WCCI 2010 IEEE World Congress on Computational Intelligence
July, 18-23, 2010 - CCIB, Barcelona, Spain CEC IEEE

Generating a Novel Sort Algorithm using Reinforcement Programming

Spencer K. White, Tony Martinez and George Rudolph

Abstract— **Reinforcement Programming (RP) is a new approach to automatically generating algorithms, that uses reinforcement learning techniques. This paper describes the RP approach and gives results of experiments using RP to generate a generalized, in-place, iterative sort algorithm. The RP approach improves on earlier results that 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. Results 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. RP has also been used to generate three binary addition algorithms: a full adder, a binary incrementer, and a binary adder.**

I. Introduction

Reinforcement Programming (RP) is a new technique to automatically generate algorithms. This paper introduces RP by showing how RP can be used to solve the task of learning an iterative in-place sorting algorithm. Some comaprisons with earlier work by Kinnear [1], [2], which 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.

Perhaps the most important result gives practical bounds on the empirical complexity of learning in-place, iterative sorts. Using RP, we found empirically that:

- a 4-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, and
- RPSort learns in less than 200,000 iterations, 90% of the time.

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 Reinforcement Learning (RL), the Q-Learning algorithm [3], [4], [5], with GP and with Genetic Algorithms (GAs).

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.

GAs [6] [7], [8], [9] 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 Reinforcement Learning (RL) [10], [11], [12], [13], [14], 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 reinforcement learning 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 Reinforcement Learning (RL) has been used successfully in game playing and robot control applications [4], [5]. 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, non-deterministic 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, Q-value) pairings 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.

RP techniques have been used to generate code for more than just sorting: a full adder, a binary incrementer, and a binary adder [15]. This paper focuses on a sorting algorithm, however.

Section II reviews Kinnear's technique for using GP to

Spencer White may be contacted at 27921 NE 152nd St. Duvall, Washington, USA, 98019 (email: Skwhite314@gmail.com).
Tony Martinez is with the Computer Science Department, Brigham Young University, Provo, Utah, USA 84602 (email: martinez@cs.byu.edu).
George Rudolph is with the Department of Mathematics and Computer Science, The Citadel, Charleston, South Carolina, USA 29409 (email: george.rudolph@citadel.edu).

learn a generalized in-place iterative sorting algorithm. Section III discusses RP concepts. Section IV describes training the RP system to generate RPSort. Section V describes results of experiments with RPSort, compares those results to Kinnear's earlier GP results, and briefly describes results for the binary addition algorithms. Section VI is the conclusion.

II. A Review of GP-generated Sorts

Genetic Programming most commonly involves using genetic algorithms to modify a population of expression trees, where each individual tree is an executable program. [16], [17]. This functional approach to program structure and behavior is powerful and elegant. The set of usable functions is limited only by what can be represented as a function. The tree structure allows a GP to represent arbitrarily complex programs in a uniform way, and execute them in Functional languages such as LISP.

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. 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.

The "most fit" sort Kinnear's system generated was a BubbleSort. Obviously, it is general enough to sort any size list, but it is inefficient for large lists. There are several other problems as well. 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. RP overcomes these issues.

III. States, Actions, Transitions and Rewards

This section briefly describes the states, actions, transitions and rewards used for RPSort.

A. States

Figure 1 shows an example of a state for RPSort. 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 corresponds roughly to the "external environment" or agent inputs. The RP-specific portion contains relationships between the data and variables that allow generalization. In RPSort, the relationships and constraints contained in the RP-specific portion of the state translate into the conditionals that handle program control flow. The RP-specific portion of each state corresponds roughly to a portion of an agent's internal state.

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

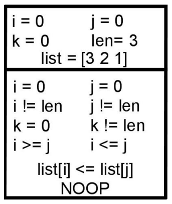

Fig. 1. An example state structure for RPSort.

j are both bound variables. k is a free variable. A variable that directly references the input, or has an explicit purpose is called a *bound variable*. A *free variable* has no predefined purpose. We included free variables in our experiments in order to see what the system would do with them as it learned.

The RP-specific portion of the state contains a set of Boolean flags and the last action 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. Including the action in the state, as a practical matter, also simplifies the process of associating (state,action) pairs with q-values and rewards during training.

