan of an individual, the entire tree behaves more deterministically, reflecting the best learned actions to take in each maze region, relative to the options provided in the genome. Over evolutionary time, the genomes provide more appropriate decision trees and action options for the RLS to work with.

By adding alternate forms of choice nodes, such as choice-4, choice-2, choice-1, where the integer suffix denotes the number of action arguments, the RGP's learning effort comes under evolutionary control. As evident in several runs of the RGP in various domains, the density of broad choices (e.g. choice-4s) increases initially but gradually gives way to narrower choices (e.g. choice-2s and choice-1s) as good

solutions become encoded into the genome and the need for learning decreases—a classic signature of the Baldwin Effect. The choice-1 node merely serves as an RLS monitor wrapper of single actions to insure that their payoffs can be propagated back through the reinforcement chain.

In this example, learning has an implicit cost due to the nature of the fitness function, which is based on the average reinforcement per timestep of the agent. So an agent that moves directly to a goal location (or follows a wall without any explorative "bumps" into it) will have higher average reinforcement than one that investigates areas off the optimal path. Initially, explorative learning helps the agent find the goal, but then evolution further hones the controllers to follow shorter paths to the goal, with little or no opportunity for stochastic action choices. Hence, the average reinforcement (i.e., fitness) steadily increases, first as a result of learning (phase I of the Baldwin Effect) and then as a result of genomic hard-wiring (phase II) encouraged by the implicit learning cost [12].

To achieve Lamarckism, RGP can replace any choice node in the genomic tree with a direct action function (often with a choice-1 wrapper) for the action that was deemed best for that node. Hence, if the choice node for R1 in Figure 4 learns that north is the best move from this region (while choices for R2 and R3 find eastward moves most profitable, and R4 learns the advantage of southward moves), then prior to reproduction, the genome can be specialized to:

```
(if (between 0 x 5)
  (if (between 0 y 5) (move-north) (move-east))
  (if (between 6 x 8) (move-east) (move-south)))
```

This represents an optimal control strategy for the example, with no time squandered on exploration.

4. Reinforcement learning in RGP

Reinforcement Learning comes in many shapes and forms, and the basic design of RGP supports many of these variations. However, the examples in this paper use Q-learning [22] with eligibility traces, so the discussion will focus on those mechanisms.

Conventional RL is essentially *on the job training*, in that the system uses a *control strategy* to work on a problem (e.g., search in a virtual environment) while simultaneously building a *policy*: knowledge about the most appropriate actions to take in different situations/states. In *on-policy* RL, the current policy governs control choices (i.e., exploitation dominates exploration), while in *off-policy* RL, the controller runs independently of the policy, with new information learned by the (often explorative) controller being used to direct policy updates.

Q-learning is an *off-policy temporal differencing* form of RL. In conventional RL terminology, $Q(s, a)$ denotes the value of choosing action a while in state s. Temporal differencing implies that to update $Q(s, a)$ for the current state, s_t, and most recent action, a_t, use the difference between the current value of $Q(s_t, a_t)$ and the

sum of a) the reward, r_{t+1}, received after executing a_t in s_t, and b) the discounted value of the resulting new state, s_{t+1}, where this value is based on the best possible action that can be taken from s_{t+1}, or $\max_a Q(s_{t+1}, a)$. Hence, the complete update equation is:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\big[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)\big] \quad (1)$$

Here, γ is the discount rate and α is the step size or learning rate. The expression in brackets is the *temporal-difference error*, δ_t. Thus, if performing a in s leads to positive (negative) rewards and good (bad) next states, then $Q(s, a)$ will increase (decrease), with the degree of change governed by α and γ.

In conventional RL, all possible states, Σ, are determined prior to any learning, with each state typically a point in a space whose dimensions are the relevant environmental factors and internal state variables of the agent. So for a maze-wandering robot, the dimensions might be discretized x and y coordinates along with the robot's energy level. Conversely, in RGP, each individual GP trees determines its own Σ in a manner that generally partitions a standard RL state space into coarser regions. Whereas a basic Q-learner would divide an N × M maze into NM cell states and then try to learn optimal actions to perform in each cell, an RGP individual divides the same maze into a number (normally much less than NM) of region states and uses RL to learn a single proper action for every cell in each region. Thus, evolution proposes state-space partitions and possible actions for each partition, while learning finds the most appropriate of those actions.

In RGP, the trail through a program tree from the root to a choice node embodies an RL state. In other words, the Q-learning state of the agent-environment duo can only be found by running the tree in the current context and registering the choice node that gets activated. The program thus serves as a state-classification tree with action options at the leaves. Hence, during Q-learning, the temporal-difference update of $Q(s_t, a_t)$ must wait until the succeeding run of the tree, since only then is s_{t+1} known.

For example, from position (2, 5) in Figure 5, if the agent moves north, then the simulator immediately knows that the new cell is (2, 6). However, the RL states are determined by the GP tree, not simply by a 1-1 mapping between cells and states (as in standard RL). So the tree must run again, with the agent positioned at (2, 6), before RGP ascertains that a state transition has occurred between R1 and R2. At that point, Q(R1,north) can be updated via:

$$Q(\text{R1, North}) \leftarrow Q(\text{R1, North}) + \alpha\big[r_{t+1} + \gamma \max_a Q(\text{R2}, a) - Q(\text{R1, North})\big] \quad (2)$$

To implement these $Q(s, a)$ updates, the core activity of Q-learning, within GP trees, RGP employs *qstate* objects, one per choice node. Each qstate houses a list of *state-action pairs* (SAPs), where the *value* slot of each SAP corresponds to $Q(s, a)$. For each GP tree, a *qtable* object is generated. It keeps track of all qstates in the tree, as well as those most recently visited and the latest reinforcement signal. When the tree runs in a particular context, the qstate s_{t+1} corresponding to the newly activated choice node is recorded by the qtable, and $V(s_{t+1})$ (or $\max_a(Q(s_{t+1}, a))$

is then decayed, added to the last reinforcement signal (received in the transition from s_t to s_{t+1}), and this sum is temporally differenced with $Q(s, a)$ to yield an incremental change to $Q(s, a)$.

This basic scheme will then support a wide array of reinforcement-learning mechanisms, which typically differ in their methods of estimating $V(s_{t+1})$ and then updating $V(s_t)$ or $Q(s_t, a)$ [17]. Furthermore, a few simple additions to the SAP objects enable eligibility tracing and full backups, both of which greatly speed the convergence of Q-learning to an optimal control strategy.

RGP employs replacing traces, a form of eligibility tracing, where each SAP has an associated eligibility, $e(s, a)$, which denotes the degree to which $Q(s, a)$ can be updated by the current reinforcement signal. If the SAP for (s', a') is the most recently active (i.e., it dictated the agent's last action) then $e(s', a') = 1$. But for each timestep since (s', a')'s last activation, $e(s', a')$ decays by $\gamma\lambda$, where λ is the eligibility decay rate. Also, if the last active SAP was for (s', a^*), where $a' \neq a^*$, then $e(s', a')$ is reset to 0, indicating that (s', a') should not receive reinforcements that were earned by a competing SAP (i.e., one with the same state but a different action).

Replacing traces then control that amount of temporal-difference error (δ_t) that each $Q(s, a)$ can use for updating:

$$Q(s, a) = Q(s, a) + \alpha\delta_t e(s, a). \tag{3}$$

Figure 6 illustrates this basic process, wherein the GP tree sends a *move* command to the simulator/problem-solver, which makes the move and returns a reinforcement to the RLS, which stores it and waits until the next run of the GP tree to determine the abstract state, $s_{t+1} =$ R3 of the problem solver. The RLS then computes the temporal difference error and sends it to the most recently activated SAP, (R2, North), which relays a decayed version to its predecessor, and so on back through the sequence of active SAPs. Note that (R1, North) appears twice along this chain and therefore receives two reinforcements in different states of decay. Furthermore, (R1, West) receives the decayed reinforcement but must reset its eligibility to 0, since (R1, North) was the most recent move from R1 that earned the reward, and (R1, North) and (R1, West) are competing SAPs. Figure 6 separates the SAPs from the GP tree and uses a different maze than Figures 4 and 5 only for ease of illustration.

Replacing traces facilitate convergence to an optimal policy by allowing credit assignment to filter back through a long chain of state-action pairs for each new reinforcement signal. For instance, if SAP_t, $SAP_{t+1}, \ldots, SAP_{t+n}$ is a sequence of active SAPs that lead to a large reward at time $t + n$, then without eligibility tracing, standard Q-learning would only relay a portion of the reward back to SAP_{t+n-1}. Further back-propagations of the reward along this chain would have to wait until the agent repeated the subsequence SAP_{t+n-2}, SAP_{t+n-1}, but again, the reward would only propagate to SAP_{t+n-2}. So eligibility tracing enables many of the SAPs that are involved in both successful and unsuccessful meta-actions to immediately receive their due rewards or penalties.

