Computational Intelligence
An International Journal

Computational Intelligence, Volume 28, Number 2, 2012

AUTOMATIC ALGORITHM DEVELOPMENT USING NEW REINFORCEMENT PROGRAMMING TECHNIQUES

SPENCER WHITE,[1] TONY MARTINEZ,[2] AND GEORGE RUDOLPH[3]

[1]*27921 NE 152nd St. Duvall, Washington*

[2]*Computer Science Department, Brigham Young University, Provo, Utah*

[3]*Department of Mathematics and Computer Science, The Citadel, Charleston, South Carolina*

Reinforcement Programming (RP) is a new approach to automatically generating algorithms that uses reinforcement learning techniques. This paper introduces the RP approach and demonstrates its use to generate a generalized, in-place, iterative sort algorithm. The RP approach improves on earlier results that use genetic programming (GP). The resulting algorithm is a novel algorithm that is more efficient than comparable sorting routines. RP learns the sort in fewer iterations than GP and with fewer resources. Experiments establish interesting empirical bounds on learning the sort algorithm: A list of size 4 is sufficient to learn the generalized sort algorithm. The training set only requires one element and learning took less than 200,000 iterations. Additionally RP was used to generate three binary addition algorithms: a full adder, a binary incrementer, and a binary adder.

Received 10 August 2009; Revised 27 July 2010; Accepted 6 March 2011; Published online 23 April 2012

Key words: reinforcement programming, genetic programming, reinforcement learning, automatic code generation, state representation.

1. INTRODUCTION

Reinforcement Programming (RP) is a new technique that uses reinforcement learning (RL) to automatically generate algorithms. In this paper, RP is used to solve the task of learning an iterative in-place sorting algorithm. Comparisons with earlier work by Kinnear (1993a,b) that used genetic programming (GP), are made. Experiments show that RP generates an efficient sorting algorithm in fewer iterations than GP. The learned algorithm is a novel algorithm, and is more efficient than the sort learned by GP. The generated program is simple to understand for a human. Unlike GP, the program does not grow more complex over time. Instead, the learned program only grows complex enough to solve the problem efficiently. RP learns to make unnecessarily complex programs more efficient by pruning. In the comparisons performed, RP is more efficient than GP at finding a program, and finds more effective solutions. In this paper, the term RPSort refers to the learning algorithm or the sorting algorithm it generates—the meaning should be clear from the context.

Perhaps the most important result gives practical bounds on the empirical complexity of learning in-place, iterative sorts. Using RPSort, we found empirically that: a four-element list is sufficient to learn an algorithm that will sort a list of any size; the smallest training set requires only one list: the list sorted in reverse order; RPSort learns in less than 200,000 iterations.

This reduces the empirical complexity of learning an in-place iterative sort to about the same complexity as learning TicTacToe.

This paper assumes that the reader is familiar with RL, the Q-Learning algorithm (Watkins 1989; Mitchell 1997; Sutton and Barto 1998) and with GP and with Genetic Algorithms (GAs). Discussion of other efforts related to automatic program generation is given in Section 7.

Both RL and GAs attempt to find a solution through a combination of stochastic exploration and the exploitation of the properties of the problem. The difference between RL and

Address correspondence to George Rudolph, Department of Mathematics and Computer Science, The Citadel, TH 225, Charleston, South Carolina 29409, USA, e-mail: george.rudolph@citadel.edu

GA lies in the problem formulation and the technique used to solve the problem. Similarly, RP and GP both use stochastic exploration combined with exploitation, but the techniques and program representations differ dramatically.

GAs (Holland 1975; Goldberg 1989; Fonseca and Fleming 1993; Nonas 1998) model Darwinian evolution to optimize a solution to a given problem. GP uses GAs to generate programs in an evolutionary manner, where the individuals in a population are complete programs expressed as expression trees.

In typical RL (Kaelbling, Littman, and Moore 1996; Baird 1995; Jaakkola, Singh, and Jordan 1995; McGovern and Barto 2001; Sutton, Precup, and Singh 1999), an agent is given a collection of states and a transition function with an associated reward/cost function. Often, there are one or more goal states. The objective of the agent is to learn a policy, a sequence of actions that maximizes the agent's reward. RP uses RL algorithms, where the policy being learned is an executable program. The policy is formulated so that it can be translated directly into a program that a computer can execute.

It is well-known that RL has been used successfully in game playing and robot control applications (Mitchell 1997; Sutton and Barto 1998). Like GP, one of the aims of RP is to extend automatic program generation to problems beyond these domains.

The RPSort algorithm uses a dynamic, nondeterministic Q-Learning algorithm with episodic training to learn the sort algorithm. We formulate the problem in terms of states, actions/transitions, rewards and goal states. A training set comprised of sample inputs and their corresponding outputs are given to the system. The system then learns a policy that maps the sample inputs to their corresponding outputs. This policy is formulated such that it can be directly executed as a computer program. By properly formulating the state representation, the generated program can generalize to new inputs.

The RP version of Q-Learning is dynamic in two ways. The algorithm starts with one, or a few, start states, and grows (state, action) pairs as needed during learning. Empirically, this strategy results in a much smaller representation than a fixed table that contains Q-values for all possible (state-action) pairs. Once a correct policy has been learned, a pruning algorithm is used to delete unneeded states from the representation, thus optimizing the learned policy. While this dynamic scheme works well for small applications, it is not clear that it will scale up to support more complex applications. More will be said about this in Section 7.

Section 2 reviews Kinnear's work with GP and sorts. Section 3 discusses RP concepts. Section 4 describes training the RP system to generate RPSort. Section 5 describes results of experiments with RPSort, and compares those results to Kinnear's earlier GP results. RP techniques also have been used to generate code for a full adder, a binary incrementer, and a binary adder (White, Martinez, and Rudolph 2010; White 2006). Section 6 discusses these algorithms. Section 7 reviews related work. Section 8 is the conclusion.

2. GENETIC PROGRAMMING AND SORTING ALGORITHMS

This section may properly belong under related works; however, some of the discussion on RP assumes the reader has read this material first.

2.1. Genetic Programming

GP has proven to be a flexible method for developing algorithms to solve tasks such as evolving sorting networks (Koza et al. 1997; Sekanina and Bidlo 2005) and developing quantum algorithms (Massey, Clark, and Stepney 2005; Spector et al. 1999).

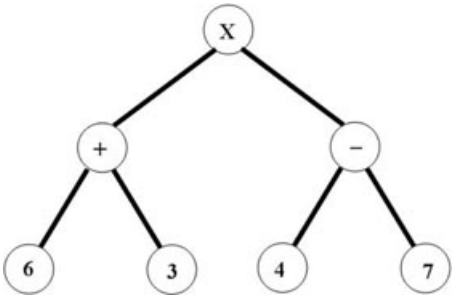

FIGURE 1. A Simple GP-generated program tree.

A common way of implementing a GP system involves using GA techniques to modify a population of trees (Koza 1989, 1990, 1992), where each tree is a complete program. Each internal node in the tree is a function, with a branching factor equal to the number of parameters associated with the function. Each leaf node is a terminal (a nonfunction). Figure 1 shows a simple program. This program tree, executed depth-first from left to right resolves to $(6 + 3) \times (4 - 7)$. The numbers are the terminals, and the mathematical operators are the functions.

A method is provided for determining the fitness of any given tree. Crossover and mutation functions search through the space of possible trees with the expectation that the general fitness of the population will improve over time.

This functional approach to program structure and behavior has powerful implications. First, functional programming languages like LISP can directly express and execute the expression tree structures of the GP-generated program. Second, the set of usable functions is limited only by what can be represented as a function. Usually, the programmer uses domain knowledge (such as operations that are important for an iterative sort, for example) to choose a limited subset of functions to use. Third, and finally, the tree structure allows a GP to represent arbitrarily complex programs. The fitness of any particular generated program is affected by the set of functions made available to the system, but not by the complexity of the problem.

GP differs from standard GA's in one important respect. In standard GA's, the genome representation is a fixed set, with no change to its length or complexity. In GP, program trees can grow to arbitrary lengths. Tree sizes among the population tend to grow rapidly during evolution. Consequently, GP requires task-specific methods to optimize program size without preventing the system from finding a program that is complex enough to solve the problem. This is a nontrivial issue.

2.2. Evolving a Generalized Sorting Algorithm

In Kinnear (1993a), Kinnear describes the challenges presented by using GP to evolve a sort, as well as the techniques he used. Understanding his work helps in understanding RP and the comparisons made later in the paper. Evolving a generalized sort with GP is challenging: there is no limit to the length of a list that requires sorting, so it is impossible to evolve a truly generalized sort by using lists of the maximum size. A collection of techniques must be used, including intelligent ways to select the set of functions to be used in the system, a way to vary the inputs used to determine program fitness, and a way to penalize tree complexity to avoid simple memorization. The programmer formulates these aspects of the problem, not the machine.

TABLE 1. Functions Used in Kinnear's GP-Generated Sort.

Terminals	
len	The length of the list being sorted
index	An iterator variable taking on the value of the current index when in an iterator function, or zero otherwise
Arithmetic Functions	
$n - m$	Returns the value (n m)
$n + 1$	Returns the value ($n + 1$)
$n - 1$	Returns the value ($n - 1$)
Sequence Comparison and Manipulation Functions	
order(x,y)	Puts the elements at indices x and y in the correct order
swap(x, y)	Swaps the elements at indices x and y
wismaller(x,y)	Returns the index of the smaller element at the indices x and y
wibigger(x,y)	Returns the index of the bigger element at the indices x and y
if-lt(x,y,function)	Executes function if the element at index x is less than the element at index y
less(x,y)	Returns 1 if the element at x is less than the element at y, 0 otherwise.
if (test) function	Performs function if test is not 0
Iterative Function	
dobl(start, end, function)	Performs a for-loop from start to end, executing function

Kinnear chose functions and terminals for his GP system that narrowed the focus of his experiments to evolving in-place iterative sorting algorithms—algorithms that only use a small number of iterator variables and the computer memory already occupied by the list. Recursive sorts and non-in-place iterative sorts were not treated. Table 1 lists the functions he chose. Not all of the functions were used concurrently; different subsets of the functions listed were used in different experiments, with different results.

The most-fit algorithm that was generated by Kinnear's system was a BubbleSort. Kinnear, as a result of his experiments, determined that there was an inverse relationship between program size and generality. The more complex a program is, the less general the resulting program will be. Bubblesort is a very simple program with very little complexity in its corresponding GP tree. The inverse relationship implies that Bubblesort is the most fit of all the sorting algorithms, under the set of functions and terminals chosen by Kinnear.

In an effort to decrease the size of the resulting programs, Kinnear included a size penalty in his fitness function. The deeper a program tree is, the less fit the program is considered to be. A strong enough penalty is sufficient to ensure that a program will be simple enough to be general. In fact, a sufficiently large size penalty appears to guarantee at least one generalized sorting program in the population at the end of the trials. It appears that the size penalty prevents the program from learning to use exploits and loopholes in the training set to create sneaky solutions that do not generalize well.