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. This distinction is what allows generalization–one RP-specific state can represent many possible states.

B. Actions, Transitions and Rewards

Table I lists the actions for RPSort, which 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, they can not be incremented. This is to avoid out-of-bound array errors.

Only two actions give immediate rewards, TERMINATE and SWAP. Any state in which the last action performed is TERMINATE is a potential goal state. There is a reward for terminating when the list is sorted, and a penalty 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 not use this check in any way. A sorting algorithm is not required in order 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 are sorted with respect to each other after the swap. In other words, a reward was given for finding a sub-solution. Any other SWAP incurs a penalty of -10. This strategy 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

TABLE I

ACTIONS AND REWARDS USED IN RPSORT.

Action	Reward	Penalty	Description
NOOP	0	0	Initial ”last action” only.
TERMINATE	100	-100	End.
INCI	0	0	Increments i
INCJ	0	0	Increments j
INCK	0	0	Increments k
SETIZERO	0	0	Set $i = 0$
SETJZERO	0	0	Set $j = 0$
SETKZERO	0	0	Set $k = 0$
SWAP	10	-10	Swaps the values in $list[i]$ and $list[j]$

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.

When an action is executed, it causes a state transition, or a change in state. A state transition may lead to the creation of a new state, a state that has already been visited, or back to the same state (as defined by RP state equivalence). As state transitions are explored, the probability distribution for the state transitions is constructed, including adding new states over time.

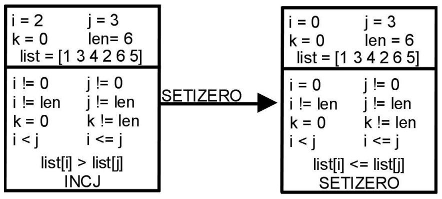

Fig. 2. An example state transition for RPSort.

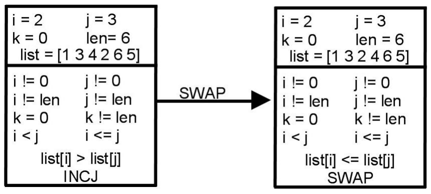

Fig. 3. SWAP modifies the training list.

Figure 2 demonstrates a state transition. The arrow indicates the transition, and the label on the arrow indicates the action performed. Recall that since the list is indexed starting at 0, i=2 references the third element in the list, and j=3 references the fourth. Figure 3 demonstrates the way an action can affect the training instance. Note that 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. As above, the arrow indicates a transition and the label above the arrow indicates the action.

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.

IV. 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.

TABLE II

SYMBOLS USED IN Q-LEARNING ALGORITHM.

$Q(s,a)$	current Q-value for state-action pair (s,a)
$\hat{Q}(s,a)$	updated Q-value for state-action pair (s,a)
α	a decaying weight, where $a_n = \dfrac{1}{1 + visits_n(s,a)}$
$visits_n(s,a)$	the number of times state-action pair (s,a) has been visited during learning
γ	discount factor
r_i	the i-th reward for performing action a in state s
$p(r_i\|s,a)$	the probability of receiving reward r_i when taking action a in state s
$p(s_j\|s,a)$	the probability of transitioning to state s_j from state s when taking action a
σ	a rate of decay on the effect of visits to a state-action pair
π_n	the policy executed by the system at time step n

RPSort uses episodic, dynamic, non-deterministic, delayed Q-Learning for training. 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. Table II defines the symbols used in the equations that describe policy execution and learning.

$$\pi^* = argmax_a(Q(s,a)) \quad (1)$$

(1) expresses policy execution mathematically: In a given state s, choose the action a with the highest q-value and execute it. As with other applications where there are goal states, policy execution stops when the system reaches a goal state, rather than continuing forever.

$$\hat{Q}(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha[\sum_i (p(r_i|s,a)r_i + \gamma \sum_j p(s_j|s,a) argmax_{a'} Q(s_j,a')] \quad (2)$$

(2) is used to update Q-values during learning. The version given here is a probabilistic form of the update equation, such as those given in [4].