By taking full backups, a reinforcement learner further enhances the spreading of control information throughout the RLS state space. In Q-learning, full backups

are off-line transfers of evaluations from SAPs to their upstream nearest neighbors in the activation sequences that were invoked during problem solving. To perform full backups, each SAP, (s, a), keeps track of the qstates that have resulted in the past by performing action a in state s. In a complete Markovian situation, only one such successor state will exist per SAP, but in non-Markovian models, a probability distribution is associated with the possible successor states.

Since the reinforcement learner in RGP often works in a state space of much coarser granularity than the underlying problem space, the transitions from an SAP are typically stochastic at the abstract level. For example, in Figure 4, the abstraction given by the GP tree yields one qstate per region, R1–R4. The SAP (R1, North) will then have two possible outcomes: R1 or R2. The RLS associated with this GP tree must therefore keep track of all transitions from (R1, North) in order to learn the relative probability of each possibility.

With the transition distributions in place for each SAP, full backups are synchronously computed and become the new evaluations of the SAPs:

$$Q(s, a) \leftarrow \sum_{s'} \hat{\rho}^{a}_{ss'} \left[\widehat{\Re}^{a}_{ss'} + \gamma \max_{a'} Q(s', a') \right] \tag{4}$$

where $\hat{\rho}^{a}_{ss'}$ is the transition probability from s to s' on action a, and $\widehat{\Re}^{a}_{ss'}$ is the expected value of the reinforcement for that transition.

The frequency of full backups is parameterized in RGP, with a typical value being once every problem-solving episode, where an episode is one attempt at the problem. For example, in a maze-searching problem, each agent may get n tries to find its way from start to goal. The number of synchronous sweeps through the whole qstate space per backup is also parameterized.

The combination of Q-learning, replacing traces and full backups equip RGP trees with the full capabilities of conventional reinforcement learning systems, thus enabling the embodied control strategy to adapt during fitness testing.

4.1. Maze search examples

Maze searching is a popular task in the RL literature, partly due to the clear mapping from states and actions to 2d graphic representations of optimal strategies (i.e., grids with arrows). Despite this graphic simplicity, the underlying search problem is quite complex, since the agent lacks any remote sensing capabilities, let alone a bird's-eye view of the maze. So trial and error is the only feasible approach, and learning from these errors is essential for success.

Figure 7 illustrates a 5×5 maze with start and goal on the western edge, while Figure 9 shows a 10×10 maze with a start point in the southwest (a few columns east of the western edge) and goal site on the eastern edge but hidden behind an L-shaped barrier. Both mazes include a few subgoals along the optimal path, so agents have opportunities for gaining partial credit. Reinforcements are 10 for the main goals, 2 for each subgoal, -1 for hitting a wall, and 0 for all other moves. Agents are also penalized -1 for repeating any cell that occurred within the past

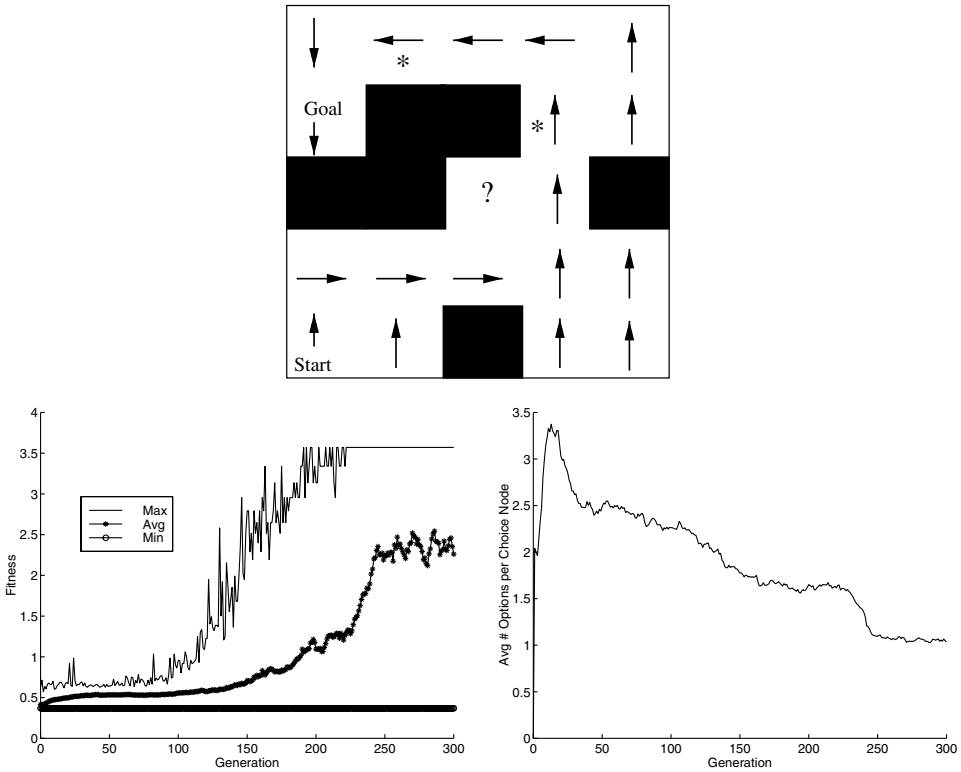

Figure 7. Small 5 × 5 maze and a successful navigation strategy developed by RGP. The question mark denotes a cell without a hard-wired movement action, but which is never visited by this strategy.

20 (10) moves in the large (small) maze (i.e., *minimum loop* = 21 (11)). In the 10 × 10 maze, the optimal path has 20 steps, with a total payoff of 18 (1 goal plus 4 subgoals), while the 5 × 5 maze has a shortest path of 11 steps with a total payoff of 14. Thus, any agent who takes the shortest path will have an average reinforcement per timestep, $\overline{R}$, of 0.9 in the large maze and 1.333 in the small maze. Agent fitness is computed as $e^{\overline{R}}$, so maximum fitness is 2.46 in the large maze, and 3.57 in the small maze.

The RGP functions (with number of arguments in parentheses) are:

1. Logical: and(2), or(2), not(1), in-region(4)
2. Conditional: if(3)
3. Monitored Action: mve(0), mvw(0), mvn(0), mvs(0)
4. Monitored Choice: pickmove(0)

The *in-region* predicate, in-region(x1, x2, y1, y2), returns true iff the x coordinate of the agent's location is in the closed range [x1, x2] and the y coordinate is within [y1, y2]. The 4 *move* actions are for moving east, west, north and south, respectively. These actions expand into single-action choice nodes so that the resulting

```
(if (and (or (or (in-region 1 2 4 2) (or (in-region 0 0 1 0) (in-region 1 3 3 4))) (and (not (in-region
1 2 4 2)) (or (in-region 2 1 2 4) (in-region 3 1 3 2))))(not (and (and (in-region 0 1 0 1) (in-region
3 4 1 0)) (or (in-region 1 1 2 0) (in-region 0 0 0 4)))))(if (not (or (or (in-region 3 1 4 1) (in-region
2 1 3 2))(and (in-region 3 2 3 0) (in-region 4 0 0 1))))(if (in-region 3 3 0 3)(mvn)(if (and (in-
region 3 0 4 4) (in-region 0 1 4 3)) (if (in-region 0 3 2 0) (mvs) (pickmove)) (if (in-region 3 1 3
2) (pickmove) (mvw)))) (if (and (not (in-region 1 4 1 2)) (in-region 1 1 0 0))(mvs)(if (in-region 0
0 1 1) (if (in-region 0 3 2 3) (mve) (mvs)) (mvw))))(if (in-region 0 2 1 2)(if (not (in-region 4 3 3
1))(if (not (in-region 0 2 2 4)) (if (in-region 2 2 1 0) (mve) (mve)) (pickmove))(if (and (in-region
4 1 3 1) (in-region 4 4 1 0))(if (in-region 0 3 4 0) (mvw) (mvn))(if (in-region 4 0 1 3) (pickmove)
(mvn))))(if (and (or (in-region 0 2 2 3) (in-region 1 0 4 0)) (in-region 3 4 0 1))(if (and (in-region
0 0 4 0(in-region 1 1 4 1))(if (in-region 1 4 0 4) (mvn) (mve))(mvn))(if (and (in-region 2 0 2 0)
(in-region 1 1 4 4))(if (in-region 0 1 4 0) (mve) (pickmove))(if (in-region 0 2 1 4) (mvs) (mvn))))))
```

```
(if (in-region 1 3 3 4)
    (if (in-region 3 3 0 3) (mvn) (mvw))
    (if (in-region 0 2 1 2)
        (if (not (in-region 0 2 2 4)) (mve) (pickmove))
        (if (in-region 0 2 1 4) (mvs) (mvn)))))
```

Figure 8. Lisp code for the most fit individual of generation 300 of the 5×5 maze search (above), and its logically-equivalent intron-free version (below).

reinforcement signals can be propagated through the reinforcement learning system to the other choice nodes. *Pickmove* is the only true trial-and-error learning function. It expands into a choice node with all 4 action possibilities. The *if*, *and*, *or* and *not* functions are standard. Terminals for an N $\times$ N maze are the integers 0 through $N - 1$; all maze indexing is 0-based. Strong typing of the RGP trees insures that action and choice nodes occur only at the leaves. The GP uses two-individual-tournament selection with single-individual elitism.