While Kinnear's GP system does generate sorting algorithms, there are several problems. First, Kinnear's experiments failed to always produce a general sorting algorithm without severe complexity penalties. Second, many different programs had to be evaluated during each iteration. Third, the program trees had a tendency to grow large very rapidly, hurting generalization and increasing complexity. Fourth, the solution that is considered the most fit is inefficient for sorting large lists. RP overcomes these issues.

3. REINFORCEMENT PROGRAMMING

RP uses RL techniques to automatically generate programs. The policy is learned and then externalized and executed as a computer program. This requires careful formulation of state, and requires that learning stop after convergence, so that the policy can be translated to efficient source code.

Sutton and Barto (1998) give the following semi-formal, agent-centric description of the RL problem. An agent acts within an environment at each of a sequence of discrete time steps, $t = \{0, 1, 2, 3\}$. At each time step t, the agent receives some representation of the environment's state, $s_t \in S$, where S is the set of possible states. The agent then selects an action, $a_t \in A(s_t)$, where $A(s_t)$ is the set of actions available in state s_t. One time step later, as a result of its state and action, the agent receives a numerical reward, r_{t+1}, and transitions to a new state, s_{t+1}. At each time step, the agent implements $\pi_t(s, a)$, a probabilistic policy for choosing action a in state s at time t. The policy maps (s, a) pairs to probabilities, and the mapping is denoted as π_t.

RL methods specify how the agent changes its policy (updates or learns) as a result of its experience over time. As we have previously stated, the agent's goal is to maximize the total amount of reward it receives over a long time. Sutton and Barto (1998) identify three RL methods: Dynamic Programming, Monte Carlo, and Temporal Difference. Dynamic programming methods are well developed mathematically, but require a complete, accurate model of the environment. Monte Carlo methods do not require a model and are conceptually simple, but are not suited for incremental computation. Finally, temporal-difference methods require no model and are fully incremental, but are more complex to analyze. The three methods also differ with respect to their efficiency and speed of convergence.

RP, as an approach to generating algorithms, could make use of any, or all of these methods for a given application or problem. The Q-Learning algorithm is a form of Temporal Difference learning. Some comments in this section are made with the Q-Learning algorithm in mind, though they also apply more generally.

The basic RP approach is as follows:

(1) Define a set of states.
(2) Define a set of actions, and which actions are available for which states, and how transitions occur.
(3) Define a reward function.
(4) Define a value function.
(5) Define a policy update function.
(6) Define a training set, and incorporate training instances as part of the state.
(7) Train the system to convergence.
(8) Prune the resulting policy to remove unneeded states, if pruning is not part of the learning algorithm used.
(9) Translate the policy into executable code (in this case, decision tree rules).

These steps are the subject of this section and Section 4.

The process above is much like any other RL formulation, however two observations are worth noting. First, the RP version of Q-Learning dynamically grows the mapping function $(state, action) \rightarrow Q\text{-}value$, starting from nothing, and adding pairs as-needed. Second, is policy translation. An optimal policy can be executed, as-is, by a machine. In some RL scenarios, learning continues forever; in others, learning stops when the algorithm converges to a solution. As the stated goal of RP is to generate executable programs, RP requires that learning stops and the policy is translated into executable program code.

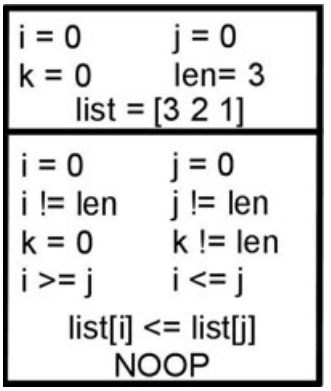

FIGURE 2. An example state structure for RPSort.

3.1. RP State Representation

The state is divided into two parts, *data-specific* and *RP-specific*. The data-specific portion contains the training instance and any variables being used. It may also contain the size of the training instance, and any other instance-specific information. It corresponds roughly to the external environment. The RP-specific portion contains relationships between the data and variables that allow generalization.

An important property of the state is *state equivalency*. Two states are *equivalent* if and only if the RP-specific portions of the two states are the same. The data-specific portion is not used when determining if two states are equivalent to each other. The data-specific portion is only for containing the training instance and the specific values of the variables. Including the data-specific portion in state equivalency would not allow the system to generalize. Instead, the system would simply memorize specific instances.

A variable that directly references the input, or has an explicit purpose is called a *bound variable*. A *free variable* has no predefined purpose.

There are some basic rules about constructing the state, and determining what should be in the RP-specific portion of the state. Bound variables should have Boolean flags in the RP-specific portion of the state that assert facts or relationships in the state. Any element of the input that a bound variable references should also be in the RP-specific portion of the state. The number of bound variables is determined by the nature of the problem. The number of free variables is something to be experimented with. Sometimes the system will learn to make use of the free variable in some way, sometimes it will not. For RPSort, some of the solutions used a free variable as a way of determining if there had been any swaps because the last time both bound variables had equaled 0. The question of how many bound variables to use is open. The RP-specific portion also typically contains the last action performed. This is to give some context to the current state of the system.

For RPSort, a training instance is a list of integers, which is modified when some actions are performed. Thus, the data contained in the list has an effect on the state transitions. The complete state has both a data-specific portion and an RP-specific portion. Figure 2 illustrates this. Information above the line is data-specific, information below the line is RP-specific.

The data-specific portion of the state contains the list being sorted, the length of the list (len) three variables, i, j, k. i and j are both bound variables. k is a free variable. The RP-specific portion of the state contains a set of Boolean flags and the last action

TABLE 2. Actions and Rewards Used in RPSort.

Action	Reward	Penalty	Description
NOOP	0	0	Initial "last action"
TERMINATE	100	−100	Stop
INCI	0	0	Increments i
INCJ	0	0	Increments j
INCK	0	0	Increments k
SETIZERO	0	0	Sets $i = 0$
SETJZERO	0	0	Sets $j = 0$
SETKZERO	0	0	Sets $k = 0$
SWAP	10	−10	Swaps the values in $list[i]$ and $list[j]$

performed. The Boolean flags are: whether $i = 0$, $j = 0$, $k = 0$, $i < j$, $i > j$, $i = len$, $j = len$, $k = len$, and $list[i] > list[j]$. The list is indexed as a 0-based array, (0 to $len - 1$). If $i = len$ or $j = len$, then as a default relationship $list[i] <= list[j]$ to avoid out-of-bounds array errors.

The data-specific portion contains all the information needed to enable a list to be sorted. The RP-specific portion is general enough that lists longer than those the system is trained on can be sorted.

When the policy is converted to a program, the RP-specific portions of each state become the conditionals that control the program, providing the information required to execute the correct sequence of functions to accomplish the goal. Learning creates many states that will never be visited once the system follows a fixed policy after training. This observation allows for some pruning and program simplification. Pruning is discussed in Section 3.5.

3.2. Actions and Rewards

The actions for RPSort, shown in Table 2, were chosen to be as primitive as possible. A NOOP action is provided as a starting "last state" for the initial state. Some restrictions are put on what actions can be chosen, depending on the current state. If any of the three variables i, j, k are equal to len, for example, they cannot be incremented. This is to avoid out-of-bounds array errors in the implementation.

The fewer actions included in the system, the better. Obviously, the complexity of the state transitions is dependent on the number of actions. Furthermore, because the last action performed by the system is typically made a part of each state, the number of *possible* states grows at least linearly with the number of available actions. Generalization implies that the number of actual (state, action) pairs is much less than the number that is possible, however.

Commonly, the list of actions available to the system includes TERMINATE, which simply tells the system to stop executing the program. This constitutes arriving at the goal state. This allows the system to learn to terminate at the best possible state. All other rewards or penalties can be set at small percentages of the reward or penalty for TERMINATE.

Only actions TERMINATE and SWAP give immediate rewards or penalties. All other immediate rewards and penalties are 0. Any state in which the last action performed is TERMINATE is a potential goal state. There is a reward of 100 for terminating when the list is sorted, and a penalty −100 for terminating when the list is unsorted. The system checks a list to see if it is sorted only when TERMINATE is taken. The actual sorting algorithm does

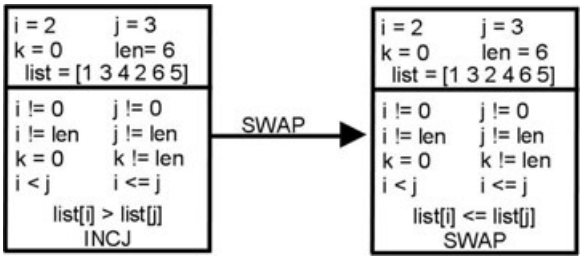

FIGURE 3. SWAP modifies the training list.

not use this check in any way. A sorting algorithm is not required to determine if the list is sorted. It is only necessary to check that each pair of adjacent elements is in ascending order.

SWAP gives a reward of 10 if $i < len, j < len, i < j$, *and* the two swapped elements end up sorted with respect to each other. In other words, a reward was given for finding a sub-solution. Any other SWAP call incurs a penalty of -10. This rewards swaps that progress the list towards a solution, and penalizes swaps that have no effect, result in an out-of-bounds array error, or move the list away from being sorted.

This set of actions is more primitive than those used by Kinnear. There are no explicit "order" instructions, only the ability to swap two elements. The only explicit iteration is the main while-loop that repeatedly executes the policy. Any iteration in the generated sorting algorithm is learned by the system.

3.3. State Transition and the Role of Rewards

A state transition function defines how an action changes the system state. It defines, therefore, how different states are related to each other, and the way in which the state space grows from the original state(s). A state transition may lead to a new state, a state that has already been visited, or even back to the current state (as considered from the perspective of the RP-specific portion of the state). As state transitions are explored, the probability distribution for the nondeterministic nature of the transitions is constructed.

Figure 3 demonstrates a state transition. The arrow indicates the transition, and the label on the arrow indicates the action performed. Recall that because the list is indexed starting at 0, i = 2 references the third element in the list, and j = 3 references the fourth. Figure 3 also demonstrates the way an action can affect the training instance. In the case of RPSort, SWAP is the only action that modifies the list during training. This is similar to how a robot might modify the environment, or a player agent might modify the game state during a game.

Figure 4 demonstrates how RPSort generalizes specific states into meta-states that match multiple input states. The RP-specific portion of states 1A and 1B is identical. Only the value of i in the data-specific portion is different. Because the RP-specific part of both states is the same, the two states are equivalent from the perspective of the learning system. Suppose at some point (*state, action*) pairs for states 1A and 2 are added to the system. Later, when the state represented by 1B occurs during training, the data-specific portion of 1A changes to match 1B, and associated Q-values are accordingly updated.