Algorithm 1 gives the episodic training algorithm used for RPSort and other RP-based algorithms. Algorithm 2 is the learning/update algorithm for RP. This algorithm is an expansion of line 12 in Algorithm 1.

Algorithm 1: Episodic training algorithm for RP.

```
Let T be the full training set;
Let X be a subset of T;
while at-least-one-list-fails and $iteration < 100000$ do
    Add one (state, action, Q-value) triple, per action, to the
    Q-table for each list in the training set, if the triple does
    not exist;
    while $notMaxedOut = true$ do
        $iteration \leftarrow iteration + 1$;
        $notMaxedOut \leftarrow$ false;
        for each list L in X do
            initialize start state $s$ with $L$;
            learn (L,Q-table,...,);
        if $s.visits < 100000$ then
            $notMaxedOut \leftarrow$ true;
for each list L in X do
    Execute the current policy to see if it will sort the list
    correctly;
    if any L fails to sort then
        at-least-one-list-fails ← true;
if at-least-one-list-fails = false then
    for each list L in T do
        Execute the current policy to see if it will sort the list
        correctly.;
        if any L fails to sort then
            at-least-one-list-fails ← true;
if $iteration > 1000000$ then
    Indicate a failure to converge;
```

Because the algorithm dynamically adds states during learning, this algorithm is different than an algorithm that uses a fixed table 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 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.

Algorithm 2: The delayed Q-Learning algorithm for RP.

```
Let $X$ be a list from the training set;
$iteration \leftarrow 0$;
Initialize $s$ with $X$;
while ($s.lastAction \neq TERMINATE$) and
($iteration < 1000$) do
    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 $Q(s,a)$;
if $s.visits < \frac{MaxVisits}{2}$ then
    return;
$iteration \leftarrow 0$;
Initialize $s$ with list $X$;
while ($s.lastAction \neq TERMINATE$) and
($iteration < 2000$) do
    Choose action $a$ with highest Q-value ;
    $s' \leftarrow$ Execute $a$ on $s$;
    update $Q(s,a)$;
```

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, non-zero rewards were held constant, but γ and the penalties were varied. A γ value of 0.6 to 0.9 with a penalty of -160 to -180 leads to convergence the most often. That is, 90% to 95% of the time, convergence occurred within 200,000 iterations.

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 V.

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 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.

When it comes time to convert the learned policy 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. Not all states that are learned will be visited during execution of the program after training. This observation allows for some pruning and program simplification.

The pruning algorithm is straightforward. For each list in the training set, execute the policy, and mark each state that is visited. After all lists have been presented, delete all unvisited states. The remaining states constitute the pruned policy.

V. 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.

A. 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 90-95 percent of the time: the list [4, 3, 2, 1]. Longer lists generally came up with the same solution, with some minor random variations due to the non-deterministic nature of the technique.

B. Free Variables

Another interesting result was the lack of any use of the free variable k. Some solutions made use of it but only in a minor way. The most common result had no use of k at all. If we remove actions related to k from the system, the system learns faster, but the generated result is the same. We expect this result, if RPSort is working correctly.

C. The RP-generated Algorithm

Before pruning unnecessary states, the most commonly generated policy contains 226 states. Pruning reduces this to 17 states. Table III shows the pruned policy. Note that the ID column is not a part of the policy, it is for ease of reference only. In the comparative columns, such as "i vs. j", a symbol $=$ means "i equals j; a "$<$" means " $i < j$".

Algorithm 3 displays the program consistently produced by RPSort. The main while-loop keeps the policy executing until a goal state is reached–any state in which the last action is TERMINATE. 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.