During fitness testing, each agent gets 3 attempts at the maze, i.e., 3 reinforcement-learning *episodes*, with a maximum of 50 steps per attempt (i.e., *max-steps* $= 50$). Each choice node selects actions randomly during the first 16 visits (i.e., *free trials* $= 16$), after which the SAP with highest value gets priority. The discount, γ, and decay, λ, rates for RL are both 0.9, while $\alpha = 0.1$ is the learning rate (i.e., step-size parameter). Many RL systems use a much higher α value, but a lower value seems more appropriate for the non-Markovian situations incurred by RGP's coarse state-space abstractions: it is dangerous to allow the reinforcement of any one move to have excessive influence on a $Q(s, a)$ value when it is unclear whether action a in state s will yield anything close to the same result on another occasion. Table 1 summarizes these details.

Table 1. Tableau for RGP used in a 5×5 and a 10×10 maze-search problem

Objective:	Find optimal strategy for traversing the maze from start to goal.
Terminal set:	$0 \cdots \mathrm{N} - 1$ (for an N $\times$ N maze)
Function set:	and, or, not, in-region, if, mve, mvw, mvn, mvs, pickmove
Standard fitness:	$e^{\overline{R}}$
GP Parameters:	population $= 500$, generations $= 400$, minimum loop $= 21$ $p_{mut} = 0.5$, $p_{cross} = 0.7$
RL Parameters:	$\alpha = 0.1$, $\gamma = 0.9$, $\lambda = 0.9$, episodes $= 3$, max-steps $= 50$ free trials $= 16$, penalty $= -1$, goal reward $= 10$, subgoal reward $= 2$

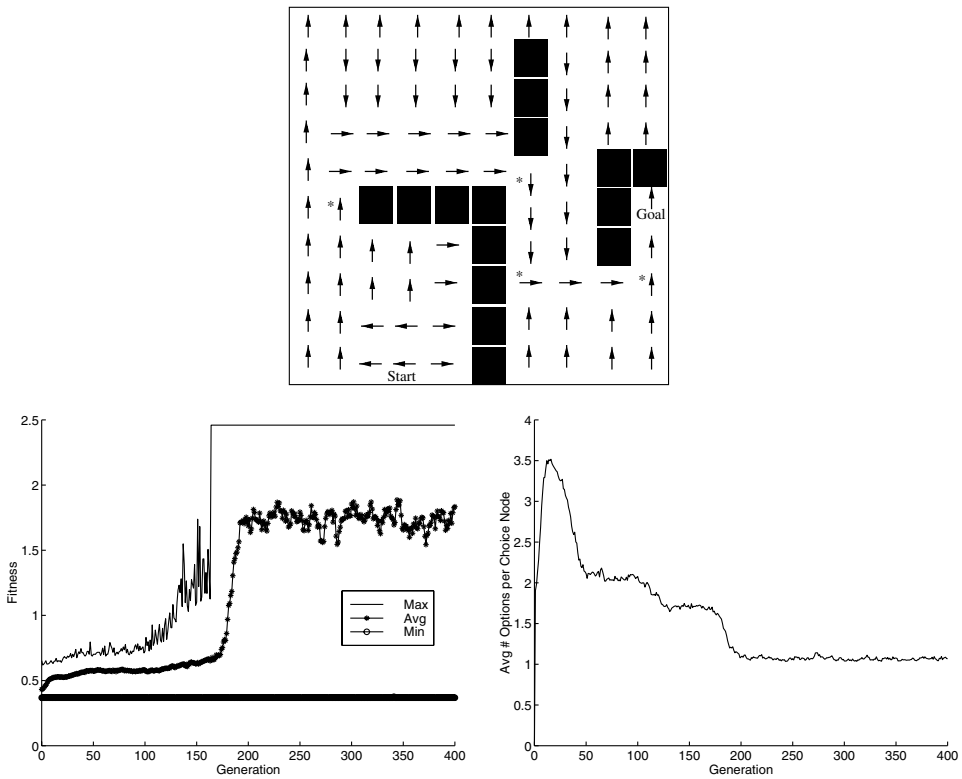

Figure 9. Large 10×10 maze and a successful navigation strategy developed by RGP. Asterisks indicate subgoal cells.

Figures 7 and 9 show the mazes along with the fittest strategy for the final generation, as depicted by arrows. Figures 8 and 10 display the originally evolved and simplified intron-free code underlying each strategy. Figures 7 and 9 also include fitness graphs and plots of the average learning effort per generation. This is simply the average number of decisions made at all of the active choice nodes in the population, where *active* means that control comes to the node at least once during fitness evaluation. An average near 4 reveals a preponderance of pickmove nodes, while values closer to 1 indicate a dominance of single-action choice nodes.

In both cases, note the very slow progress in the first 100 generations, followed by a rapid increase from generation 100 to 175 (big maze) or 200 (small maze). Since the GP uses elitism, the rugged maximum-fitness plots in these transient periods reflect stochastic behavior, which has only one source: the pickmove function. Hence, the agents use learning to evolutionary advantage, as is characteristic of the first stage of the Baldwin Effect. But then, near generation 175 in the 10×10 search, an optimally hard-wired agent emerges and fitness shoots up to the maximum value; the 5×5 search exhibits a gradual rise all the way to the optimal path. The stability of the maximum curve after this ascent entails a total absence of active learning nodes in the highest-fitness individuals, although the population-average learning

```
(if (and (in-region 1 3 0 4)(or (or (not (in-region 4 9 7 0))(or (in-region 5 3 4 7) (in-region 8 9 5
8)))(or (not (in-region 2 5 7 0)) (in-region 2 7 1 0)(if (or (or (or (in-region 9 9 8 0) (in-region 5
0 1 4))(in-region 1 8 6 3))(or (or (in-region 9 1 2 7) (in-region 2 0 2 8))(and (in-region 4 5 2 9)
(in-region 8 9 3 5))))(if (or (in-region 1 5 1 1) (not (in-region 7 5 7 1)))(if (or (in-region 1 1 5 3)
(in-region 5 2 4 3))(mvs)(if (in-region 1 6 4 6) (mvn) (pickmove)))(if (not (in-region 4 4 8 7))(if
(in-region 2 7 3 0) (pickmove) (pickmove)(if (in-region 0 8 8 1) (mvw) (mvn))))(if (or (in-region
3 8 7 5)(or (in-region 8 0 9 7) (in-region 1 1 5 5)))(if (not (in-region 1 8 1 2))(if (in-region 5 9 8
8) (pickmove) (mve))(if (in-region 5 1 8 6) (mve) (mvw)))(if (and (in-region 8 9 5 1) (in-region 8
4 7 2))(if (in-region 5 4 6 0) (mvs) (mvw))(if (in-region 2 3 0 1) (mvw) (mvn)))))(if (in-region 6 6
7 8)(if (or (or (in-region 6 0 9 0) (in-region 2 0 8 3)) (not (in-region 4 9 7 0)))(if (in-region 1 6 3
2)(if (in-region 5 5 7 8) (mvw) (mvw))(if (in-region 0 2 9 9) (mve) (mvw))(if (not (in-region 7 7 6
6))(if (in-region 6 9 1 7) (mvs) (mvs))(if (in-region 9 7 9 2) (mve) (mvs)))) (if (or (or (in-region 6
0 9 0) (in-region 2 0 7 3)) (not (in-region 6 2 8 7))) (if (or (in-region 4 8 2 2) (in-region 1 5 0 6)) (if
(in-region 4 0 6 3) (mvw) (mve)) (if (in-region 1 7 2 8) (mvs) (mvn))) (if (and (in-region 8 9 5 1)
(in-region 8 6 3 7)) (if (in-region 3 9 1 4) (mvs) (mve)) (if (in-region 1 4 6 0) (mvn) (pickmove))))))
```

```
(if (in-region 1 3 0 4)
    (if (in-region 1 1 5 5)
        (if (not (in-region 1 8 1 2))
            (if (in-region 5 9 8 8) (pickmove) (mve))
            (mvw))
        (if (in-region 2 3 0 1) (mvw) (mvn)))
    (if (in-region 6 6 7 8)
        (if (in-region 0 2 9 9) (mve) (mvw))
        (if (or (in-region 4 8 2 2) (in-region 1 5 0 6))
            (mve)
            (if (in-region 1 7 2 8) (mvs) (mvn)))))
```

Figure 10. Lisp code for the most fit individual of generation 400 of the 10×10 maze search (above), and its logically-equivalent intron-free version (below).

rates remain above 1.0 until convergence, which is never complete due to the high (0.5 per genotype) mutation rate.