In fact, any input state that matched the RP-specific portion of the state would trigger the same action, assuming that the algorithm was following a fixed policy at that point. During learning, the learning system reasons over the RP-specific portion of the state, not directly using the data-specific portion. This causes behavior that appears to be nondeterministic in that the same action taken from the same partially observable state leads to different states.

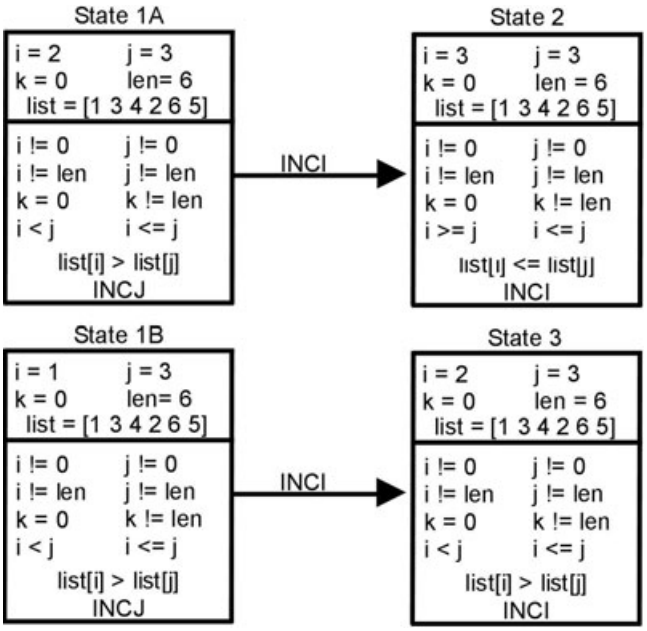

FIGURE 4. An example of generalization in RPSort.

However, when both data- and RP-specific inputs are considered, the process is understood to be perfectly deterministic.

This *apparent* nondeterminism is an expected and necessary consequence of the fact that the $(state, action)$ pairs in the system must generalize over multiple input states. In effect, for RPSort, the system is learning multi-dimensional decision surfaces, with actions as the classifications.

Rewards are provided when the system arrives at the goal state with the proper solution. This guides the system into creating a program that performs the required task. Otherwise, a penalty may be provided when TERMINATE is performed to make the system averse to performing that action in the wrong state. Small rewards may be provided when the system performs an action that generates a partial solution. Similarly, a small penalty may be provided when the system performs an action that undoes a partial solution. This aids in rapid convergence to a solution. RPSort uses a form of delayed Q-learning, because most immediate rewards are 0. It takes time for the consequences from a TERMINATE action to propagate back through the chain of actions to the start.

Assigning rewards does not require any knowledge of what the solution to the task ought to be. It only requires knowing whether a particular action is desirable in a particular state.

3.4. Pruning the Policy

Policy translation is just as important as learning the policy. The better the translation, the more efficient the resulting program will be. State pruning will streamline the code and make it easier to understand. Pruning involves determining which states are not going to be visited at all during program execution and eliminating them from the policy. Recall that the RP-specific portion of the state is used for this process, and the data-specific portion is ignored. Algorithm 1 lists the pruning algorithm in pseudocode.

Algorithm 1 Pruning Algorithm

```
for each list in the training set do
  State s ← Execute the policy with the list as input;
  repeat
    Mark s as visited;
    Choose the action a with the highest Q-value from s;
    State s' ← Execute a in s;
    s ← s';
  until action == TERMINATE
end for
Remove all unvisited states and associated actions;
```

The pruning algorithm is the same as if we were executing the fixed policy, except we mark each (state) as being visited, and we repeat the process for each list in the training set.

One way to guarantee that no important states or transitions have been deleted from the pruned policy is to use an exhaustive training set as input. As with learning, this is impractical for large input spaces.

Apart from that, we have to make the basic assumption that any training set we use for pruning includes all important cases, including generalization; if not, as we will see with RPSort, each failure is detected and training resumes with the failed case included in the training set.

3.5. Policy Translation

The fundamental concept behind policy translation in RP is this: The relationships and constraints contained in the RP-specific portion of the state become the conditionals that handle program control flow; the actions become conditionally executed actions.

A number of different methods can be used to translate a policy into source code. One method we have used is to create a decision tree. The different features of the states become the features of the decision tree, and the action to perform in each state becomes the classification. The policy's transition function is used to determine the next state. If the number of different states is small, a simple exhaustive search can be performed to create the tree with the least number of leaves and shallowest depth. If the number of states is large, however, other heuristics could be used to construct the decision tree.

Once the decision tree is created, program execution is simple. The current state is passed into the decision tree. The state descends through the branches of the tree until a leaf node is reached. The action at that leaf node is executed, modifying the state. This process repeats until a goal state is reached.

We give a small example to illustrate how a decision tree is constructed. For convenience, the RP-specific portion of the state will be referred to simply as "the state," ignoring the data-specific portion of a state. A *split* refers to dividing a set of states into groups according to a specified variable or aspect of the RP-specific portion of the state.

The process of constructing the decision tree takes place as follows: first, the set of state-action pairings being used is analyzed to see if they all have the same action. If so, that node of the tree becomes a leaf node with that action as the classification. If not, all the different attributes from the RP-specific portion of each state are used to recursively

TABLE 3. A Simple Policy.

State Number	Last Action	i	Action
1	*NOOP*	0	*INCI*
2	*INCI*	1	*INCI*
3	*INCI*	2	*TERMINATE*

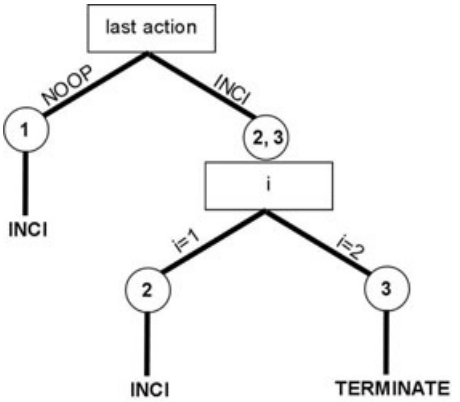

FIGURE 5. Splitting on last action then i.

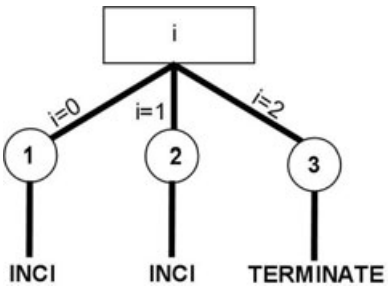

FIGURE 6. Splitting on i.

construct the rest of the tree. The attribute that provides the sub-tree with the least number of leaf nodes is used at that level. The resulting decision tree, automatically generated from the learned policy, can be translated directly into source code.

Table 3 shows a simple policy. The state number is for reference purposes only, it is not a part of the state. This program increments a variable i until it equals 2. The available actions are $NOOP$, $INCI$ which increments i, and $TERMINATE$ which ends policy execution.

Figures 5 and 6 show two trees that could be generated from the simple policy. The search algorithm generates the tree in Figure 5 by splitting on the last action first, then splitting on the value of i. After the first split, state 1 is all by itself because there is only one "best action" in that group. That node becomes a leaf node. The other node contains states 2 and 3. Because these two states have different best actions, a split must be performed. After the split, all the states have entered leaf nodes, resulting in a finished tree. This tree has 3 leaf nodes and a depth of 2.

Once the system is done exploring all possible trees that can be created by first splitting on the last action performed, it explores all possible trees that can be created by splitting on another state variable, in this case i. Figure 6 shows the tree that results from this split. Each node in the tree is a leaf node, so execution stops. This tree has 3 leaf nodes and a depth of 1. Because this depth is shallower than the first tree, that first tree is discarded and the new tree is kept.

Execution continues after this fashion until all possible ways of constructing the tree have been explored. The tree with the fewest leaf nodes and the shallowest depth is kept and used.

3.6. Reinforcement Programming vs. Genetic Programming

An advantage of RP over GP is the style of the programs produced. GP constructs tree-like programs, presenting a structure somewhat more familiar to programmers. RP represents programs in a format similar to what computers use: states and transitions. This format can be a strong advantage for program execution.

A second advantage that RP has over GP is that the problem that exists with tree growth in GP does not exist in RP. The way typical Q-learning techniques operate ensures an efficient solution to the problem. Rewards received at goal states are discounted as the current state gets further away from the goal. This causes the system to find a solution that maximizes the reward. Thus, the preference for short, generalizable programs over long, nongeneralizable programs is automatically provided for by RP.

One comparative disadvantage to RP is that there is a lot of initial exploration involved, at least with the delayed Q-Learning model. It can take a lot of random exploring before a well-defined policy emerges to guide the learning process. GP trees are immediately executable and can be evaluated instantly. However, the fitness of those programs is generally low until a lot of exploration of the possible search space has occurred.

4. TRAINING FOR RPSORT

RPSort is the group of algorithms developed by RP to sort lists of arbitrary size. The programs produced by RP for sorting generally have the same structure, with minor control differences resulting from the different random walks the system takes during each execution. As discussed previously, the actual resulting programs are dependent on the actions and state representation chosen for the system. This section discusses the training algorithms, training data, and example training scenarios.

4.1. Q-Learning Customization

RPSort uses episodic, dynamic, nondeterministic, delayed Q-learning for training. As mentioned in Section 3, the goal of learning is to learn the optimal policy, then translate that policy into a high-level language. This differs from typical Q-Learning applications, where the goal is to learn the optimal policy and then execute that fixed policy. Equation 1 expresses policy execution mathematically.

$$\pi^* = argmax_a(Q(s, a)) \tag{1}$$

This equation indicates that in a given state s, chooses the action a with the highest Q-value and executes it. As with other applications where there are goal states, policy execution

TABLE 4. Symbols Used in Q-Learning Algorithm.

$Q(s, a)$	current Q-value for state-action pair (s, a)
$\hat{Q}(s, a)$	estimated Q-value for state-action pair (s, a)
α	a decaying weight, where $\alpha_n = \frac{1}{(1 + visits_n(s, a))\sigma}$
$visits_n(s, a)$	the number of times state-action pair (s, a) has been visited during learning
γ	discount factor
r	the nondeterministic reward function for performing action a in state s
σ	a rate of decay on the effect of visits to a state-action pair, $0 < \sigma \leq 1$
π	the policy executed by the system

stops when the system reaches a goal state, rather than continuing forever.

$$\hat{Q}(s, a) \leftarrow (1 - \alpha)\hat{Q}_{n-1}(s, a) + \alpha[r + \gamma max_{a'}\hat{Q}(s', a')] \quad (2)$$

Equation (2) is the update equation used on lines 10 and 18 of 3. The reader may have expected the more familiar form like equation 6.6 of Sutton and Barto (1998), which handles deterministic rewards and state transitions. The form used here is equation 13.10 of Mitchell (1997), and handles nondeterministic reward and transition functions. The two forms are equivalent when α and σ are set to 1. RPSort would converge using the deterministic update equation, but we wanted to be as general as possible when developing concepts for RPSort and RP.