Algorithm 3: RP-generated Sort Algorithm.

```
$list[] \leftarrow$ list to be sorted;
$len \leftarrow$ length of $list[]$;
$i \leftarrow 0$;
$j \leftarrow 0$;
$lastact \leftarrow NOOP$;
while $lastact \neq TERMINATE$ do
    if $j \neq 0$ then
        if $j \neq len$ then
            if $list[i] > list[j]$ then
                $lastact \leftarrow SWAP$;
                $swap(list, i, j)$;
            if $lastact = INCI$ then
                $lastact \leftarrow INCJ$;
                $j++$;
            else
                if $lastact = INCJ$ then
                    $lastact \leftarrow INCI$;
                    $i++$;
                else
                    if $lastact = (swap)$ then
                        $lastact \leftarrow (INCJ)$;
                        $j++$;
        else
            if $lastact = INCI$ then
                if $i \neq len$ then
                    $lastact \leftarrow (SETJZERO)$;
                    $j \leftarrow 0$;
                $lastact \leftarrow TERMINATE$;
            else
                if $lastact = INCJ$ then
                    if $i \neq 0$ then
                        $lastact \leftarrow INCI$;
                        $i++$;
                    else
                        $lastact \leftarrow SETJZERO$;
                        $j \leftarrow 0$;
    else
        if $i \neq 0$ then
            $lastact \leftarrow SETIZERO$;
            $i \leftarrow 0$;
        else
            $lastact \leftarrow INCJ$;
            $j \leftarrow 0$;
```

D. Proof that RPSort is a Generalized Algorithm

It is impossible to exhaustively check lists of all possible lengths in order 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

TABLE III
RPSORT POLICY.

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

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. Since 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 $j = len$, since 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. Since 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.

E. 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 1000 (at intervals of 100) and determining the average number of comparisons needed to sort the lists. Figure 4 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 non-trival lists, RPSort is superior to the other three algorithms.

F. 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. Since 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.

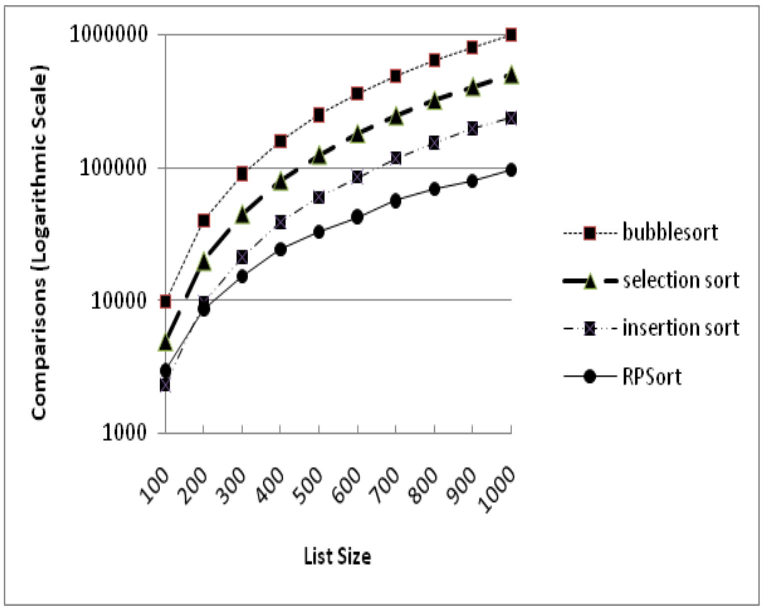

Fig. 4. RPSort compared to other iterative, in-place sorts.

G. 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 1000 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. Since 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. Lastly, 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 in order 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.

H. A note on Binary Addition Problems

The problems described in this section do not deal with RPSort, but outline how RP learned to solve three other problems. This indicates that the RP approach is capable of solving tasks beyond RPSort.

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-bit number. The system used five actions, TERMINATE plus setting the each of the outputs $true$ or $false$. 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. It is straightforward to show, by exhaustive testing, that the generated algorithm performs correctly.

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 number. Addition is effectively mod 2^n—the result of adding 1 to the n-bit number that is all 1's is all 0's, ignoring the overflow. 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 non-general solutions. More than 3 bits led to the same solution as that learned using 3 bits. The system is trained on all training instances during each iteration. The most commonly learned policy contains 254 states. Pruning trims this to 9 states.

RP generates a generalized incrementer. 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.

A general binary adder takes in two binary numbers of any length (number of digits) and returns a binary number representing their sum. The shorter of the two input numbers is padded with leading zeros so that the numbers have the same number of bits. The output number is also the same length as the inputs, again effectively addition mod 2^n. 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 system is trained on all training instances for each iteration. The most commonly learned policy contains 913 states. Pruning trims this to 48 states. As with the binary incrementer, there is no way to exhaustively test all possible inputs to determine algorithm correctness. However, by looking at the program structure, it is straightforward to see that the generated algorithm is a correct generalized binary adder. [15] gives the algorithm and a proof of this.