The learning graphs show the classic Baldwinian progression, with an initial increase in learning rate followed by a gradual decline as learned strategy components become hard-wired. The learning drop correlates with the fitness increase, with the final plunge occurring during convergence: the lack of exploratory moves on the path to the goal facilitates a maximum average reward.

In the maze figures, an arrow indicates a hard-wired move from the corresponding grid cell, as dictated by the optimal RGP tree. Early in the evolution, these diagrams contain many open cells, indicating multi-action choice points. Note that although the arrowed regions guarantee an optimal path from start to goal, they do not promise optimality from any possible start point; a good many are covered, but several regional control strategies send the agent into a wall. Interestingly enough, many of these *satellite* regions (i.e. those lying off the minimal path) evolve optimal strategies during the generations in which stochastic action choices are still being made; as long as agents are wandering off the optimal path and into the satellites, then selection pressure for good satellite strategies still exists. But once hard-wired optimal solutions emerge, agents cease to wander into the satellite cells, and their regional strategies become selectively neutral, hence amenable to genetic

drift. Starting the agents at random initial positions might provide a more robust solution, but probably at the expense of convergence time.

4.2. Performance comparison

Any performance comparison between RGP and standard GP or other GP variants is initially biased against RGP due simply to its increased run time. In the tests conducted below, the RGP typically ran at half the speed of standard GP. Hence, to justify using the RGP in general practice, we need to find hints of improved on-line (i.e., average over all individuals and all generations) or off-line fitnesss (i.e., absolute best over all generations). To get a rough graphic estimate of both measures, we compare the evolutionary progressions of four EAs in a variety of static and dynamic mazes. The four EAs are:

1. a standard GP,
2. a standard GP with one extra function-set member: randmove(0),
3. an RGP, and
4. an RGP with 20% Lamarckism (on an individual, not tree-node, basis)

As shown in Table 2, the RGP employs the same function set as in the previous examples, while the standard GP lacks a *pickmove* equivalent, plus its four *move* functions (one for each direction) are not monitored. For the second EA, *randmove* is a function that randomly selects a move in one of the 4 directions. It does not keep track of reinforcements nor send information to previously-called randmove nodes. Hence, it represents the stochastic exploration of the early stages of RL, but without the credit assignment and adaptivity.

This has the same gist as Hinton and Nowlan's [4] classic Baldwin Effect experiments, where *wildcard* genes encoded for random choices. However, in our tests the number of episodes, 10, and maximum number of moves per episode, 15 (20 for the hardest of the 3 scenarios) give only 150–200 total possible learning attempts per individual. This is a very optimistic estimate, since a) not all moves will involve trial-and-error choices and b) each move is only a subset of an entire strategy. Conversely, Hinton and Nowlan employ 1000 random search attempts per individual, where each attempt embodies a complete solution. Thus, one would

Table 2. Tableau for evolutionary algorithms used in comparative maze runs

Objective:	Find optimal strategy for traversing the maze from start to goal.
Terminal set:	$0 \cdots 4$
Function set:	and, or, not, in-region, if, mve, mvw, mvn, mvs, pickmove
Evolutionary Algorithms:	GP, GP + Random Nodes, RGP, Lamarckian RGP
Standard fitness:	$e^{\hat{R}}$
Runs:	100 per algorithm per maze
GP Parameters:	Population = 50, Generations = 50, minimum loop = 11 $p_{mut} = 0.5$, $p_{cross} = 0.7$ $p_{lamarck} = 0.2$
RL Parameters:	$\alpha = 0.1$, $\gamma = 0.9$, $\lambda = 0.9$, episodes = 10, max-steps = 15 or 20 free trials = 8, penalty = -1, goal reward = 10, subgoal reward = 2

expect a much more impressive Baldwin Effect in their experiments than with this GP + Randmove variant.

In Lamarckian RGP, reverse encoding of learned moves into the genome is on a per-individual basis, so 20% of the maze walkers have all of their active multiple-move choice nodes converted into single-action nodes (for the action that gave the best results for that choice node during the run) immediately prior to reproduction. Intron (i.e., unexecuted) choice nodes are not subject to Lamarckian rewrites.

All RGP and Lamarckian RGP runs employ eligibility traces but not full backups, due to the time-consuming nature of the latter and the need to perform many runs.

4.3. Static maze comparison results

First, consider the simple 5×5 maze in Figure 11, which only requires the agent to make one turn from start to goal, although other more complicated, but equally optimal (i.e. shortest path), solutions exist, all with a fitness of 3.2. In the graphs of Figures 11, 12, and 13, each curve represents the average of the best-of-generation fitnesses for 100 runs of 50 individuals over 50 generations. As Figure 11 clearly illustrates, Lamarckian RGP finds optimal solutions much faster on average than the other 3 EAs, with basic RGP outperforming the two GP variants.

When the task becomes more difficult (Figure 12), Lamarckian RGP also dominates, while basic RGP exhibits a very slight advantage over standard GP. Here, maximum fitness is 5.75, and the fitness plot indicates that none of the EAs can guarantee an optimal solution in a 50-individual, 50-generation search process.

Finally, on the most difficult of the 3 comparison tests (Figure 13), RGP overtakes Lamarckian RGP, while the two GP variants clearly lose ground. Since maximum fitness is 4.95, the scenario once again proves too difficult for guarantees of opti-

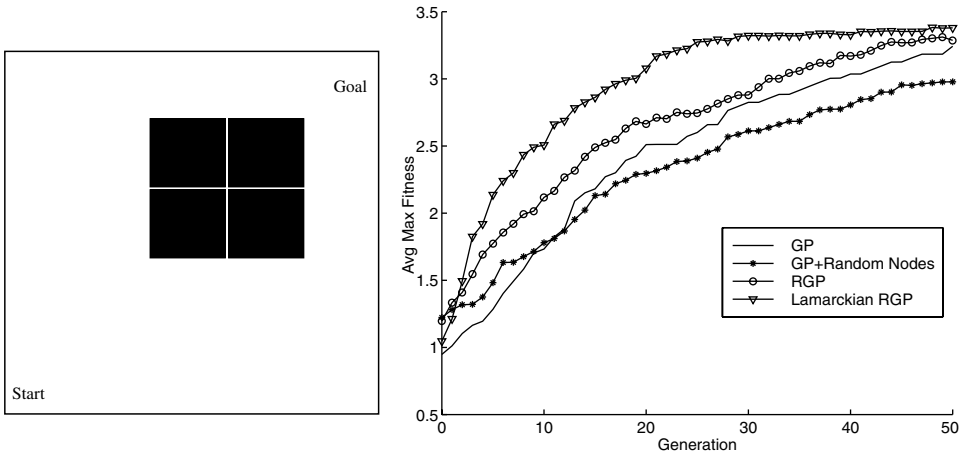

Figure 11. Comparative fitness progressions (right) of 100 runs each of the 4 EAs on the easy 5×5 maze (left).

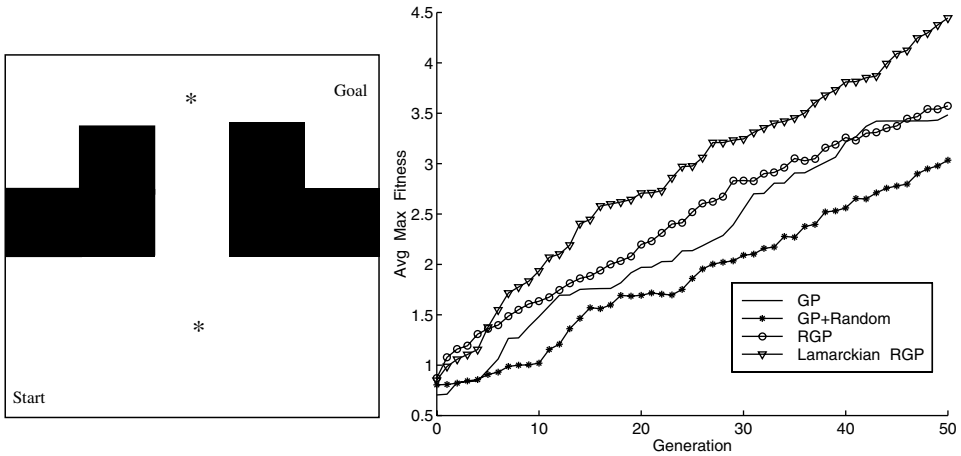

Figure 12. Comparative fitness progressions (right) of 100 runs each of the 4 EAs on a medium-difficulty 5×5 maze (left). Asterisks indicate subgoal locations.

mality during a relatively short search process. But again, the RGP variants show clear signs of accelerated convergence over the conventional GP.

In general, the three static-maze comparisons reveal a significant advantage to the reinforced GPs with respect to total evolutionary effort (i.e., fitness gain per individual tested), whether via Baldwinian or Lamarckian processes. This stems from the RGP individual's ability to a) test different actions and learn from the results, and b) use evolution to regulate the degree of learning necessary for any given search stage.