Algorithm 2 gives the episodic training algorithm used for RPSort. Algorithm 3 is the learning/update algorithm, an expansion of line 10 in Algorithm 2.

The purpose of the variable $iterationCount$ in Algorithm 2 is to cut off training if the algorithm goes a long time with at least one list failing to sort correctly. The value is only changed on line 6. The number 1,000,000 is a somewhat arbitrary heuristic limit. Recall that RP uses delayed Q-Learning. Training needs to go on long enough that the impacts of rewards and penalties will propagate back from TERMINAL states to any associated start state. Because the learning algorithm must terminate, "long enough" cannot be infinite.

The variable $s.visits$ in Algorithm 2 is a count of how many times the specified start state has been visited during the current training epoch. If a start state s has not been visited a certain number of times, the algorithm assumes more training is required, in order for changes to propagate back to the state. Note that the value is only checked in Algorithm 2—it is incremented during applicable update steps in Algorithm 3. The limit here of 100,000 is a heuristic that was chosen empirically. This number would likely increase for more complex applications.

Table 4 gives definitions of the symbols used in equations in the algorithm.

The loops in Algorithm 3 propagate the effect of a nonzero reward back to one or more start states. The variable $propagationIteration$ provides a finite cut off for this phase of training if the algorithm goes a "long time" without reaching a terminating state. The algorithm checks to see if the first propagation loop has provided enough training by checking the number of times the terminating state has been visited. If so, it stops and goes on to the next training instance. If not, iterate a second time. These controls are empirical heuristics that appear to work. These numbers also will likely change for more complex applications.

Because the algorithm dynamically adds (*state,action*) pairs during learning, this algorithm is different than if we used a system with predefined values. If the current state has no known "best-so-far" policy, or if the "best-so-far" policy has a negative Q-value, an action

Algorithm 2 Episodic training algorithm for RPSort

Require: training set T;
Require: $X \subset T$;
while *at-least-one-list-fails* and $iterationCount < 1000000$ **do**
  at-least-one-list-fails $\leftarrow$ false
  Add one (state, action), per action, to the Q-table for each list in the training set, if the pair does not exist;
  $maxedOut \leftarrow$ false;
  while $maxedOut =$ false **do**
    $iterationCount \leftarrow iterationCount + 1$
    $maxedOut \leftarrow$ true;
    for $L \in X$ **do**
      initialize start state s with L;
      learn (L, *Q-table*,...,);
    end for
    if $s.visits < 100000$ **then**
      $maxedOut \leftarrow false$;
    end if
  end while
  for all $L \in X$ **do**
    $sorted \leftarrow policy.execute(L)$;
    if $sorted == false$ **then**
      at-least-one-list-fails $\leftarrow$ true;
    end if
  end for
  if *at-least-one-list-fails* == false **then**
    for all $L \in T$ **do**
      $sorted \leftarrow policy.execute(L)$;
      if $sorted == false$ **then**
        at-least-one-list-fails $\leftarrow$ true
      end if
    end for
  end if
end while
if $iterationCount > 1000000$ **then**
  $failureToConverge \leftarrow true$;
end if

is chosen at random (each action chosen with equal probability) and tested. Otherwise, the action with the highest Q-value is executed 50% of the time. The other 50% of the time an action is chosen at random. These parameters for exploration/exploitation allow for a lot of exploration, but still permit enough exploitation as to truly evaluate the policy.

Values for γ, the rewards and penalties were determined empirically. The ratio of the reward and penalty is more important than their actual values (i.e., reward/penalty gave exactly the same results as (10*reward)/(10*penalty)). In other experiments, nonzero rewards were held constant, but γ and the penalties were varied. A γ value of 0.6 to 0.9 with a penalty

Algorithm 3 The delayed Q-Learning algorithm for RPSort

Let X be a list from the training set;
$propagationIteration \leftarrow 0$;
Initialize s with X;
while $(s.lastAction \neq TERMINATE)$ and $(propagationIteration < 1000)$ **do**
  $propagationIteration \leftarrow propagationIteration + 1$;
  if $(s.lastAction == NOOP)$ or (highest Q-value for any action < 0) or (random number $>$ ExploreThreshold) **then**
    Choose random action a;
  else
    Choose action a with highest Q-value;
    $s' \leftarrow$ Execute a on s;
    update $Q(s, a)$;
  end if
end while
if $s.visits < \frac{MaxVisits}{2}$ **then**
  STOP;
end if
$propagationIteration \leftarrow 0$;
Initialize s with list X;
while $(s.lastAction \neq TERMINATE)$ and $(propagationIteration < 2000)$ **do**
  $propagationIteration \leftarrow propagationIteration + 1$;
  Choose action a with highest Q-value
  $s' \leftarrow$ Execute a on s
  update $Q(s, a)$
end while

of -160 to -180 leads to convergence in less than 200,000 iterations in 90%–95% of attempts using these values.

200,000 was arrived at by repeating attempts to generate a sorting algorithm and increasing the maximum number of iterations allowed until convergence occurred in 90%–95% of the attempts, and increasing the iterations did not increase the success rate, but reducing the iterations decreased the success rate.

4.2. Training Data

Generating training data is simple. A list length is provided (len) as input, and an exhaustive set of lists is created (all possible permutations of the numbers 1 to len). These lists are ordered lexicographically. Experiments to determine the size of len, and a minimal training set, needed to achieve a generalized sort are discussed in Section 5.

Generating all permutations of a very large lists is computationally prohibitive. The training set of lists initially starts out with the backwards list–the list sorted in descending order–which is also the last list in the lexicographical ordering. From the standpoint of sorting in ascending order, this list is the most unsorted list. All other lists can be reached from the backwards list using swaps that lead towards a solution. Using other lists risks not reaching all possible states, and will potentially require retraining with additional lists. Using

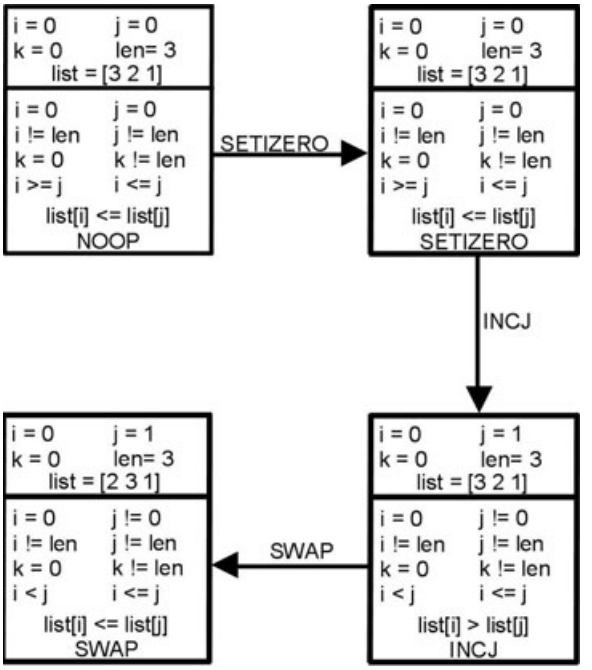

FIGURE 7. A scenario including SWAP.

just the backwards list typically results in a generalized sort without the need for additional lists. At each iteration, each list in the training set is used in the system. Training of the system proceeds for 200,000 iterations. Over 90% of the time the system converged to a solution within that number of iterations. As a test, the resulting policy is used to try to sort all of the lists in the exhaustive set in lexicographical order. If a list is found that cannot be sorted by the policy, that list is added to the training set, testing is stopped, and training begins again. This repeats until all the lists in the exhaustive set can be sorted. The reason for this method is to use a minimal training set. Having a minimal training set decreases the number of evaluations needed, especially if the length of the list is large.

4.3. Examples of Training

This section provides specific examples of scenarios that can happen during training. It shows how the set of (*state, action*) pairs grows as different actions are explored. The effects of actions on the training example will be shown, as will some of the resulting goal states. The way rewards backpropagate through the state space will not be reviewed, because the reader is presumably familiar how that occurs in Q-Learning. The values and formulas described in Section 4.1 are used here.

The first scenario will build the system state shown in Figure 7, starting from the initial state. Initially, only the top left state exists in the system. Because no other states exist, an action is randomly chosen. Suppose that action is SETIZERO. Now the top right state is added to the system. Because this scenario happens at the very beginning of training, Figure 7 represents a random walk through the state space. The random walk occurs because

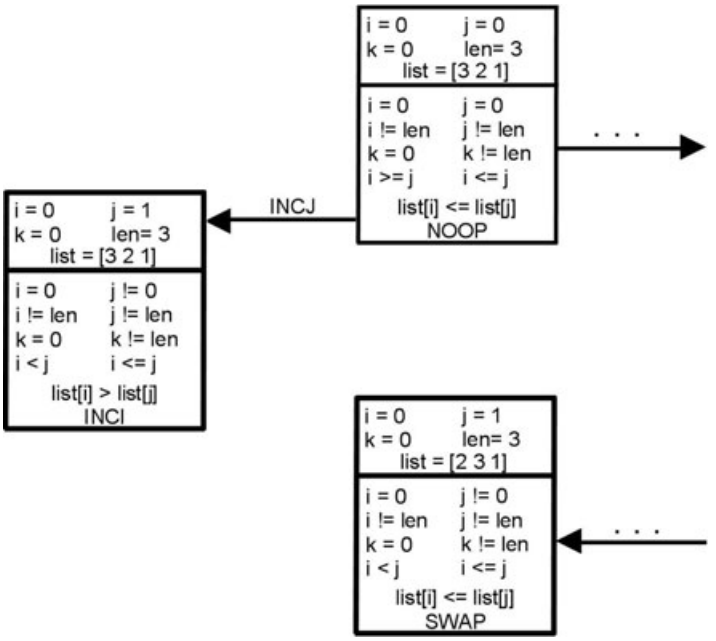

FIGURE 8. State after INCJ.

there is no reward information available for any of the actions from each state during the first iteration.

After action INCJ is chosen next, the bottom right state is added. And finally, Figure 7 shows the system after SWAP. SWAP sorts list elements at positions 0 and 1 with respect to each other. This provides a small positive reward. The random walk continues until TERMINATE is chosen in some state.

Consider now that the state space has been explored until a goal state is reached (the TERMINATE action is executed). Because only one iteration has been performed, the effects of the reward or penalty have not yet reached the starting state. Training repeats with the same list instance, to further explore, grow and shape the state space, many times. This is the reason for the maxed loop in Algorithm 2.