VI. CONCLUSIONS AND FUTURE WORK

This paper introduced Reinforcement Programming (RP) as an approach to automatically generate programs by using Reinforcement Learning (RL) techniques to generate an iterative, in-place sort algorithm. We compared these results to Kinnear's earlier work generating sorts with genetic programming (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 that 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. Experiments with RP showed that:

- A 4-element list is sufficient to learn an algorithm that will sort a list of any size.
- The smallest training set required for correct learning includes only one list, 90% of the time.
- Learning required less than 200,000 iterations, 90% of the time.

These results reduce the empirical complexity of learning an in-place iterative sort to about the same complexity as learning TicTacToe.

Potential future work with RP includes the following: 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; simulating Turing Complete languages; learning recursion; handling Partially Observable Markov Decision Processes; formalizing the process of modeling RP-states; using RL methods other than Q-Learning; and expanding the range of applications to real-world problems.

REFERENCES

[1] J. Kenneth E. Kinnear, "Evolving a sort: Lessons in genetic programming," in *Proceedings of the 1993 International Conference on Neural Networks*, vol. 2. San Francisco, USA: IEEE Press, 28 -1 1993, pp. 881–888.

[2] ——, "Generality and difficulty in genetic programming: Evolving a sort," in *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*, S. Forrest, Ed. University of Illinois at Urbana-Champaign: Morgan Kaufmann, 17-21 1993, pp. 287–294.

[3] C. J. Watkins, "Learning from delayed rewards," Ph.D. dissertation, Cambridge university, 1989.

[4] T. M. Mitchell, *Machine Learning*. New York: McGraw-Hill, 1997.

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

[6] J. H. Holland, *Adaptation in natural and artificial systems*. Ann Arbor, MI: The University of Michigan Press, 1975.

[7] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1989.

[8] C. M. Fonseca and P. J. Fleming, "Genetic algorithms for multiobjective optimization: Formulation, discussion and generalization," in *Genetic Algorithms: Proceedings of the Fifth International Conference*. Morgan Kaufmann, 1993, pp. 416–423.

[9] E. Nonas, "Optimising a rule based agent using a genetic algorithm," Department of Computer Science, King's College London, Tech. Rep. TR-98-07, April 1998.

[10] L. P. Kaelbling, M. L. Littman, and A. P. Moore, "Reinforcement learning: A survey," *Journal of Artificial Intelligence Research*, vol. 4, pp. 237–285, 1996.

[11] L. C. Baird, "Residual algorithms: Reinforcement learning with function approximation," in *International Conference on Machine Learning*, 1995, pp. 30–37.

[12] T. Jaakkola, S. P. Singh, and M. I. Jordan, "Reinforcement learning algorithm for partially observable Markov decision problems," in *Advances in Neural Information Processing Systems*, G. Tesauro, D. Touretzky, and T. Leen, Eds., vol. 7. The MIT Press, 1995, pp. 345–352.

[13] A. McGovern and A. G. Barto, "Automatic discovery of subgoals in reinforcement learning using diverse density," in *Proc. 18th International Conf. on Machine Learning*. Morgan Kaufmann, San Francisco, CA, 2001, pp. 361–368.

[14] R. S. Sutton, D. Precup, and S. P. Singh, "Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning," *Artificial Intelligence*, vol. 112, no. 1-2, pp. 181–211, 1999.

[15] S. K. White, "Reinforcement programming: A new technique in automatic algorithm development," Master's thesis, Brigham Young University, 2006.

[16] J. R. Koza, "Hierarchical genetic algorithms operating on populations of computer programs," in *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence IJCAI-89*, N. S. Sridharan, Ed., vol. 1. Morgan Kaufmann, 20-25 Aug. 1989, pp. 768–774.

[17] J. Koza, "Genetic programming: A paradigm for genetically breeding populations of computer programs to solve problems," Dept. of Computer Science, Stanford University, Technical Report STAN-CS-90-1314, Jun. 1990.