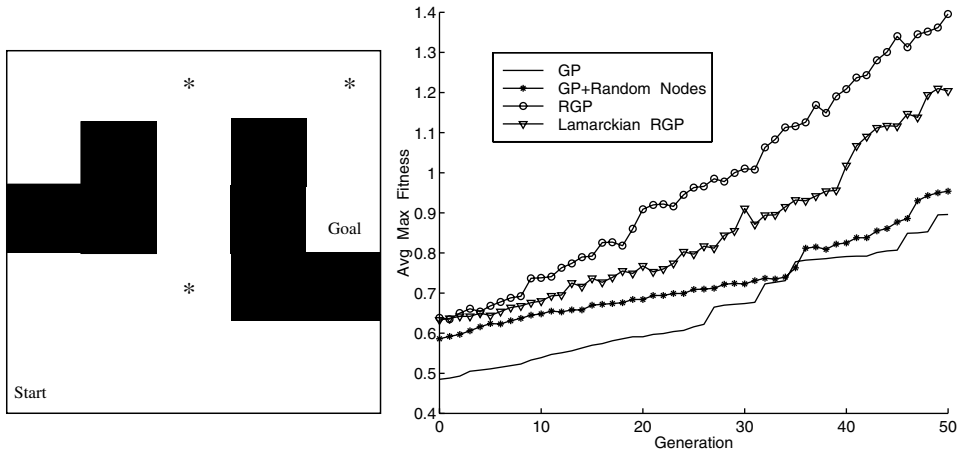

Figure 13. Comparative fitness progressions (right) of 100 runs each of the 4 EAs on a difficult 5×5 maze (left). Asterisks indicate subgoal locations.

4.4. The dynamic maze task

Intuitively, the added flexibility of learning should aid evolutionary search in dynamic domains, but the implicit Markov assumptions behind RL may be counterproductive in these changing environments. Clearly, an empirical study is needed.

Consider a new maze variant in which some cells act as doors, with a probability, p_o, of being open on any particular timestep. Based on the primitive GP functions used previously, both RGP and (standard) GP will encounter the same problem in the dynamic domain: the agent lacks sensors and thus cannot see that a door is closed until it runs into it (and incurs a collision penalty). In GP runs, the agent will merely continue banging into the door, which may or may not re-open in time for the agent to complete its mission. In RGP, the information gained by the collision may eventually bias move selection away from the door, although possibly after repeated attempts at the door. Conversely, if the door is open, the GP agent will move through it and remember nothing, whereas the RGP agent may receive positive reinforcements for the move (due to goals or subgoals encountered on the other side of the door) and thus bias future move choices in the direction of the door. Alternatively, if most moves into the door result in collisions, then these statistics will favor moves away from the door. So regardless of whether the door opens frequently or seldom, the RGP agent would appear to have an advantage in the long run, since it could tailor its move probabilities to p_o.

However, during fitness testing, other elements come into play. Let R be the optimal route through a maze in which all of the D doors along R are open. In GP runs, if a genome encodes R, then the agent may run along R, meet a few closed doors, and get sub-optimal fitness. However, it may get lucky and encounter D open doors, thus harvesting maximal fitness on the one episode that it runs. Now consider an RGP agent whose genome also encodes R. Since RGP agents run through several episodes in order to learn, this agent may meet D open doors in one episode, but the odds of this happening each time obviously diminish with the number of episodes. Since fitness is the average performance over all episodes, the agent will have sub-optimal fitness, even if its R strategy includes a few learning nodes that enable it to avoid some collisions with the closed door and occasionally take an alternate (but sub-optimal) route to the goal.

For the GP runs, any R strategist with optimal fitness will transfer its (possibly mutated and recombined) genotype into the next generation. As the number of R strategists increases, the probability that at least one of them will encounter D open doors during its fitness test will increase, and thus the best-of-population individual will remain at the optimal level for many generations. In the RGP case, the best individuals will proliferate, but the odds against many consecutive episodes with D open doors will remain prohibitive. Hence, the RGP fitness graph will never plateau at the optimal fitness, and standard GP will win most comparison tests in dynamic domains.

The legitimacy of this comparison rests on the problem interpretation. Maze search may represent problem solving in a noisy domain, where a closed door constitutes a *noisy opening*. In another domain, such as cryptanalysis, the noise might be the mistaken encoding of the letter A as X, instead of the intended Y, in 60% of

the $A \rightarrow Y$ translations. This would correspond to $p_o = 0.4$ in the maze domain. A good decryption scheme could then wade through the misleading X's and eventually capitalize on the Ys to break the code. Similarly, the good maze strategist recognizes all closed doors along the optimal route as noisy openings and *breaks through* them to find an optimal route. Under this problem interpretation, multiple-episode testing of standard GP agents is clearly unecessary, since it would only increase the computational burden and decrease progress toward the goal by masking good solutions behind sub-optimal fitness values.

In an alternate, equally plausible, problem interpretation, the goal is to evolve a general-purpose strategy for handling a non-static environment. Here, all agents (using any of the 4 EAs) must run over several episodes to assess their flexibility on the dynamic maze.

Consequently, two testing modes for dynamic mazes are employed: mandatory multiple episodes (MME) and optional multiple episodes (OME). In MME, all EAs run through the same number of episodes for each agent. In OME, the GP agents run only one episode, while the GP + Randmove, RGP and RGP + Lamarckism agents (a.k.a. *learning agents*) obey the following rule: if a) the maximum number of episodes has not been run, and b) at least one choice point of the phenotype was encountered during the previous episode, then run another episode.

In short, OME agents prematurely abort their fitness tests if the previous episode involved no learning. This criteria does not insure that all learning has ceased, since the dynamics of the environment can influence which leaves of the GP tree are executed during an episode. However, it gives relatively hard-wired learning agents a chance to derive maximal fitness from one or a few optimal episodes.

4.5. Dynamic maze comparison results

Figure 14 shows a 6×6 dynamic maze containing 5 fixed barriers, 6 doors, and 4 subgoal cells. When the lower left door and one of the two upper right doors are open, the optimal route takes 7 moves and yields 14 reinforcement points, giving a fitness value of $e^2 = 7.34$. However, when the doors are rarely open, a safer strategy takes the upper route, which involves 11 moves and pays 14 points for a fitness value of 3.56. In the four comparison scenarios, the variant parameters are p_o, t_u, the number of episodes per fitness test, and the mode: OME or MME. Here, t_u is the period length (in terms of the number of agent moves) between stochastic toggling of door states. All doors are independently considered for updating every t_u moves. As before, each EA is run 100 times with a population of 50 agents over 50 generations with the parameters of Table 2, and the comparison graphs display the average over the 100 runs of the highest-fit agents at each generation.

In the OME scenario of Figure 15, the standard GP clearly beats the learning GPs, as predicted above. This domination persists with smaller or larger values of p_o, although it diminishes with more complex optimal paths in other mazes.

The switch to an MME scenario (Figure 16) quickly evens the score, revealing no clear advantage for standard GP with respect to evolutionary effort. Then, by increasing the number of mandatory episodes from 10 to 25, RGP and Lamarckian

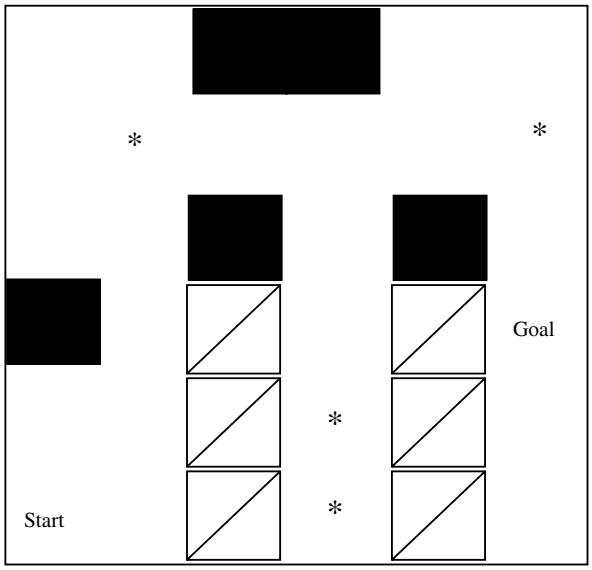

Figure 14. The 6×6 dynamic maze used in the 4 comparison tests of the 4 EAs. Squares with diagonal lines denote doors, and asterisks represent subgoals.