Figure 8 shows the first and last state explored in Figure 7. The ellipses indicate the other states visited in-between. Imagine during the second iteration of training, that the first action performed from the start state is INCJ. Figure 8 shows this. Because the state is new, there is no reward information to guide the search, so the system selects SWAP at random. Figure 9 shows the state after SWAP. It happens that this new state is the last state we described from the in the earlier scenario. We expect states that will ultimately exist in the pruned policy to be visited many, many times. Training continues until TERMINATE is performed.

This repeating process proceeds for 200,000 iterations, after which the policy is tested against all the lists that are the same length as those used for training (size 3 in this case). If a list fails to be sorted by the policy, it is added to the training set, and training starts over.

One of two possibilities occur when the TERMINATE action is executed during training: when a list is unsorted, or when a list is sorted. The reward for executing TERMINATE action on an unsorted list is -160 (a large negative number). The system rapidly learns that the TERMINATE action should not be used in that state. If the list is sorted when TERMINATE

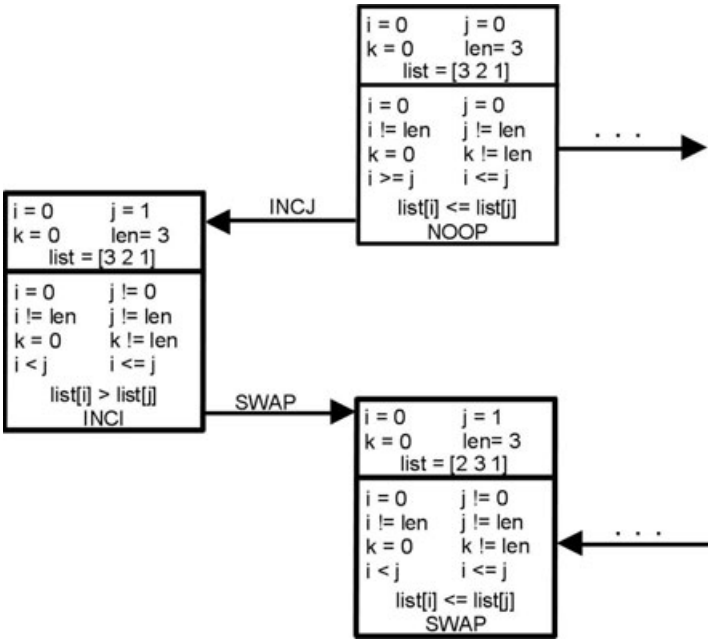

FIGURE 9. State after SWAP in iteration 2.

is performed, the reward is 100 (a large positive number). Over time, the positive and negative rewards will propagate back through the respective sequence of states and actions that lead to them.

5. RESULTS

This section describes the results obtained from experiments with an implementation of the RP system. The size of the training set is described. Then the learned policy is presented. The automatically generated decision tree is shown. An average-case comparison between the learned sorting algorithm and other common sorting algorithms is shown. Finally, a comparison between GP and RP for developing a sort is given.

Source code for RPSort and the Binary Addition problems is available from the authors on the web at http://macs.citadel.edu/rudolphg/rp/index.html. The code should compile and run on any Unix or Linux workstation or desktop, as-is. Other platforms will require some changes to the code.

5.1. The Complexity of Learning

The most interesting and surprising result of our experiments with RP have to do with training and the complexity of generating a generalized sort algorithm. Specifically, the length of list required for learning, the number of lists in the training set, the duration of training, and the size of the system, tells us that learning an iterative, in-place sort is not very complex.

Experiments showed that lists of size 4 are necessary and sufficient for learning. The training set required only one list in 90–95% of training runs: the list [4, 3, 2, 1]. Longer

TABLE 5. RPSort Policy.

ID	Last Action	i	j	i vs. j	list[i] vs. list[j]	Action to Perform
1	NOOP	i=0	j=0	$i=j$	list[i] $\leq$ list[j]	INCJ
2	INCI	i=?	j=?	$i=j$	list[i] $\leq$ list[j]	INCJ
3	INCI	i=?	j=?	$i<j$	list[i] $>$ list[j]	SWAP
4	INCI	i=?	j=?	$i<j$	list[i] $\leq$ list[j]	INCJ
5	INCI	i=?	j=len	$i<j$	list[i] $\leq$ list[j]	SETJZERO
6	INCI	i=len	j=len	$i=j$	list[i] $\leq$ list[j]	TERMINATE
7	INCJ	i=?	j=?	$i<j$	list[i] $>$ list[j]	SWAP
8	INCJ	i=?	j=?	$i<j$	list[i] $\leq$ list[j]	INCI
9	INCJ	i=?	j=len	$i<j$	list[i] $\leq$ list[j]	INCI
10	INCJ	i=0	j=?	$i<j$	list[i] $>$ list[j]	SWAP
11	INCJ	i=0	j=?	$i<j$	list[i] $\leq$ list[j]	INCI
12	INCJ	i=0	j=len	$i<j$	list[i] $\leq$ list[j]	SETJZERO
13	SETIZERO	i=0	j=0	$i=j$	list[i] $\leq$ list[j]	INCJ
14	SETJZERO	i=?	j=0	$i>j$	list[i] $>$ list[j]	SETIZERO
15	SETJZERO	i=0	j=0	$i=j$	list[i] $\leq$ list[j]	INCJ
16	SWAP	i=?	j=?	$i=j$	list[i] $\leq$ list[j]	INCJ
17	SWAP	i=0	j=?	$i<j$	list[i] $\leq$ list[j]	INCJ

lists generally came up with the same solution, with some minor random variations due to the nondeterministic nature of the technique.

5.2. Free Variables

Another interesting result was the lack of any use of the free variable k. Recall that free variables are variables that can be used as the algorithm sees fit. They are not logically constrained in the same way bound variables are, only to prevent illegal domain or range errors. Some generated solutions made use of it but only in a minor way. The most common generated result had no use of k at all. Note that if we remove actions related to k from the system, the system learns faster, but the generated result is the same. We would expect such a speedup if RPSort is working correctly.

The use of, and necessity for, free variables is an open question for future research.

5.3. The RP-Generated Algorithm

Before pruning unnecessary states, the most commonly generated policy contains 226 states. Pruning reduces this to 17 states. Table 5 shows the pruned policy. Note that the ID column is not a part of the policy, it is for ease of reference only.

Listing 1 displays the program corresponding to the most common policy generated by RPSort. The main while-loop keeps the policy executing until a goal state is reached. The body of the loop is the decision tree (in pseudocode) translated from the learned policy. The variables of the program are drawn from the variables in the data-specific portion of the state, and the conditionals are drawn from the RP-specific portion of the state.

LISTING 1. RP-Generated Sort Algorithm.

```
/*list[] is list to be sorted; */
len = list.length;
i=0;
j=0;
lastact=NOOP;

while (lastact != TERMINATE) {
   if (j != 0)
   {
      if (j != len)
      {
         if (list[i] > list[j])
         {
            lastact=SWAP;
            swap(list, i, j);
         }
         else{
            if (lastact == INCI)
            {
               lastact=INCJ;
               j++;
            }
            else{
               if (lastact == INCJ)
               {
                  lastact=INCI;
                  i++;
               }
               else{
                  if (lastact == SWAP)
                  {
                     lastact\gets (INCJ);
                     j++;
                  }
               }
            }
         }
      }
      else{
         if (lastact == INCI)
         {
            if (i != len)
            {
               lastact=SETJZERO;
               j=0;
            }
            else
            {
               lastact=TERMINATE;
            }
         }
         else
         {
            if (lastact == INCJ)
            {
               if (i != 0)
               {
                  lastact=INCI;

                  i++;
               }
               else{
                  lastact=SETJZERO;
                  j=0;
               }
            }
         }
      }
   }
   else{
      if (i != 0)
      {
         lastact=SETIZERO;
         i=0;
      }
      else{
         lastact=INCJ;
         j=0;
      }
   }
}
```

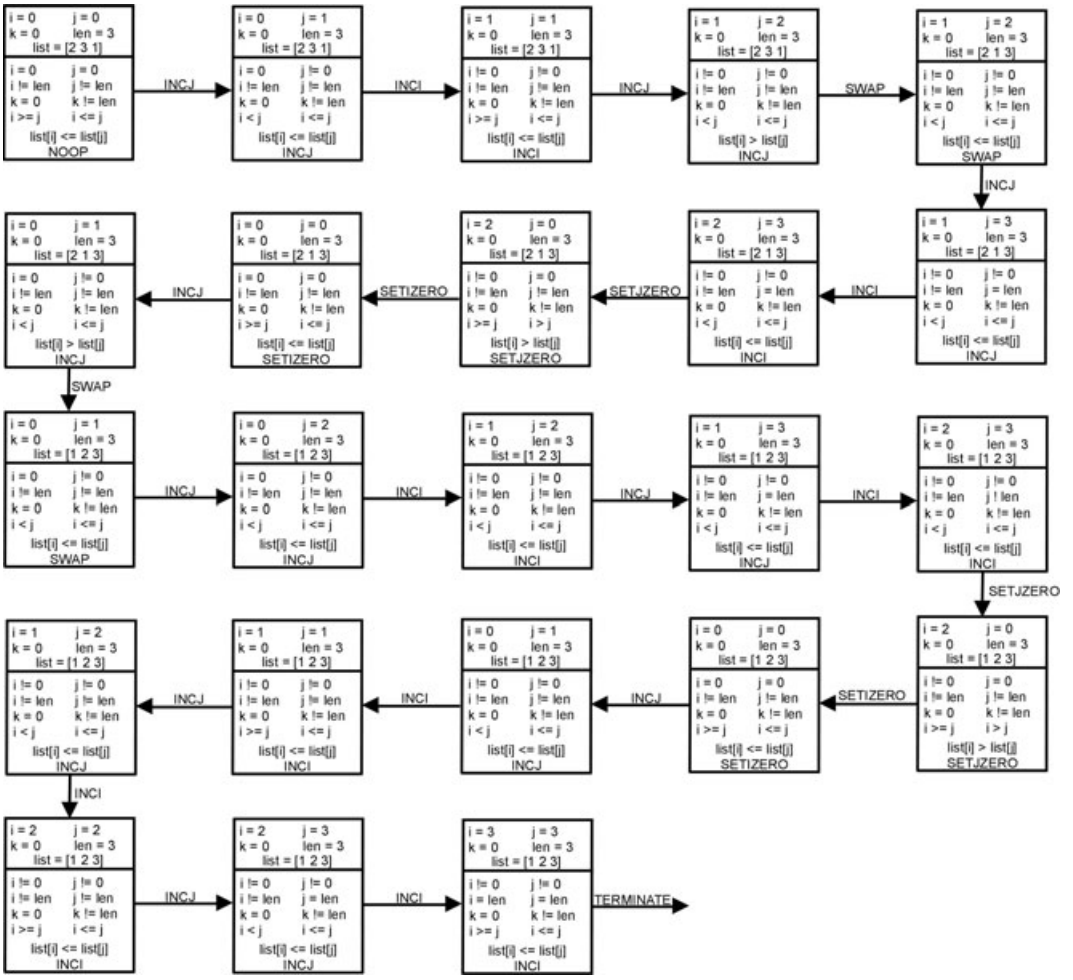

FIGURE 10. Sorting list of length 3.

A comparison with a simple depth-first search for a working program would be interesting, given the time and complexity bounds established here by RPSort. This is a topic for potential future work.

5.4. Proof that RPSort Is a Generalized Algorithm

Figure 10 demonstrates the state transitions that occur while sorting the list [2, 3, 1]. Each state is labeled according to the ID column in Table 5. The variable k is not included in the table because it is not used in the generated algorithm.

It is impossible to exhaustively check lists of all possible lengths to guarantee algorithm correctness. However, we can use a proof. Suppose we partition the set of all possible lists into two classes: lists already sorted in ascending order, and lists with at least one pair of adjacent elements out of order. We show that the algorithm will sort each class of lists correctly. The length of list is irrelevant.

Assume the algorithm has been passed a list sorted in ascending order. Whenever $i = j$, INCJ is performed. Whenever $list[i] \leq list[j]$ and $lastact = INCJ$, INCI is performed.

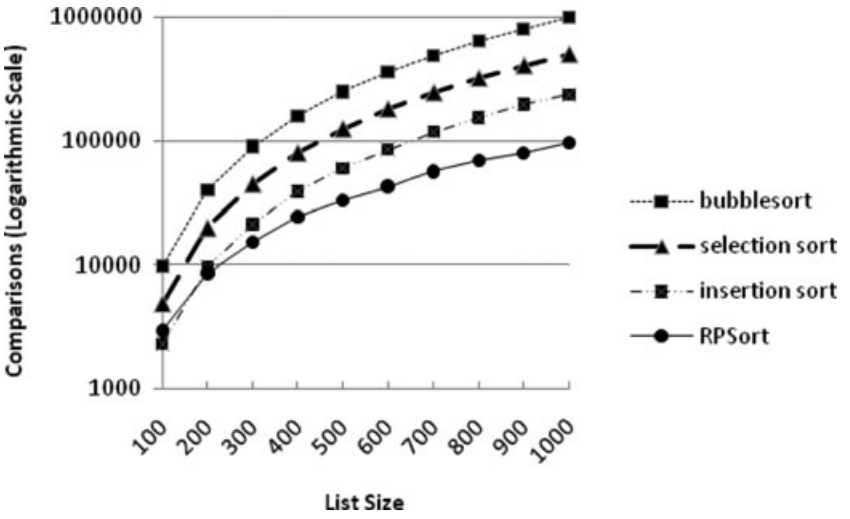

FIGURE 11. RPSort compared to other iterative, in-place sorts.

Because the list is sorted, the algorithm alternates between performing INCJ and INCI, until $j = len$, regardless of the length of the list. When $j = len$ and $lastact = INCJ$, INCI is performed. At this point, $i = len$ and $j = len$, because both i and j have been incremented len times. The algorithm terminates when $i = len$ and $j = len$. Thus, algorithm terminates correctly when the list is sorted.

Now assume the algorithm has been passed an unsorted list. Two adjacent elements that are out of order with respect to each other must exist in the list, by definition. The sublist up to that point in the list is sorted. Thus i and j will be alternately incremented, as if the list were sorted, up to that point, until that adjacent out-of-order pair is reached. When that pair is reached, the last action performed will have been INCJ. Prior to that action, $i = j$. When that pair is reached, $i = j - 1$ and $list[i] > list[j]$. The adjacent out-of-order pair is swapped, putting them in order with respect to each other. INCJ is then performed. At this point, $i = j - 2$. This gap increases each time a pair such that $list[i] > list[j]$ is found, until $j = len$. When $j = len$, the last action performed is INCJ. The algorithm specifies that INCI is performed next. Because i will be at most $j - 2$ before this action is performed, incrementing i will not make $i = len$. The algorithm will therefore set $i = 0$ and $j = 0$, and begin again. Any swaps made during a given pass through the list cannot unsort the list, and at least one more pair is relatively in order. When only two elements are out of place, the algorithm will not terminate until at least the next pass through the list.

We have shown that RPSort will sort any list correctly. Therefore, RPSort is a generalized sort.

5.5. Comparing Various Sorts

Average case growth-rate comparisons were performed for RPSort, Bubble Sort, Selection Sort, and Insertion Sort. For Bubblesort, Selection Sort, and Insertion Sort, the average case was directly calculable. The average case for RPSort was approximated by randomly selecting 100,000 lists for list lengths of 100 to 1,000 (at intervals of 100) and determining the average number of comparisons needed to sort the lists. Figure 11 contains the results. Note that the y-axis is on a logarithmic scale.

Bubblesort is the worst of the four algorithms from the start, followed by Selection Sort, then RPSort. Insertion sort starts off as the best of the four sorting algorithms. Between list sizes 150 and 175, RPSort starts using the least number of comparisons on average. This suggests that for nontrivial lists, RPSort is superior to the other three algorithms.

5.6. Is RPSort Really Learning?

We wanted to ascertain whether RP is just randomly creating programs until a successful program is found, or performing a directed search. We used Random Program Generation (RPG) to randomly create a computer program using the same state representation used by RP. The entire state space explored by a given RP application is iterated over, and each state is assigned a random action. Thus, an entire policy is created at random. The random policy is then executed in the same fashion as that learned by RP. By executing the policy on every training instance used in RP, the random policy can be compared to the policy learned by RP. One billion random policies were created and executed on all the lists of size 4. No random policy successfully sorted all of the lists. Because a generalized sort was learned by RP within 200,000 iterations, it is clear that, in this case, RP performs much better than RPG. In particular, RP is not merely creating random programs until a successful one is found.

5.7. Comparing RP and GP on Generating a Sort

RP appears more efficient at generating a sorting algorithm than GP when comparing the results in this section with those obtained by Kinnear. In his work, a single run consisted of 1,000 population members operated on for 49 generations. Each population member at each generation was tested on 55 randomly generated lists. This results in at least two million evaluations. Because each fitness check is on a mostly new set of lists, storing fitness for population members carried forward cannot be done. In learning RPSort, the RP system, evaluates and updates the policy 200,000 times for the list [4, 3, 2, 1], a much smaller number of evaluations.

RP also converges consistently to an algorithm that can sort a list of any size. Even if GP converged to a solution that sorted the lists used to evaluate fitness, generality was not assured without strict complexity penalties. The algorithms developed by GP to sort the lists that actually did generalize were inefficient algorithms. RPSort, on the other hand, has a very efficient average case.

Last, the program developed by RP for sorting does not require any hand simplification (having a human alter the code) to make it easy to understand, whereas even the simplest of the sorts evolved by GP in Kinnear's papers requires simplifying to make the program understandable without careful examination of the resulting code. Furthermore, the policy-based structure of the resulting RP program is more natural to computers than the tree structure of GP-developed programs.

5.8. Limitations of RP

RP is limited only by the limitations of the underlying algorithms or models being used. For example, if an algorithm other than Q-Learning is used to generate a policy, the number of states and the size of the lookup table may not be a concern. If a learned policy is translated into something other than a decision tree, the use of an exhaustive algorithm to search for an optimal tree would not be an issue. Some of these issues are more fully discussed in the next section.

TABLE 6. Full Adder Inputs and Outputs.

X3	X2	X1	A1	A0
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

TABLE 7. Full Adder Action Set.

Action	Description
$NOOP$	A "nonaction." Not selectable by the agent
$TERMINATE$	Ends policy execution
$A1 = True$	Sets output $A1$ to true (1)
$A1 = False$	Sets output $A1$ to false (1)
$A0 = True$	Sets output $A0$ to true (0)
$A0 = False$	Sets output $A0$ to false (0)

6. RP BINARY ADDITION ALGORITHMS

This section briefly lists the states, actions, rewards and the generated algorithms for three binary addition algorithms: a full adder, a binary incrementer, and a binary adder. Each of these uses the same algorithms and techniques as RPSort. Each one is an extension of the previous one. The purpose is to show that RP has been used for more than just sorting. White (2006) gives more detail on these for the interested reader.

6.1. Full Adder

The full adder circuit is a circuit that takes in, as input, three bits (single-digit binary numbers) and outputs the sum of those bits as a two-digit binary number. X1, X2 and X3 are the inputs, A0 and A1 are the outputs, shown in Table 6.

The state representation is very simple. It consists of three Boolean inputs, and two integer outputs. The outputs are integers because they are tri-state variables. A value of -1 means that the variable has not been initialized yet, otherwise the value refers to the actual output (0 and 1). The state also contains a reference to the last action performed. There is no data-specific portion, the entire state is RP-specific.

The system used five actions, shown in Table 7. As with RPSort, any state in which the last action performed is $TERMINATE$ is a potential goal state. If any output variable is uninitialized or if the output variables do not match the correct output when in the goal state, the transition from the previous state to the goal state receives a penalty. Otherwise, the transition receives a reward.

The system is trained on all training instances at each iteration. The most commonly learned policy contains 171 states, which pruning trims to 24 states. Listing 2 shows the

generated algorithm. For presentation purposes, nested if-then-else clauses have been replaced by a case statement. It is straightforward to show, by exhaustive testing, that this algorithm performs correctly.

LISTING 2. Full Adder Algorithm.

```
switch lastact
   case (NOOP)
      if (!X1)
         if (!X2)
            if (!Cin)
               S = False;
            else
               Cout = False;
         else
            if (!Cin)
               Cout = False;
            else
               Cout = True;
      else
         if (!X2)
            if (!Cin)
               Cout = False;
            else
               Cout = True;
         else
            if (!Cin)
               Cout = True;
            else
               S = True;
   case (Cout=True)
      if (S < 0)
         S = True;
      else
         if(S)
            TERMINATE;
   case (S=True)
      if  (Cout < 0)
         Cout  = True;
      else
         if (Cout = True)
            TERMINATE;
   case (S=False)
      if (Cout < 0)
         Cout = False;
      else
         if (Cout)
            TERMINATE;
```

6.2. Binary Incrementer

A binary incrementer takes a binary number of any length and modifies the bits so that the resulting binary number has a value of one plus the original binary number. In the special case when the input is all 1's, the result of adding 1 is all 0's, ignoring the overflow.

TABLE 8. Binary Incrementer Actions.

Action	Description
NOOP	A Initial action, not selectable by the agent
TERMINATE	Ends policy execution
INCI	Increments i
SETIZERO	Sets $i = 0$
INCJ	Increments j
SETJZERO	Sets $j = 0$
SETTRUE	Sets $dest[i]$ to 1
SETFALSE	Sets $dest[i]$ to 0

The training set includes all eight 3-bit numbers. Experiments showed that 3 bits are necessary and sufficient to generate a generalized binary incrementer algorithm. Fewer than 3 bits led to nongeneral solutions. More than 3 bits led to the same solution as that learned using 3 bits.