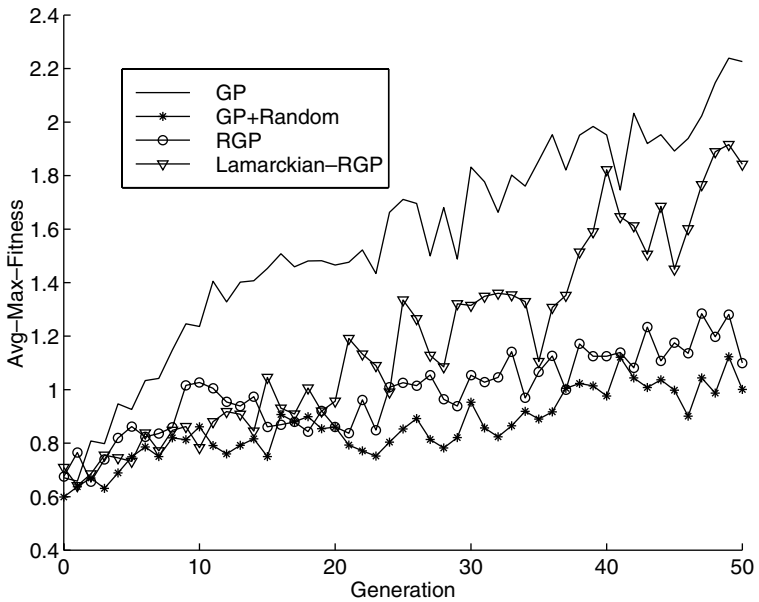

Figure 15. Comparison of the 4 EAs on an optional-multiple-episode (OME) run through the dynamic maze of Figure 14. All agents with Learning GPs used 10 episodes for fitness assessment, while GP agents used one episode. $t_u = 4$ moves and $p_o = 0.2$.

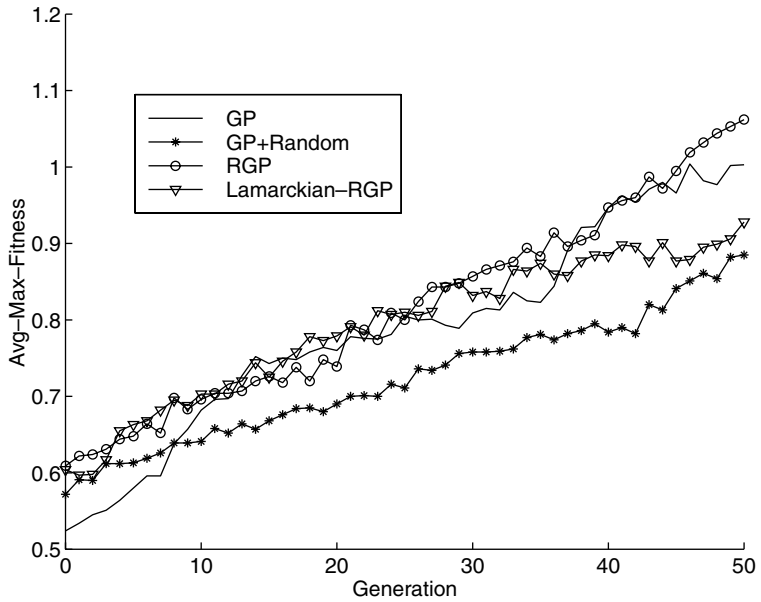

Figure 16. Comparison of the 4 EAs on a mandatory-multiple-episode (MME) run through the dynamic maze of Figure 14. All agents used 10 episodes for fitness assessment. $t_u = 4$ moves and $p_o = 0.2$.

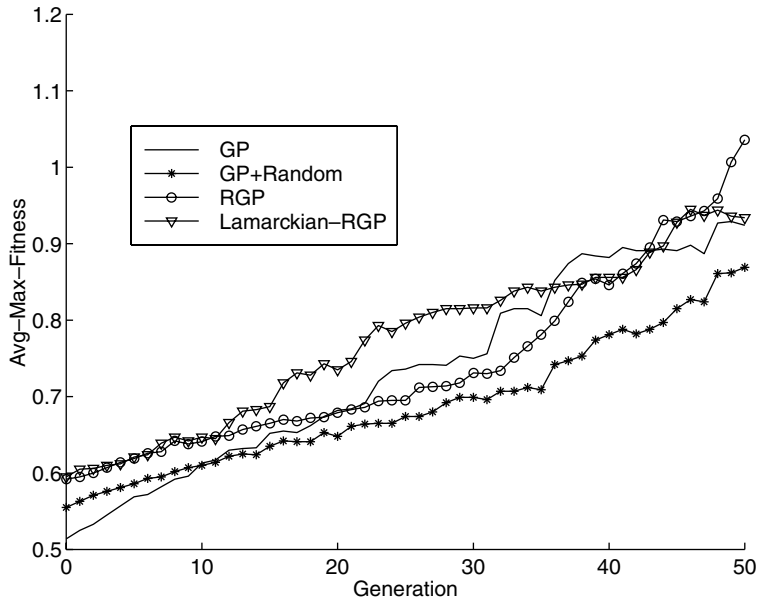

Figure 17. Comparison of the 4 EAs on a mandatory-multiple-episode (MME) run through the maze of Figure 14. All agents used 25 episodes for fitness assessment. $t_u = 4$ moves and $p_o = 0.2$.

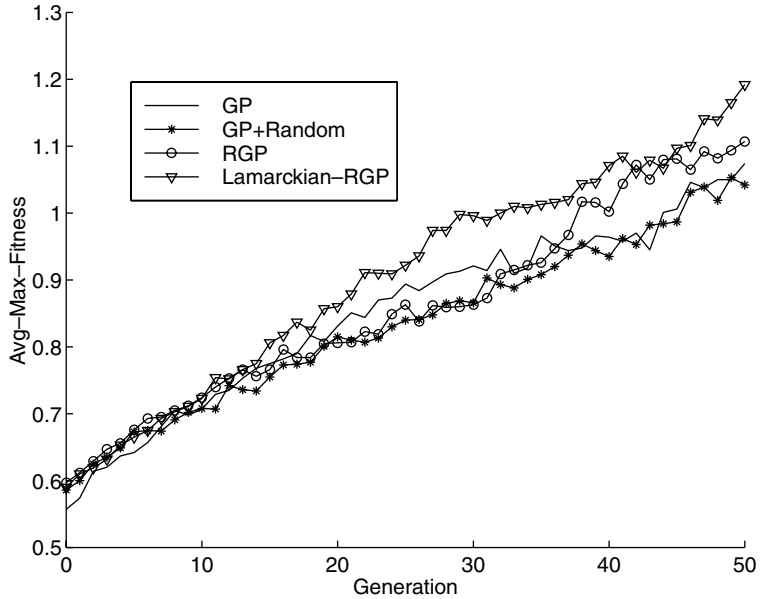

Figure 18. Comparison of the 4 EAs on a mandatory-multiple-episode (MME) run through the maze of Figure 14. All agents used 25 episodes for fitness assessment. $t_u = 1$ move and $p_o = 1.1$.

RGP gain a slight advantage, as shown in Figure 17. Then, in Figure 18, learning enhancement increases when the doors are updated more frequently ($t_u = 1$) but with lower odds of actually opening ($p_o = 0.1$). Since very low p_o values indicate a nearly static maze, RGP gains an expected advantage, as documented in the earlier comparisons.

Finally, to test adaptation to dynamics at an inter-generational timescale, the three static 5×5 mazes of Figures 11, 12, and 13 were run in sequence at 20-generation intervals in 100 OME runs of the 4 EAs. The graph of Figure 19 shows that RGP and Lamarckian RGP hold an advantage during each of the 3 stages, as predicted by the earlier static runs. However, the added flexibility of learning gives no apparent cushion against the abrupt maze changes at generations 20 and 40, since the curves for all 4 EAs drop precipitously at those two generation gaps. The rise at generation 60 simply reflects the switch from the hardest back to the easiest maze. In other maze sequences, RGP adapts well to some inter-generational maze changes, but isolating and classifying those types of changes that RGP could handle proved difficult.

4.6. Comparison summary

The comparisons indicate that the potential advantages of RGP may be greater in static than dynamic domains, with learning providing a nice boost toward a stationary goal. This contradicts the general intuition that flexible agents should display

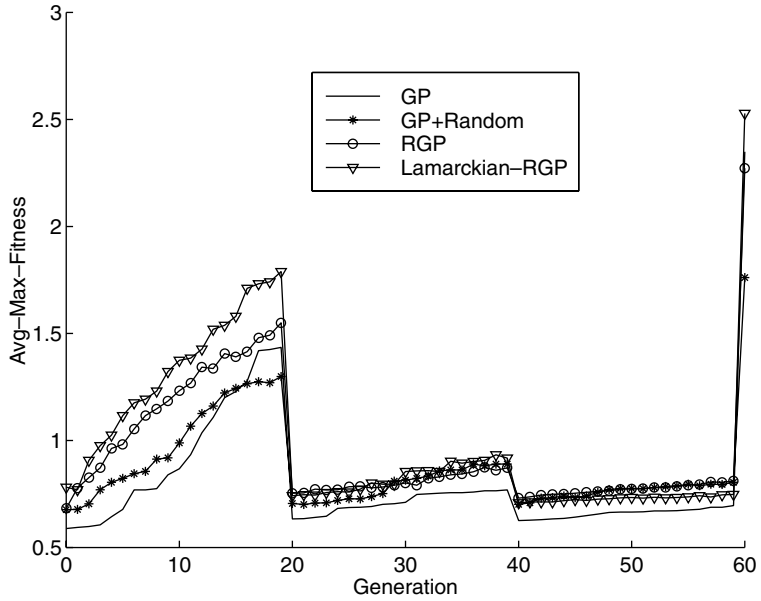

Figure 19. Comparison of the 4 EAs on an optional-multiple-episode (OME) run in which maze changes occurred only at pre-defined generational intervals: 20, 40 and 60. The maze progression involves the easy, medium and hard 5×5 mazes of Figures 11, 12, and 13, with a brief return to the easy maze in the final generation.

their prowess in changing environments. Possibly the non-Markovian situation confounds reinforcement learning, or, alternatively, the above examples may simply lack the complexity necessary for a thorough assessment.