The data-specific portion of the state has five integers: i, j, $source$, $dest$ and len. i is a bound variable, and j is a free variable. $source$ and $dest$ are initialized from the current training instance. len stores how many bits are in the training instance. The RP-specific portion contains several Boolean flags indicating whether $i = 0$, $j = 0$, $i = len$, and $j = len$, and the last action performed. It also contains the values of the following bits: $source[i]$, $dest[i]$, $source[i-1]$, $dest[i-1]$. The inputs are treated as binary sequences, and are indexed from 0 to $len - 1$. If $i = 0$, then $source[i-1] = 0$ and $dest[i-1] = 0$. If $i = len$, $source[i] = 0$, and $dest[i] = 0$. Similar to RPSort, these constraints avoid array out-of-bounds issues.

Table 8 lists the actions used for this problem. Any state where $TERMINATE$ is the last action performed is a potential goal state. The following restriction is placed on action selection: if $i = len$, then $SETFALSE$ and $SETTRUE$ cannot be chosen. This avoids out-of-bounds array errors.

LISTING 3. Binary Incrementer Algorithm.

```
if (i  != len)
   if(source[i]=0)
      if (dest[i]=0)
         SETTRUE;
      else
         TERMINATE;
   else
      if (dest[i]=0)
         INCI;
      else
         SETFALSE;
else
   TERMINATE;
```

A reward is provided when the system executes $TERMINATE$ and $dest$ is the successor to $source$ (all bits are set to the correct values). A penalty is given if $TERMINATE$ is executed in any other situation.

The system is trained on all training instances during each iteration. The most commonly learned policy contains 254 states. Pruning trims this to a mere 9 states.

We prove that the algorithm is a generalized incrementer by analyzing its behavior. Starting at the least-significant bit, the algorithm scans the number. At each location, it inverts the bit and increments i either until it sets a bit to 1 or until $i = len$. This is the exact procedure for incrementing a binary number.

6.3. General Binary Adder

A general binary adder takes in two binary numbers of any length (number of digits) and returns a binary number representing their sum. If inputs have different numbers of bits, the shorter of the two is padded with beginning zeros so that the numbers have the same number of bits. The output number is also the same length as the inputs. In the event that the sum of the two input binary numbers requires more binary digits to represent it than are available, the overflow is ignored. For example, $110 + 101 = 1,011$, however the binary adder would return $110 + 101 = 011$. In effect, this is modular addition. It was an implementation decision to require the solution to have the same number of bits as the inputs. The general binary adder problem we formulate as learning a generalized many-to-one function that maps inputs to an output based on examples.

The training set consists of the cross product of the set of 4-bit numbers with itself (because two binary number inputs are needed). 4 bits is the smallest number that is necessary and sufficient to learn a generalized binary adder. Experiments showed that 3 bits or fewer did not lead to a generalized adder, while 4 bits or more did.

The data-specific portion of the state has two variables, i and j, the two input binary numbers, $source1$ and $source2$, the binary number that $source1$ and $source2$ maps to, called map, $dest$, an array of integers the same length as the input numbers and a variable len, which is the length (in number of bits) of the input numbers. i is a bound variable and j is a free variable. Any element in $dest$ can have one of three values: -1, 0, and 1. -1 means that specific position is uninitialized. The other two values represent their corresponding binary values. The RP-specific portion of the state has four Boolean flags, the last action performed and variables $zcount$, $ocount$, $lastzcount$, $lastocount$, $dloc$ and $lastdloc$. The flags indicate whether $i = 0$, $i = len$, $j = 0$, $j = len$. $zcount$ indicates how many zeros are indexed by $source1[i]$ and $source2[i]$ (either 0, 1, or 2 zeros for a given position i). $ocount$ indicates how many ones are indexed by $source1[i]$ and $source2[i]$. $lastzcount$ and $lastocount$ track how many zeros and ones (respectively) are referenced by $source1[i-1]$ and $source2[i-1]$. $dloc$ and $lastdloc$ reference the value in $dest[i]$ and $dest[i-1]$, respectively. All bit sequences are referenced as a zero-based array (0 to $len - 1$). If $i = 0$, then $lastzcount = 2$, $lastocount = 0$, and $lastdloc = -1$. If $i = len$, then $zcount = 2, ocount = 0$, and $dloc = -1$. These constraints avoid array index out-of-bounds errors.

The actions used by the general binary adder system are the same ones used by the binary incrementer, shown previously in Table 8. Any state where the last action performed is $TERMINATE$ is a potential goal state. A reward is given when the system terminates with $dest$ initialized and set to the correct answer. Otherwise, a penalty is given. If $SETTRUE$ or $SETFALSE$ sets $dest[i]$ to match $map[i]$, a reward is given of $\frac{1}{10}^{th}$ the termination reward. Otherwise, a penalty of $\frac{1}{10}^{th}$ the termination penalty is given. Note

that the system is not using addition to learn an addition algorithm. The system is learning a mapping between given inputs and outputs, and so is rewarded for finding a partial match.

Some actions are not possible from some states. $SETTRUE$ and $SETFALSE$ cannot be performed if $i = len$, to avoid array index out-of-bounds errors. If $dloc = 1$ then $SETTRUE$ cannot be performed. if $dloc = 0$ then $SETFALSE$ cannot be performed. These constraints prevent the system from gaining potential infinite rewards by repeatedly performing a reward-giving action.

The system is trained on all training instances for each iteration. The most commonly learned policy contains 913 states. Pruning trims this to 48 states. Listing 4 shows the generated algorithm.

As with the binary incrementer, there is no way to exhaustively test all possible inputs to determine algorithm correctness. However White et al. (2010) gives a proof of correctness.

7. RELATED AND FUTURE WORK

This section discusses aspects of RP and RPSort in relation to other efforts. Several areas for potential future work with RP are discussed, as it was convenient to include them in the relevant section.

7.1. Genetic Programming

GP has been discussed in Section 2, and in various comparisons in other sections. Both RL and GAs attempt to find a solution through a combination of stochastic exploration and the exploitation of the properties of the problem. The difference between RL and GA lies in the problem formulation and the technique used to solve the problem. Similarly, RP and GP both use stochastic exploration combined with exploitation, but the techniques and program representations differ dramatically.

Potential future work includes: More extensive comparisons between RPSort and related GP-generated sorts, involving runtimes, complexity measures and other metrics; developing RP solutions to other problems that GP has been used to solve.

7.2. Reinforcement Learning

The classic Q-Learning algorithm has three characteristics that have to be dealt with to use it successfully in practical implementations:

(1) The table of (state, action) pairs becomes prohibitive for large numbers of states, if a fixed-size lookup table is used.
(2) The proper balance between stochastic exploration and exploitation of prior learning must be determined, to avoid local minima.
(3) Q-Learning is an off-policy algorithm, meaning that it learns a different policy than the one it follows while learning.

The data structure used to store the pairs for RPSort is a Binary Search Tree. This provides fast lookup times for large trees as long as the tree is nearly balanced. It assumes

LISTING 4. Binary Adder Algorithm.

```
if (i  != len)
   if (dloc < 0)
      switch zcount
         case(0)
            if (lastzcount = 0)
               SETTRUE;
            else
               if (lastzcount = 1)
                  if (lastdloc = 0)
                     SETTRUE;
                  else
                     if (lastdloc = 1)
                        SETFALSE;
               else
                  if(lastzcount = 2)
                     SETFALSE;
         case(1)
            if (lastzcount = 0)
               SETFALSE;
            else
               if (lastzcount = 1)
                  if (lastdloc = 0)
                     SETFALSE;
                  else
                     if (lastdloc = 1)
                        SETTRUE;
               else
                  if (lastzcount = 2)
                     SETTRUE;
         case(2)
            if (lastzcount = 0)
               SETTRUE;
            else
               if (lastzcount = 1)
                  if (lastdloc = 0)
                     SETTRUE;
                  else
                     if (lastdloc = 1)
                        SETFALSE;
               else
                  if (lastzcount = 2)
                     SETFALSE;
   else
      if (dloc = 0)
         INCI;
      else
         if (dloc = 1)
            INCI;
else
   TERMINATE;
```

that the number of (state,action) pairs stored in the tree is small compared to the total possible number of pairs.

This approach avoids the complexity, unlearning problems and sensitivity to initial parameters of using neural networks as function approximators (Whiteson and Stone 2006).

Nevertheless, as the size of the search tree grows large, it may suffer from the same size problems as other table-driven approaches. More recent models such as NEAT+Q (Whiteson and Stone 2006), LSPI (Lagoudakis, Parr, and Bartlett 2003), and KLSPI (Xu, Hu, and Lu 2007) show promise in overcoming the limitations of the neural net-based function approximators. Algorithms like these could be suitable replacements for Q-Learning in RP. Algorithms like KLSPI have an added benefit. KLSPI incorporates a pruning mechanism as part of training. Such algorithms remove the need for a separate pruning step to optimize the learned policy after learning, which is needed when using Q-Learning.

Another interesting alternative to Q-Learning to consider is Tree-Based Batch Mode RL (Ernst, Geurts, and Wehenkel 2005). This algorithm formulates the problem as a sequence of kernel-based regression problems, and makes it possible to use any regression algorithm, rather than stochastic approximation algorithms. The main requirement for its use in RP would appear to be converting any training set to the four-tuples required by the algorithm.

The need to balance exploration and exploitation is something that RP shares with GP and other evolutionary algorithms. These parameters were determined empirically for RPSort.

The off-policy characteristic is not a problem for RPSort. What matters is the correctness of the learned policy, and the efficiency of the learning algorithm. The method does not necessarily have to match the result, because this is not an in situ control program.

7.3. Other Automatic Generation Methods

There are many theoretical approaches to automatic program synthesis, other than GP. Srivastava, Gulwani, and Foster (2010) discuss one example, which has similar goals, and demonstrates the program synthesis using sorts, mathematical problems and line drawing algorithms. However, the mechanisms used are much different than RL mechanisms.

Srivastava et al.'s synthesis method treats program generation as a generalization of program verification. Programs are specified using sequences of formal logic clauses as preconditions, postconditions and control-flow templates. Program statements are inferred in the process of constructing solutions that satisfy the specified constraints. In essence, this approach treats program generation as a constraint-satisfaction problem.

Some of the algorithms developed, such as Bubblesort and Selection Sort, we have discussed in connection with RPSort. The recursive sorts Quicksort and Mergesort, Srivastava et al. have developed using control-flow templates to guide behavior. The recursive class of sorts is an area considered for future work in RP.

None of the programs mentioned above represent novel algorithms. RPSort, on the other hand, is a novel sorting algorithm that outperforms the other iterative, in-place sorts. Using automated program generation to discover novel mechanisms is likely to be valuable in the future, much as chess-playing programs have helped discover new lines in chess.