One can argue that representational choice plays a key role in the comparative results. For example, the *in-region* predicate clearly aids the RGP philosophy by providing a built-in state-space abstraction mechanism. A basic GP may profit from a different set of primitives than RGP, thus yielding true comparisons all the more difficult. Still, on the tests above, the primitive set appears useful for all four EAs, as evidenced by the easiest scenario, where all EAs find the optimal solution, but at different rates.

The addition of RL increases the computational effort of a single fitness test by about 50% for a single-episode learning test. But for multiple-episode learning, the effort/episode ratio decreases substantially, since a) the cost of generating the RL data structures is paid only for the first episode, and b) as learning progresses, more actions are chosen directly rather than stochastically, more efficient solutions are discovered, and fewer episode time-outs occur.

In general, further testing on a variety of static and dynamic problems is necessary to judge the computational tradeoffs of RGP versus standard GP. This paper is intended more as a proof of principle than of optimality, but the above comparisons give some support to the claim that reinforcement learning can enhance genetic programming.

5. The potential dual advantages of RGP

Ideally, RGP should benefit both GP and RL. As shown above, the added plasticity that RL gives to GP trees can speed evolutionary convergence to good solutions via Baldwinian and/or Lamarckian mechanisms. Conversely, the use of GP to determine proper state abstractions for RL could represent a huge savings for RL systems that get bogged down in extremely large fine-grained search spaces.

Of course, the hybrid bears added computational costs. The learning GP trees require more space and time to execute than standard GP trees, and although a single RL session in the abstracted state space will often run many orders of magnitude faster than in the detailed state space, the evolutionary effort to find the proper abstraction can dominate run-time complexity. This does not preclude the possibility of mutual improvements, but the potential for such is clearly problem specific and probably only empirically ascertained.

As a simple analysis, consider a fine-grained state space of size n^m with m dimensions defined by variables $V_1, \ldots, V_m$, where each variable has n possible discrete values. Although the space contains n^m points, it has a maximum of only nm possible variable values, and a minimum of n. It is these legal values that form the basis for GP's terminal set. For example, the *between* and *in-region* functions use the (shared) possible X and Y coordinate values as their terminal arguments.

If the GP has F functions with at most k arguments each, and the trees have maximum depth d, then assuming all trees are full, the search space has size $F^{N-1}(mn)^N$, where $N = k^d$. Thus, the search space and hence the computational complexity of the GP is polynomial in both m and n. Conversely, in the fine-grained reinforcement learner, the complexity scales exponentially with m as ATn^m, where A is the size of the RLS's action set, and T is the number of free trials. Clearly, GP complexity greatly exceeds that of RL for all practical sizes of m, k and d, but in the limit, as m exceeds N, the GP's state-space partitions would appear to benefit RL. However, to evolve appropriate abstractions for large m may also require increases to k and/or d, so even in the extreme case, the comparison of search spaces gives no convincing argument that GP can assist RL. Everything hinges on the GP's ability to cleverly manage its search of the abstraction space, and this type of efficiency defies any convincing proof for the general case.

In the previous static maze tests, standard RL finds solutions several orders of magnitude faster than RGP, thus confirming the advantage of RL over RGP on small-scale *typical RL* problems. Our own computational limitations prevented RL-RGP comparisons on huge mazes, but as discussed below, Iba's [7] GP/RL hybrid yields performance improvements over RL on certain tasks.

6. Related work

Essentially, RGP inverts the typical control flow of a tree-based genetic program. For example, whereas Koza [9] attacks the broom-balancing-on-a-moving-cart problem with a set of primitives whose aggregated program returns a cart-movement

command from the top of the tree; the corresponding RGP solution involves primitives that attempt to classify the current problem state (in terms of the cart's velocity, the broom's angle, etc.) and thereby funnel control to a leaf node that houses a movement command or a monitored reinforced choice of such commands. Thus, RGP enforces a different modelling scheme, one which typically requires strong typing of the primitive functions. As with standard GP, designing function sets is more of an art than a science in RGP, but the task is no more complicated, and possibly more natural, when viewed from RGP's classify-and-act perspective.

No discussion of hybrid evolutionary algorithms and reinforcement learners can ignore the extensive research on classifier systems (CS) [5, 11, 16, 24]. Both RGP and CS employ a classify-and-act cycle to solve problems, and both perform learning and evolution on a single representation. Since each path from root to leaf in an RGP tree represents the antecedent of one rule—while the leaf houses a (possibly non-deterministic) consequent—the entire tree embodies a rule set. Thus, evolution in RGP works on collections of rules, akin to the Pittsburgh model of classifiers [3]. Conversely, learning occurs at the individual rule level in both RGP and CS. Whereas classic bucket-brigade learning involves elegant, but indirect, credit assignment, RGP uses the direct transfer of (decaying) reinforcements among rule consequents. However, just as eligibility tracing and full backups speed up reinforcement passing in RL, a host of feedback accelerators exist for classifiers as well [14, 15].

To date, the only direct combination of tree-based GP and RL is Iba's QGP system [7]. It uses GP to generate a structured search space for Q-Learning. Given a set of possible state variables (e.g. w, x, y, z), QGP evolves Q-tables with variable combinations as the dimensions. For example, the genotype (TAB ($^*\times$y) ($+$z 5)) specifies a 2-*d* table with xy as one dimension and $z+5$ as the other. The individual states in this table have the same level of abstraction and scope: each circumscribes the same volume in the underlying continuous state space. In several multi-agent maze-navigation tasks, QGP generates useful Q-tables to simplify RL, and in situations with many possible state variables, QGP outperforms standard RL, which flounders in an exponential search space.

In contrast to QGP, which applies GP to improve RL, RGP uses RL to enhance GP. While Iba constrains his GP trees to a small set of functions and terminals for generating well-formed Q-tables, RGP sanctions the evolution of amorphous decision trees that embody heterogeneous abstractions of the RL search space. One qstate in RGP may represent a single maze cell, while another, in the same GP tree, can encompass several rows and columns or even a concave region or a set of disjoint regions. This reflects the philosophy that the proper abstractions are not necessarily homogeneous partitions of a select quadrant of the search space. Unfortunately, our approach incurs a much larger evolutionary search cost than Iba's, yielding the present RGP an unlikely aid to standard RL. But for improving standard GP, RGP holds some promise, since it endows GP trees with behavioral flexibility.

Whereas QGP strongly couples GP and RL, RGP allows evolution to determine the degree of learning needed for a particular problem, thus facilitating the standard

Baldwininan transition from early plasticity to later hard-wiring in static problem domains.

In the other previous GP/RL hybrid, Teller's use of credit assignment in neural programming [19] more closely matches the goals of our RGP research: to supplement genetic programming with internal reinforcements in order to increase search efficiency. However, the differences between RGP trees and neural programs are quite extreme, as are the associated reinforcement mechanisms. While RGP trees are typically control-flow structures, neural programs involve data flow between distributed neural processors. Internal reinforcement of neural programs (IRNP) closely resembles supervised learning in conventional artificial neural networks: the desired system outputs are compared to the actual outputs over a series of training instances, and correlations between the two form the basis for internal updates. Conversely, RGP is designed for reinforcement learning in the stricter sense of the word [17]: situations where the environmental feedback signals constitute rewards or punishments but do not explicitly indicate the correct problem-solver output. The two key characteristics of RL: trial-and-error search and (potentially) delayed rewards, are intrinsic to RGP. This makes it amenable to a host of control tasks, whereas IRNP appears more tailored for classification problems.

The collective results of QGP, RGP and IRNP indicate that combinations of GP and credit-assignment harbor potential benefits for the whole spectra of adaptive systems from supervised and reinforcement learners to evolutionary algorithms.

7. Discussion

RGP supplements evolutionary search with reinforcement learning, providing a hybrid approach for attacking problems where the GP program will run several times during the course of a single fitness evaluation, and the results of each run may alter the problem environment in a manner that affects succeeding runs. Control tasks are a prime example. However, as the extraordinarily successful applications of RL to tasks such as backgammon playing [20] and space-shuttle task scheduling [26] indicate, a wide range of diverse domains become amenable to the powers of reinforcement learning when approached from the proper angle. Thus, RGP may have utility beyond our current scope.