There is another important point of contrast. Using the verification method seems to require that a programmer have training in Formal Methods, because programs are generated from formal, axiomatic specifications. RP shares some of that, in the sense that RP uses Boolean flags to specify features that the learning algorithm should generalize over. However, specifying the flags and boundary conditions in an RPState is much less formal than the verification method requires—at least in the problems that have been studied to date. It would be interesting to attempt the matrix multiply or the line drawing algorithm with RP, and compare results with (Srivastava et al. 2010).

7.4. Learning Classifier Systems

Learning Classifier Systems are a class of machine learning algorithms that combine GAs, RL, Supervised Learning (SL) and Rules into a single architecture (Urbanowicz and Moore 2009). A population of rule-based classifiers describes the behavior of some system. A GA is used to evolve better rules over time, and the evolution is influenced and aided by the RL and SL components. Holland's original observation was that "when dealing with complex adaptive systems, finding a single, best-fit model was less desirable than evolving rules that collectively model that system" (Urbanowicz and Moore 2009).

One might argue that RP could be considered a subset of LCS, because both use RL techniques, and because the decision tree code generated by RP could easily be mapped to a set of rules. However, such an assertion could be made about any algorithm that included one, but not all, of the major components of an LCS, and may or may not be useful. Furthermore, the choice to generate code in the form of a decision tree was a convenient preference, not a necessity—future applications using RP may require something different. In this context, one thing that RP demonstrates is that the added complexity of multiple interacting systems is not needed to generate a novel iterative, in-place sorting algorithm. Of course, we cannot generalize this kind of statement to more complex applications.

8. CONCLUSION

In the paper, we introduced using RP to automatically generate programs. We used RP to generate an iterative, in-place sort algorithm, RPSort. We compared these results to Kinnear's earlier work generating sorts with GP and demonstrated empirically that RP improves on those earlier results in several ways. RP generates a novel, fast, efficient, general sorting algorithm. The algorithm is faster than the most common iterative sorting algorithms available. RP generates a more efficient algorithm than GP. Our formulation uses operations that are more primitive (basic) than those operations than were used to evolve Bubblesort through GP. The state representation and transition is more "natural" to a computer than the function tree structure used by GP.

A surprising result gives practical bounds on the empirical complexity of learning in-place, iterative sorts. Using RP, we found that:

- a list of size 4 is sufficient to learn an algorithm that will sort a list of any size,
- the smallest training set required for correct learning includes only the list sorted in reverse order, and
- RP learned the sorting algorithm in less than 200,000 iterations.

This reduces the empirical complexity of learning an in-place iterative sort to about the same complexity as learning TicTacToe.

We also described briefly how RP was used to generate three binary addition algorithms. Some potential future applications for RP include creating a parity-checking function, an algorithm for solving two-variable linear systems, and a function for one-dimensional pattern recognition. While these applications are trivial, they will further explore the strengths and limitations of RP. It is expected that as RP is further explored, it will be able to handle real-world applications. It is interesting to consider generating efficient parallel code as well as code for single-CPU applications.

Future work regarding sorting algorithms includes simulating a Turing-complete language and using RP to find a Turing machine that sorts any list. The appeal behind this

approach is that it is more flexible than the approach described in this paper. It will allow for variable creation, the creation of data structures, and the possibility of learning recursion. Other possible work involves exploring different techniques of handling partially observable Markov Decision Processes and non-Markovian RL problems and applying those techniques to RP. An important future task involves *stack equivalence*. To allow the dynamic creation of variables in an RP environment, the RP-specific portion of the state must be able to determine if two memory stacks are equivalent. This will enable much more robust and general RP applications.

Work on RP could also contribute to looking at the higher-level question of how to judge a particular algorithm's or method's fitness for a particular task. For example, in some situations, an iterative sort routine is more efficient than a recursive one. In more general terms, suppose we have method A and method B for performing some task. How do we formulate and quantify how a machine might learn to pick one method over the other in the appropriate circumstances?

Additional work involves modeling and implementation. As with many RL applications, the number of possible states can get rapidly large. If there are currently n bound variables, a new bound variable added to the state adds n Booleans to reference its value with the values of all the other bound variables. This results in a $2n$-fold increase in the number of possible states. For RPSort, we used Q-Learning with a lookup table that grows over time, episodic training and nondeterministic rewards. Our contribution to the learning algorithm is to use a lookup table that grows as needed, rather than a fixed-size lookup table, or a neural network function approximator. For RPSort, the number of actual states explored is much less than the total possible number of states, so this seems like a good choice.

Working to simplify and formalize the state representation process may also improve the scalability of potential RP applications. For example, interpolating between (s,a) pairs so that the entire state space does not need to be explicitly represented and learned would have a profound impact on the amount of memory and exploration needed. Further implementation experiments would involve online vs. offline learning and exploration/exploitation strategies and the effects they have on convergence rates and the kind of solution learned by the system.

REFERENCES

BAIRD, L. C. 1995. Residual algorithms: Reinforcement learning with function approximation. *In* International Conference on Machine Learning, Tahoe City, CA, pp. 30–37.

ERNST, D., P. GEURTS, and L. WEHENKEL. 2005. Tree-based batch mode reinforcement learning. Journal of Machine Learning Research, **6**: 503556.

FONSECA, C. M., and P. J. FLEMING. 1993. Genetic algorithms for multiobjective optimization: Formulation, discussion and generalization. *In* Genetic Algorithms: Proceedings of the Fifth International Conference. Morgan Kaufmann: San Francisco, CA, pp. 416–423.

GOLDBERG, D. E. 1989. Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley Longman: Boston, MA.

HOLLAND, J. H. 1975. Adaptation in natural and artificial systems. The University of Michigan Press: Ann Arbor, MI.

JAAKKOLA, T., S. P. SINGH, and M. I. JORDAN. 1995. Reinforcement learning algorithm for partially observable Markov decision problems. In G. Tesauro, D. Touretzky, and T. Leen, editors, Advances in Neural Information Processing Systems, volume 7. The MIT Press: Cambridge, MA, pp. 345–352.

KAELBLING, L. P., M. L. LITTMAN, and A. P. MOORE. 1996. Reinforcement learning: A survey. Journal of Artificial Intelligence Research, **4**: 237–285.

KINNEAR, K. E. 1993a. Evolving a sort: Lessons in genetic programming. *In* Proceedings of the 1993 International Conference on Neural Networks, volume 2. IEEE Press: San Francisco, CA, pp. 881–888.

KINNEAR, K. E. 1993b. Generality and difficulty in genetic programming: Evolving a sort. *In* Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93. *Edited by* S. Forrest. Morgan Kaufmann: San Francisco, CA, pp. 287–294.

KOZA, J. 1990. Genetic programming: A paradigm for genetically breeding populations of computer programs to solve problems. Technical Report STAN-CS-90-1314, Department of Computer Science, Stanford University, Palo Alto, CA.

KOZA, J. R. 1989. Hierarchical genetic algorithms operating on populations of computer programs. *In* Proceedings of the Eleventh International Joint Conference on Artificial Intelligence IJCAI-89. *Edited by* N. S. Sridharan, volume 1. Morgan Kaufmann: San Francisco, CA, pp. 768–774.

KOZA, J. R. 1992. Hierarchical automatic function definition in genetic programming. *In* Foundations of Genetic Algorithms 2. *Edited by* L. D. Whitley. Morgan Kaufmann: San Francisco, CA, pp. 297–318.

KOZA, J. R., F. H. Bennett III, J. L. HUTCHINGS, S. L. BADE, M. A. KEANE, and D. ANDRE. 1997. Evolving sorting networks using genetic programming and rapidly reconfigurable field-programmable gate arrays. *In* Workshop on Evolvable Systems. International Joint Conference on Artificial Intelligence, Nagoya, Japan, pp. 27–32.

LAGOUDAKIS, M. G., R. PARR, and L. BARTLETT. 2003. Least-squares policy iteration. Journal of Machine Learning Research, **4**: 1107–1149.

MASSEY, P., J. A. CLARK, and S. STEPNEY. 2005. Evolution of a human-competitive quantum Fourier transform algorithm using genetic programming. *In* GECCO '05: Proceedings of the 2005 Conference on Genetic and Evolutionary Computation. ACM Press: New York, pp. 1657–1663.

MCGOVERN, A., and A. G. BARTO. 2001. Automatic discovery of subgoals in reinforcement learning using diverse density. *In* Proceedings of the 18th International Conference on Machine Learning. Morgan Kaufmann: San Francisco, CA, pp. 361–368.

MITCHELL, T. M. 1997. Machine Learning. McGraw-Hill: New York.

NONAS, E. 1998. Optimising a rule based agent using a genetic algorithm. Technical Report TR-98-07, Department of Computer Science, King's College London, UK.

SEKANINA, L., and M. BIDLO. 2005. Evolutionary design of arbitrarily large sorting networks using development. Genetic Programming and Evolvable Machines, **6**(3): 319–347.

SPECTOR, L., H. BARNUM, H. J. BERNSTEIN, and N. SWAMY. 1999. Finding a better-than-classical quantum and/or algorithm using genetic programming. *In* Proceedings of 1999 Congress on Evolutionary Computation, Washington, DC, pp. 2239–2246.

SRIVASTAVA, S., S. GULWANI, and J. S. FOSTER. 2010. From program verification to program synthesis. *In* POPL '10: Proceedings of the 37th ACM SIGACT-SIGPLAN Conference on Principles of Programming Languages, Madrid, Spain.

SUTTON, R. S., and A. G. BARTO. 1998. Reinforcement Learning: An Introduction. MIT Press: Cambridge, MA.

SUTTON, R. S., D. PRECUP, and S. P. SINGH. 1999. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. Artificial Intelligence, **112**(1-2): 181–211.

URBANOWICZ, R. J., and J. H. MOORE. 2009. Learning classifier systems: A complete introduction, review, and roadmap. Journal of Artificial Evolution and Applications, **2009**. doi: 10.1155/2009/736398.

WATKINS, C. J. 1989. Learning from delayed rewards. Ph.D. thesis, Cambridge University, Cambridge, UK.

WHITE, S., T. R. MARTINEZ, and G. RUDOLPH. 2010. Generating three binary addition algorithms using reinforcement programming. *In* Proceedings of the 48th Annual Southeast Regional Conference (ACMSE '10). ACM Press: New York. DOI: 10.1145/1900008.1900072 http://doi.acm.org/10.1145/1900008.1900072

WHITE, S. K. 2006. Reinforcement Programming: A New Technique in Automatic Algorithm Development. Master's thesis, Brigham Young University, Provo, UT.

WHITESON, S., and P. STONE. 2006. Evolutionary function approximation for reinforcement learning. Journal of Machine Learning Research, **7**: 877–917.

XU, X., D. HU, and X. LU. 2007. Kernel-based least squares policy iteration for reinforcement learning. IEEE Transactions on Neural Networks, **18**(4): 973–992.