Although this paper focuses on maze-search applications, RGP has been applied to a few other control problems: pursuit in a 2-*d* virtual world, and broom-balancing [9]. No detailed comparisons have yet been performed, but clearly, these and additional applications are critical to future assessments of RGP's general utility.

The maze examples all employ *choice* nodes exclusively at the leaves of the program tree. Hence, the last activity of each program execution is a call to the problem solver, which returns a reinforcement to the relevant state-action pair. However, RGP also handles internal choice nodes that dispatch and monitor general control activities. Hence, tree code can include several monitored branch points such that RGP gradually learns the best of the competing options, which can then be hard-wired into future genomes via Baldwinian or Lamarckian means.

As a simple example, the scenario of Figure 13 was run with an RGP that included two additional functions: *picknum2(2)* and *picknum4(4)*. These take only arithmetic arguments and then monitor the usage of each. Strong typing in the GP insures that picknums only appear as arguments to *in-region*. This transfers some of the state-space abstraction process to the reinforcement learner. Preliminary results show behavior on-par with the best EA in Figure 13, but not significantly better. Given the excessive run-time cost (300–500% of RGP) of this addition, no justification for these extra choice nodes currently exists. Evolution alone may be the best process for handling adaptations to internal structure.

The primary potential advantages of RGP stem from the integration of learning directly into a tree-structured GP, thus opening the way for Baldwinian and Lamarckian performance enhancements without the need for a second learning structure. One of the true beauties of classifer‘ systems [5] is their integration of problem-solving, learning and evolution into a single representation. RGP has a similar aesthetic appeal, but further research is needed to assess the practical utility of this hybrid approach.

Appendix: Implementation of RGP in Common Lisp

RGP is implemented in Allegro Common Lisp and runs on Unix and Windows operating systems. Both the GP and RL modules are standard algorithms, but the connection between the two involves some finesse, exploiting two of Lisp's most powerful utilities: macros and closures.

When a genotype RGP tree is converted into running Lisp code (a.k.a. the phenotype), most of the primitive GP functions map directly to simple Lisp functions or macros. However, choice nodes are a key exception. They are implemented as macros that expand into closures, i.e., pieces of code with local memory. In each closure, this memory houses a qstate object: the hook to RL. The closure's body is simply a message call, *perform-action*, to the qstate object.

In addition, the entire phenotype is wrapped within an outer closure, whose main local variable is a qtable object in which all qstate objects are registered. Hence, all RL data structures are local to the phenotype. This avoids any global bookkeeping chores associated with running hundreds or thousands of reinforcement learners.

Lamarckian runs require a correspondence between the original choice nodes of the genotype and the closures and qstates of the phenotype, since a Lamarckian learner will have the results of RL back-coded into the genotype (i.e., choice nodes are replaced by direct actions). Lamarckian agents are therefore pre-processed during morphogenesis so that a unique index is appended onto each *choice* call. During macro expansion, this index is then saved with the qstate. Simpler mechanisms are possible, but they require assumptions about the order of macro expansion, a potentially platform-dependent issue.

The claim that RGP integrates learning and evolution into the same GP representation is clearly one of perspective. At the implementation level, the claim is clearly false, since RGP's genotype is an s-expression, while the phenotype is a compiled lambda expression with expanded macros. But at the functional level, the genotype and phenotype differ only marginally in that the phenotype executes in a tree-like manner, with reinforcements being transferred among the leaves.

References

1. D. H. Ackley and M. L. Littman, "Interactions between learning and evolution," in Artificial Life II, C. G. Langton, C. Taylor, J. D. Farmer, and S. Rasmussen (eds.), Addison-Wesley, Reading, MA, 1992, pp. 487–509.
2. J. M. Baldwin, A new factor in evolution, American Naturalist vol. 30, pp. 441–451, 1896.
3. K. A. DeJong, "Genetic-algorithm-based learning," in Machine Learning, Y. Kodratoff and R. Michalski (eds.), vol. 3, Morgan Kaufmann: San Francisco, 1990, pp. 611–638.
4. G. E. Hinton and S. J. Nowlan, "How learning can guide evolution," Complex Syst. vol. 1, pp. 495–502, 1987.
5. J. H. Holland, Adaptation in Natural and Artificial Systems, 2nd ed., The MIT Press: Cambridge, MA, 1992.
6. C. R. Houck, J. A. Joines, M. G. Kay, and J. R. Wilson, "Empirical investigation of the benefits of partial Lamarckianism," Evolutionary Comput. vol. 5, pp. 31–60, 1997.
7. H. Iba, "Multi-agent reinforcement learning with genetic programming," in Genetic Programming 1998: Proc. Third Annual Conf., J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. Riolo (eds.), Morgan Kaufmann: San Francisco, 1998, pp. 167–172.
8. H. Kitano, "Deisgning neutral networks using genetic algorithms with graph generation system," Complex syst. vol. 4, pp. 461–467, 1990.
9. J. R. Koza, Genetic Programming: On the Programming of Computers by Natural Selection, MIT Press: Cambridge, MA, 1992.
10. J. B. Lamarck, "Of the influence of the environment on the activities and habits of animals, and the influence of the activities and habits of these living bodies in modifiying their organization and structure," Zool. Philos., pp. 106–127, 1914.
11. P. L. Lanzi and S. W. Wilson, "Toward optimal classifier system performance in non-markov environments," Evolution Comput. vol. 8, pp. 393–418, 2000.
12. G. Mayley, "Landscapes, learning costs and genetic assimilation," Evolutionary Comput. vol. 4, 1996.
13. G. F. Miller, P. M. Todd, and S. U. Hedge, "Designing neutral networks using genetic algorithms," in Proc. Third Int. Conf. Genetic Algorithms, Morgan Kaufmann: San Francisco, 1989, pp. 379–384.
14. R. L. Riolo, "Bucket brigade performance: I. Long sequences of classifiers," in Proc. Second Int. Conf. Genetic Algorithms, J. J. Grefenstette (ed.), Lawrence Erlbaum Association: Mahwah, NJ, 1987, pp. 184–195.
15. R. L. Riolo, "Lookahead planning and latent learning in a classifier system," in Proc. First Int. Conf. Simulation of Adaptive Behavior: From Animals to Animats, J.-A. Meyer and S. W. Wilson (eds.), MIT Press: Cambridge, MA, 1991, pp. 316–326.
16. G. G. Robertson and R. L. Riolo, "A tale of two classifier systems," Machine Learning vol. 3, pp. 139–159, 1988.
17. R. S. Sutton and A. G. Barto, Reinforcement Learning: An Introduction, MIT Press: Cambridge, MA, 1998.
18. S. Taylor, "Using Lamarckian evolution to increase the effectiveness of neutral network training with a genetic algorithm and backpropagation," in Artificial Life at Stanford 1994, J. R. Koza (ed.), Stanford Bookstore: Stanford, CA, 1994, pp. 181–186.
19. A. Teller, "The internal reinforcement of evolving algorithms," in Advances in Genetic Programming 3, L. Spector, W. B. Langdon, U.-M. O'Reilly, and P. J. Angeline (eds.), MIT Press: Cambridge, MA, 1999, pp. 325–354.
20. G. Tesauro, "Temporal difference learning and TD-Gammon," Commun. ACM vol. 38, pp. 58–68, 1995.
21. P. Turney, L. D. Whitley, and R. W. Anderson, "Introduction to the special issue: Evolution, learning, and instinct: 100 years of the Baldwin effect," Evolutionary Comput. vol. 4, pp. iv–viii, 1997.
22. C. Watkins and P. Dayan, "Q-learning," Machine Learning, vol. 8, pp. 297–292, 1992.

23. D. L. Whitley, V. S. Gordon, and K. E. Mathias, "Lamarckian evolution, the Baldwin effect and function optimization," in Parallel Problem Solving from Nature—PPSN III, Y. Davidor, H.-P. Schwefel, and R. Manner (eds.), Springer-Verlag: Berlin, 1994, pp. 6–15.
24. S. W. Wilson and D. E. Goldberg, "A critical review of classifier systems," in Proc. 3rd Int. Conf. Genetic Algorithms (ICGA89), J. D. Schaffer (ed.), Morgan Kaufmann: San Francisco, CA, 1989, pp. 244–255.
25. L. Yaeger, "Computational genetics, physiology, metabolism, neutral systems, learning, vision and behavior or polyworld: Life in a new context," in Artificial Life III, Proc. vol. XVII, C. G. Langton (ed.), Addison-Wesley, Reading, MA, 1994, Santa Fe Institute Studies in the Sciences of Complexity, pp. 263–298.
26. W. Zhang and T. G. Dietterich, "A reinforcement learning approach to job-shop scheduling," in Proc. Int. Joint Conf. Artificial Intelligence, C. S. Mellish, (ed.), Morgan Kaufmann: San Francisco, CA, 1995, pp. 1114–1120.